Research Paper:
Recovering the Structure of Software Systems Using Tube Graph Interconnection Clustering

Spiros Mancoridis
Dept. of Math. and Computer Science
Drexel University
3141 Chestnut Street
Philadelphia, PA, USA 19104
e-mail: smancori@mcs.drexel.edu
Phone: (215) 895-6824
FAX: (215) 895-1582

Richard C. Holt
Dept. of Computer Science
University of Toronto
10 King's College Road
Toronto, ON, Canada M5S 3G4
e-mail: holt@turing.toronto.edu
Phone: (416) 978-8726
FAX: (416) 971-2326

Abstract

An important product of the software design phase is the specification of the software structure at various levels of detail. Without reliable design documentation, significant software systems become less accessible to software engineers because structural information is buried in the intricate implementation source code. Reverse engineering techniques aim at recovering the structure of software systems, from the source code and mental models of developers, in order to make these systems more understandable to those maintaining them.

Many reverse engineering techniques rely on creating a decomposition hierarchy by recursively clustering related software components (*e.g.,* variables, procedures, classes, modules) into composite components (*e.g.,* subsystems). Component clustering is necessary for managing complexity, and therefore is an important step in the reverse engineering process. In this paper, we argue that the clustering of interconnections is also necessary during reverse engineering.

We propose an approach, based on a formalism called *tube graphs*, to specifying system structure that considers both composite software components and composite interconnections between these components. We present efficient algorithms for software interconnection clustering, which nicely complement existing software component clustering algorithms.

Keywords: Reverse Engineering, Design Extraction, Software Structure, Software Visualization, Clustering Algorithms, Formal Foundations of Software Engineering.

1 Motivation and Background

Software maintainers must understand a software system before they can modify, extend, and correct its functionality. Industrial-strength systems are difficult to understand because they are often, by their nature, large and complex. Moreover, because of their long term usage and maintenance, these systems may not have any current documentation other than the source code.

Understanding an industrial-strength system by examining its source code is an arduous task. Developers need high-level structural information to be able to navigate through the numerous software components and their interconnections. Without this structural information, developers may become overwhelmed by the amount of detail present in the source code. Figure 1 illustrates this point by showing the structure of the Linux operating system. The structure consists of approximately four hundred C source files (nodes) and the *include* connections (edges) between them. Although the diagram is complex beyond human comprehension, it is actually a fairly abstract view of the system structure. Namely, the view ignores details pertaining to functions and their invocation as well as variables and their references.

Research in reverse engineering [16] has resulted in tools that help software maintainers extract structural information from source code and, furthermore, create representations of the system structure at a higher level of abstraction than is available in the source code. While reverse enginering techniques can be used to redocument a software system [3], our focus is on applying reverse engineering techniques to recover design information in order to create software structure specifications of varying levels of detail.

An early step of a reverse engineering process is to mechanically extract information about the static source code components (*e.g.,* variables, procedures, classes, modules) and interconnections (*e.g.,* variable references, procedure and method invocations, class inheritance, module importation) using source code analysis. Based on these components and interconnections, as well as other information obtained from designers and the software documentation [1], related components of the system are clustered into composite components, called subsystems. Subsystems are then recursively clustered into higher level subsystems, and so on, until a single subsystem containing only the topmost subsystems is created. The clustering may be done manually or automatically, assuming that the clustering criteria can be stated formally. Most approaches such as Rigi [11], ARCH [15], and Star [10, 2] are semi-automatic and use a combination of both automatic and manual clustering. Reverse engineering is not necessarily a bottom-up process. In the *software reflexion models* [12] approach, engineers specify the software structure manually at any level of detail, and use tools to detect inconsistencies between the proposed software structure and the actual source code structure.

By partitioning software components into a hierarchy of subsystems, large systems become much more manageable. This point has been argued convincingly by researchers involved in Module Interconnection Languages [14, 5] and notations for software architecture [15, 4, 9, 7, 6]. Although it is commonly understood how to simplify system structure by clustering components, there is little consensus on how to cluster interconnections [13] to gain a similar simplification.

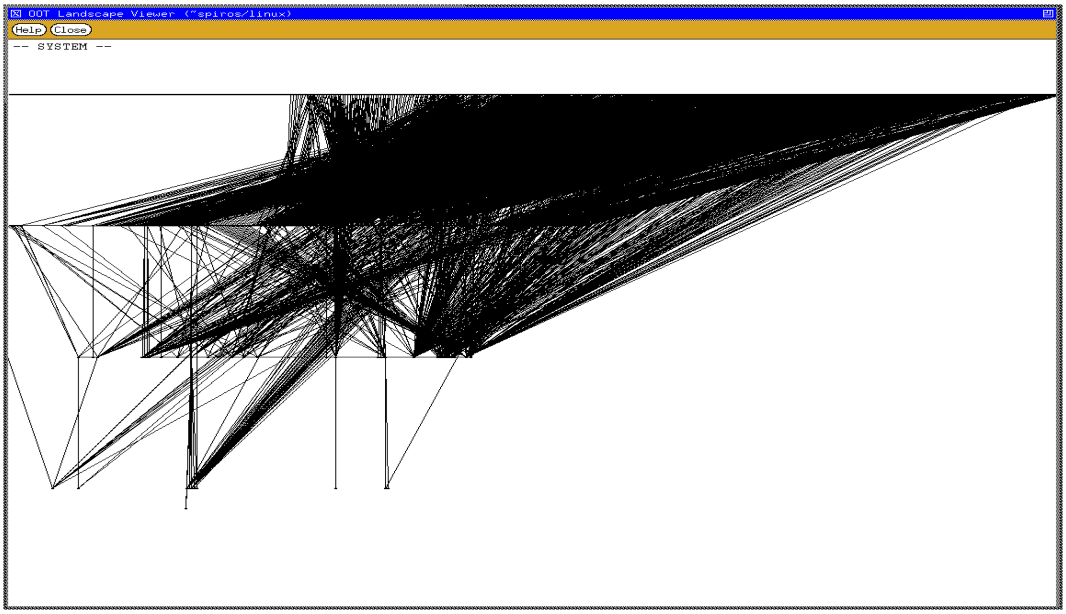

Figure 1: **Structure of Linux Before Component and Interconnection Clustering**

In this paper we introduce a formal structuring mechanism, called a *tube*, for specifying composite (clustered) interconnections between subsystems. We also present efficient algorithms to automatically cluster software interconnections.

Before proceeding with the formal characterization of tubes (Section 3), the description of the algorithms for clustering software interconnections (Section 4), and the implications of our work regarding future research in software structure specification and automatic design extraction (Section 5), we first give examples of composite components (subsystems) and composite interconnections (tubes).

2 Examples of Composite Components and Interconnections

Programs are structured using programming language components (*e.g.,* procedures, classes, modules) and interconnections (*e.g.,* calls, imports, exports, inherits). Additional structuring mechanisms to aggregate programming language components, such as subsystems, have been used for some time now. We propose a complementary structuring mechanism for aggregating interconnections, called tubes.

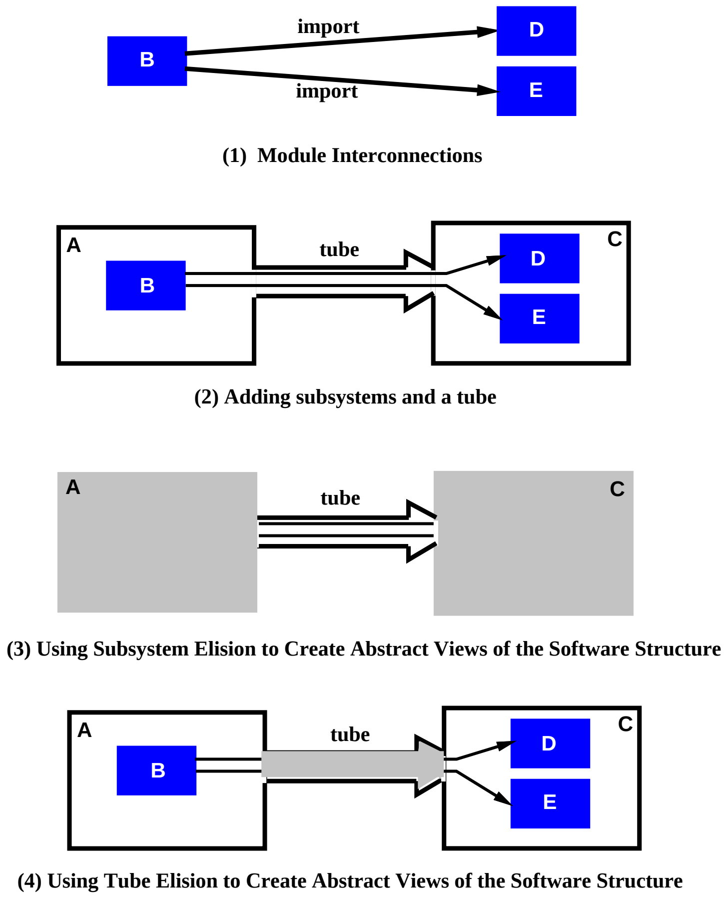

Figure 2: **Examples of Subsystems and Tubes**

Atomic Components and Interconnections

The first diagram of Figure 2 depicts the structure of a simple system, at the level of programming language constructs. This system consists of three modules. In this system, module B imports modules D and E. These imports are interconnections that enable components in B to depend on exported components in D and E. For example, procedures in B may invoke exported procedures in D and E.

Subsystems and Tubes

Using subsystems, designers can cluster related modules, as shown in the second diagram of Figure 2, where modules D and E are grouped into subsystem C and module B is grouped into subsystem A. The module interconnections, between module B and modules D and E,

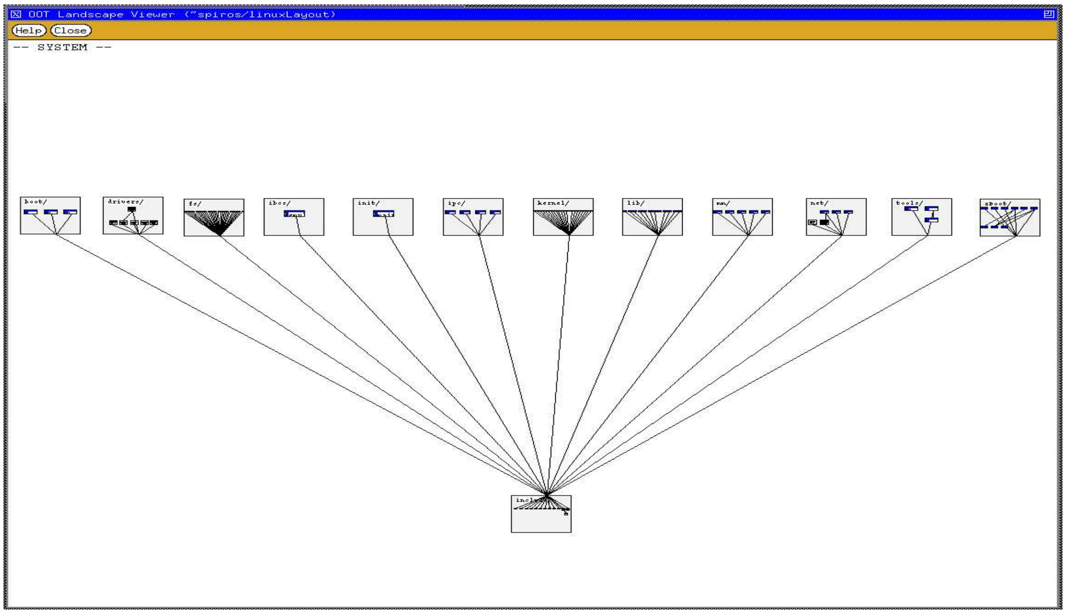

Figure 3: **Structure of Linux After Subsystem and Tube Clustering**

are clustered into a single tube, which is depicted by a large hollow arrow between subsystems A and C.

In top-down design, tubes may be used to express scoping constraints such as: module B should be allowed to import modules D and E, but modules D and E should not be allowed to import module B.

Elision of Subsystems and Tubes

Both subsystems and tubes may be elided, as shown in the third and fourth diagrams of Figure 2, in order to create more abstract views of software structures, and hence more meaningful visualizations. Elision techniques such as these enable engineers to view software structures at varying levels of detail. Moreover, they allow engineers to view certain components and interconnections in great detail, while hiding the complexity of parts of the structure without losing context.

The Structure of Linux Revisited

Figure 3 illustrates the structure of Linux after automatically clustering the C source files into a subsystem hierarchy and after clustering the *include* interconnections into tubes, using the Star system [10]. The subsystem clustering criterion was the file directory structure used to store the Linux source files. Note, the presence of tubes between subsystems and how they reduce the number of edges, and hence the clutter, in the diagram. If only subsystem clustering was used, without tube clustering, the same diagram would still be incomprehensible due to the number of *include* interconnections between the C source files.

Having given examples of, and motivated the usefulness of, subsystems and tubes, we proceed with a mathematical characterization of tubes and establish their formal relationship with subsystems.

3 Mathematical Characterization of Tubes

A *tube graph* consists of a set of vertices (V) and two sets of edges called the *containment* (C) edges and the *tube* (T) edges. The containment relation, which is a tree, represents the subsystem decomposition hierarchy. 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:

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

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

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:

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

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

□

Figure 4 shows the tube graph that corresponds to the software structure in the second diagram of Figure 2. In Figure 4, subsystem containment is shown as solid edges, and module interconnections as well as the tube are shown as dotted edges. In fact, all dotted edges are considered tubes because we treat module interconnections as atomic tubes.

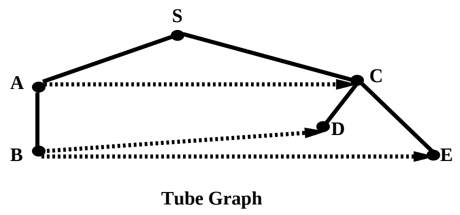

Figure 4: **Example of a Tube Graph**

This section defined a tube graph to be a containment tree with tubes between its vertices. We next discuss how tube graphs can be restricted to make them useful for specifying software structure.

3.1 Restricting Tube Edges

Criteria for automatic component clustering include pattern matching [15, 2], component coupling and cohesion metrics [11], and so on. Regardless of the clustering criteria, the result of component clustering is a subsystem decomposition hierarchy. Using component elision, the internals of subsystems may either be hidden or exposed depending on the kind of information the software engineer is looking for.

Similar to automatic component clustering, the criteria for automatic interconnection clustering vary. In our approach, the interconnection clustering criteria are closely tied to the restriction of tube graphs in various ways. More specifically, tube graphs are restricted so that interconnections between low-level components (modules) may occur only when permissions (tubes) between higher-level components (subsystems), exist.

The set of allowed permissions is determined by a set of rules, called *well-formedness rules.*

3.1.1 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 5 gives a visual representation of the set of allowed tubes. The labelled edges (C and T) represent edges in a tube graph. Each unlabelled dotted edge represents an allowed tube. 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 there is one particular tube graph TG of interest, we write T_A instead of $T_A(TG)$. Following is the formal definition of the set of allowed tubes:

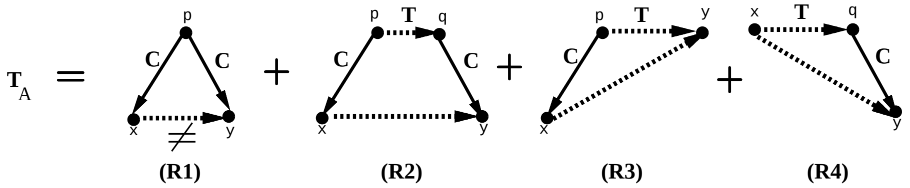

Figure 5: **A Visual Representation of Allowed Tubes, T_A**

Definition 3: Allowed Tubes

Given a tube graph $TG=(V,C,T)$, its set T_A of allowed tubes 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 to the graphs *R1* to *R4* in Figure 5.

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)

□

This concludes our formal definition of tubes and their relationship to subsystem hierarchies. In the following section we present algorithms that are useful when reverse engineering software systems using subsystems and tubes.

4 Algorithms for Clustering Software Interconnections

After the software components are clustered into subsystems, software engineers can use the algorithms presented next to cluster the interconnections between these components. When software structures are mapped into tube graphs, the problem of clustering interconnections is reduced to making the tube graph well-formed.

Figure 6 shows an example of how composite tube interconnections can be constructed from module interconnections and a subsystem decomposition hierarchy. The top of the figure shows an interconnection between modules *D* and *E*. According to the well-formedness rules, the corresponding tube graph (right) is ill-formed because the tube from *D* to *E* is ill-formed. This tube can be made well-formed by adding a tube from *B* to *E*, as shown in the middle of Figure 6. However, the corresponding tube graph (right) is still ill-formed because the newly added tube is ill-formed. This tube can become well-formed if a tube between *A* and *B* is added (bottom of Figure 6. The corresponding tube graph is now well-formed because all of its tubes are well-formed.

By making the tube graph well-formed, we have also succeeded in clustering the interconnections of the corresponding software structure. The benefits of interconnection clustering are not apparent from examining trivial software structures such as the one in Figure 6. However, the benefits become more obvious in larger systems such as Linux (Figure 3).

When specifying the complexity of our algorithms we will use the following definitions:

- |V| is the number of vertices in the tube graph,
- |T| is the number of tubes in the tube graph,
- *height* is the height of the tube graph containment tree, |V| in the worst case.

The worst case complexity of each of the following algorithms depends on the complexity of the primitive operations.[1] It has been shown elsewhere [8], that the worst case complexity of all primitive operations is constant O(1).

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. (Recall that well-formed tubes must satisfy rules *R1-R4*.) 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. That is, the tube satisfies rules *R1-R4* from Section 3.1.1. Note, that we used the notation *par(x)* to mean the parent of vertex *x*. 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
```

[1] These trivial algorithms, such as *isAncestor(x,y)* and *isRoot(x)* are not described in this paper.

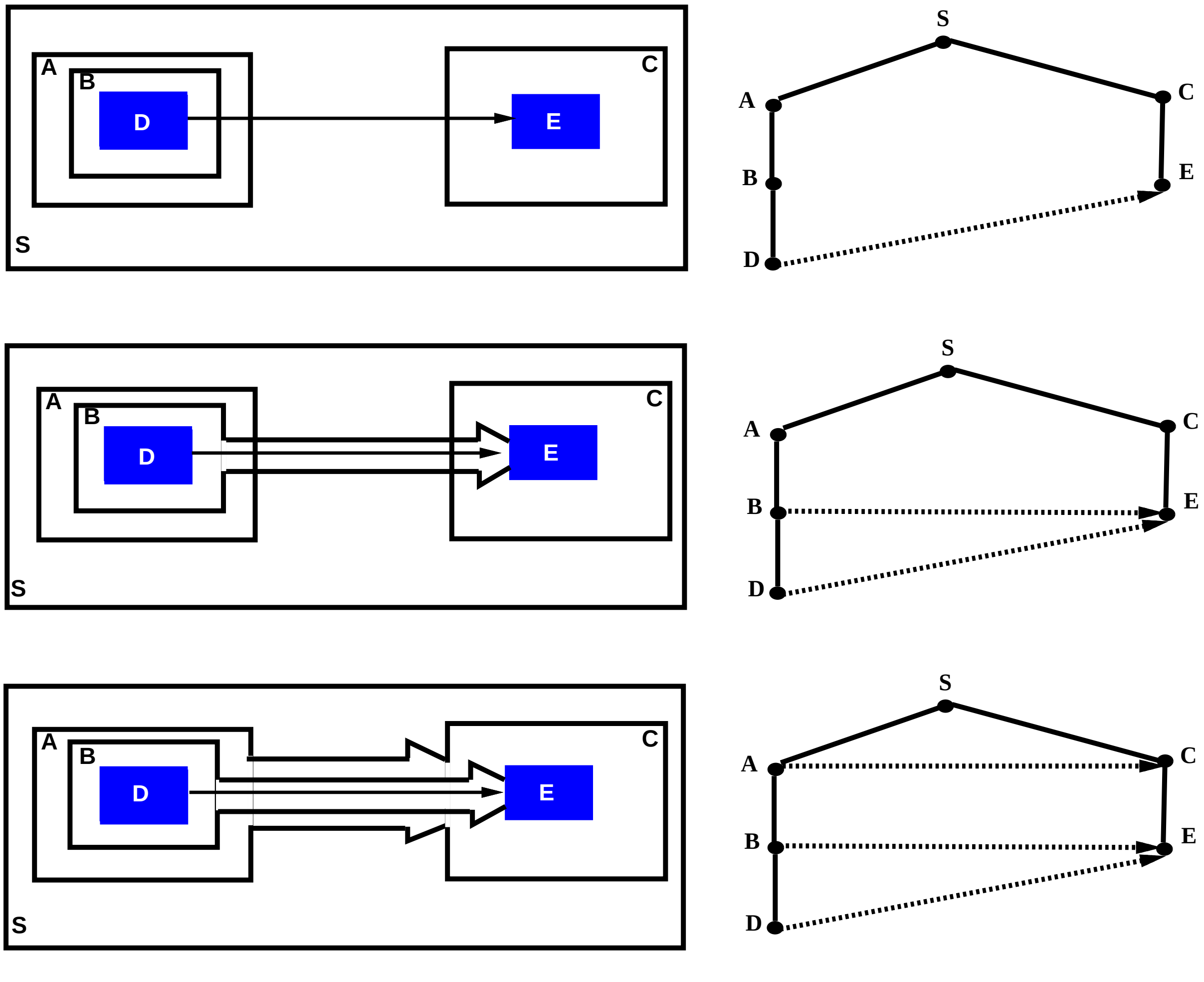

Figure 6: **Example of Tube Clustering**

□

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 each tube that is invalid or adds well-formed tubes to the tube graph so that each valid tube becomes 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 valid 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. (Note, lower levels are closer to the root.) In case 2) the algorithm adds tubes from x and to the ancestors of y (to the parent of y, then to the grandparent of y, and so on) until the added tube is at the same level as x. Case 3) is similar to case 2) except the added edges start from y. Then, we have reduced cases 2) and 3) to case 1), and 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)
                exit when level(x) = level(q)
                q ← par(q)
            end loop
        elsif level(x) > level(y) then
            loop
                addTube(p,y)
                exit when level(y) = level(p)
                p ← par(p)
            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. This algorithm should be applied to the subsystem decomposition hierarchy in order to cluster the module interconnections into tubes. 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
```

□

Each of the five algorithms has a place in the process of recovering the structure of a software system. Algorithm 1 is used to see if a connection can never become well-formed. Terminally ill-formed edges must be corrected by their removal from the tube graph. Algorithm 2 verifies that a given usage or connection is permitted by a higher level clustering connection. Algorithm 3 checks the entire structure to see that all connections are allowed.

Algorithm 4 creates higher-level tubes (clustered connections) for a given tube. Finally, Algorithm 5 performs interconnection clustering for the entire structure.

We conclude with the main contributions of this work and indications for future research opportunities.

5 Conclusions and Future Work

Hierarchical decompositions of large software systems enable developers to focus their attention on particular parts of these systems without being side-tracked by the details of other parts of the system. Most reverse engineering techniques create hierarchical decompositions based on composite software components. We propose that composite interconnections should also be part of these decompositions, given that the number of interconnections in a system almost always exceeds the number components. Hence, we propose the clustering of interconnections as a means of controlling the amount of detail associated with software interconnections.

The research contributions of our work include:

- the formal characterization of both composite components (subsystems) and composite interconnections (tubes),
- the development of efficient algorithms for automatically clustering interconnections, which complement existing algorithms for clustering components.

Our work suggests several opportunities for further research, namely the development of operations for manipulating tube graphs representing software structures. Manipulation operations on software structures include the addition and removal of modules or subsystems (addition and removal of a tube graph vertices or subtrees), the replacement of a subsystem with another subsystem (replacement of tube graph subtrees), the identification and removal unnecessary tubes, and so on. Also of interest is the effect of these operations on the well-formedness of tube graphs. For example, do these operations preserve the well-formedness of tube graphs? If not, should these operations also add new tubes in order to make tube graphs well-formed? Finally, the computational complexity of these operations is of interest.

Related research is currently being conducted to identify different kinds of tubes, as well as other kinds of well-formedness rules. These ideas will enable software designers to use software structures, such as those described in this paper, as a means of controlling the scope of software components [9].

References

[1] BIGGERSTAFF, T. J. Design Recovery for Maintenance and Reuse. *IEEE Computer 22*, 7 (July 1989), 36–49.

[2] CARMICHAEL, I., TZERPOS, V., AND HOLT, R. C. Design Maintenance: Unexpected Architectural Interactions (Experience Report). In *Proceedings of the International Conference on Software Maintenance* (1995), pp. 134–137.

[3] CHIKOFSKY, E. J., AND CROSS, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software 7*, 1 (January 1990), 13–17.

[4] DEAN, T. R. *Software Characterization Using Connectivity*. PhD thesis, Department of Computing and Information Science, Queen's University, 1993.

[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] GARLAN, D. First International Workshop on Architectures for Software Systems Workshop Summary. *Software Engineering Notes 20*, 3 (July 1995), 84–89.

[7] GARLAN, D., TICHY, W., AND PAULISCH, F. Summary of the Dagstuhl Workshop on Software Architecture. *Software Engineering Notes 20*, 3 (July 1995), 63–83.

[8] HOLT, R. C., AND MANCORIDIS, S. Using Tube Graphs to Model Architectural Designs of Software Systems. Tech. Rep. CSRI-308, Computer Science Research Institute, University of Toronto, October 1994.

[9] MANCORIDIS, S. *Controlling the Interactions of Architectural Design Components Using Scoping Rules*. PhD thesis, Department of Computer Science, University of Toronto, 1996.

[10] MANCORIDIS, S., AND HOLT, R. C. Extending Programming Environments to Support Architectural Design. In *Proceedings of the Seventh International IEEE Workshop on Computer-Aided Software Engineering* (July 1995), pp. 110–119.

[11] MÜLLER, H. A., AND KLASHINSKY, K. Rigi: A System for Programming-in-the-Large. In *Proceedings of the 10th IEEE International Conference on Software Engineering* (Singapore, 1988), pp. 11–15.

[12] MURPHY, G., NOTKIN, D., AND SULLIVAN, K. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'95)* (Washington, D.C., October 1995), pp. 18–28.

[13] PERRY, D. E. Software Interconnection Models. In *Proceedings of the 9th IEEE International Conference on Software Engineering* (Monterey, California, 1987), pp. 61–69.

[14] PRIETO-DIAZ, R., AND NEIGHBORS, J. M. Module Interconnection Languages. *The Journal of Systems and Software 6* (1986), 307–334.

[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] WATERS, R. C., AND CHIKOFSKY, E. J. Special Issue on Reverse Engineering. *Communications of the ACM 37*, 5 (May 1994), 23–93.