Repairing Software Style using Graph Grammars

Hoda Fahmy
University of Toronto
Toronto, Ontario, Canada
fahmyh@turing.toronto.edu

Ric Holt
University of Waterloo
Waterloo, Ontario, Canada
holt@plg.uwaterloo.ca

Spiros Mancoridis
Drexel University
Philadelphia, PA., U.S.A.
smancori@mcs.drexel.edu

Abstract

Often, software architects impose a particular style on the software systems they design. For large software systems, they would like to ensure that the design continues to conform to this style during the maintenance phase of the software-life cycle.

We will assume that the architectural design of a software system is available; for instance, it may have been extracted from the source code of the system using a parser. We will also assume we have a set of stylistic constraints given by the architect. For example, the architect may want to ensure that if a module X is allowed to use a procedure in a module Y, then module Y needs to export that procedure.

We define the Style Repair Problem as follows: If the current architectural design does not satisfy a set of stylistic constraints, how can we repair it so that it does? We choose to represent architectural designs as directed graphs; hence, repairing the style of these designs is equivalent to repairing the graph. We show how graph grammars can be used to automatically repair styles, and we show how this provides insight into the problem of style maintenance.

1 Introduction

With the development of large software systems came the need to describe and analyze the structure of these systems. Many researchers began using the term *software architecture*, yet such a term was and still is being used in different ways. Some researchers, such as Schwanke et al. [14] define it as the specification of the components of a software system and the permitted or allowed set of connections among these components. Others argue that there is more to software architecture than this; for instance, Garlan and Shaw argue that the principles and guidelines governing the design, should also be included in the definition of software architecture [5]. Throughout this paper, we refer to software architecture as the interconnection of components. This may also be referred to as *architectural design* of a software system.

In current practice, software architecture is commonly specified by means of diagrams accompanied by text. Most often, a component such as a file or procedure is represented as a box; nested boxes are used to indicate containment; labelled, directed, edges drawn between boxes are used to indicate relations (Figure 1(a)). Like Buxton [2], we choose to represent an architectural design by means of a directed graph. For instance, a node represents a source file, module, or subsystem; edges represent relations between files (Figure 1(b)). For example, if file B imports file H, we would have an "import"-labelled directed edge between the nodes representing files B and H. Section 2 describes the graph representation of software architecture which we assume in our work.

Designers commonly specify constraints on the interconnection of the components of a software system. For example, the designer may want to ensure that if a module X is to use a procedure in module Y, then module Y needs to export that procedure. Such constraints on the software architecture define an *architectural style* of the system. We note here that like software architecture, the term architectural style has a variety of meanings. The common link among these meanings is that an architectural style "encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their

relationships."[12] In Section 3, we provide some examples of architectural styles.

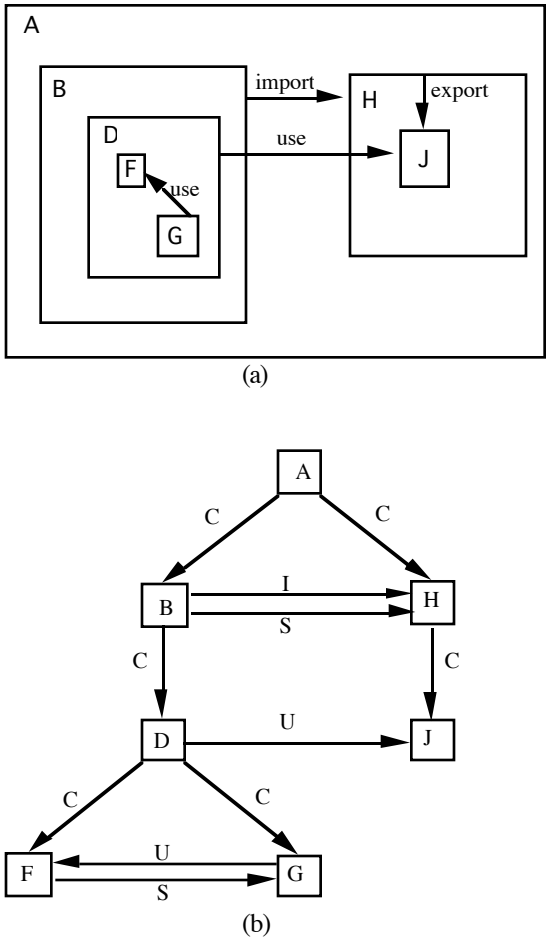


Figure 1: (a) Representing the software architecture of a system using nested boxes. (b) Representing the software architecture shown in (a) using a directed labelled graph. Legend of edge labels: C=Child, S=Sibling, U=Use, I=Import, E=Export. To avoid cluttering the graph, we do not show all the Sibling relations; assume that if there is an S-labelled edge from node A to node B, then there is one from node B to node A as well.

Given a software design represented as a graph, we can inspect the graph to determine whether the design conforms to a particular architectural style. In case it does not, it would be beneficial if we could automate the process by which the system is "repaired" so as to conform to the style. We refer to this as the Style Repair Problem. The Style Repair Problem is presented in detail in Section 4.

In this paper, we show how a graph grammar can help perform style repair. We assume we have a graph representing an existing software system which may or may not conform to a particular architectural style. We apply the graph-rewriting rules of a graph grammar. When these are applied to a system not conforming to the style, they will "repair" it so that it does. In most cases, there is more than one possible repair for a given graph, and some repairs are better than others. Sections 5 and 6 describe how we use graph grammars for style repair. Conclusions and future work are given in Section 7.

Although most software engineers are not familiar with graph grammars, there is a growing interest in using graph grammars in software engineering. For instance, Wills [16] uses graph grammars to recognize standard computational structures (referred to as clichés) in a program. Also, Dean and Cordy [3] introduce a diagrammatic syntax for expressing software architectures and provide a taxonomy of architectures characterized as sets of patterns in their language which can be recognized by graph grammars. More related to our work is the work recently reported by Le Métayer [10]. He uses graph grammars to model architectural styles. Although our motivation is very similar, we define architectural styles quite differently. In Le Métayer's work, an architectural style is represented as a graph grammar and the allowable actions of the "coordinator" are modelled by conditional graph-rewrite rules. (The coordinator is in charge of managing the software architecture.) His approach is applied to architectural styles such as the client-server architecture style.

In short, graph grammars are becoming a promising tool in software engineering and specifically software architecture.

2 Graph Representation of Software Architecture

In this work, we assume we have a *hierarchical* system such as the one shown in Figure 1. Each node represents a component in the system; the

containment information of the system is given by a *containment tree* defined by the *Contain* or *Child* (C) relation (Figure 1(b)). With the Child relation, we define the *Parent* (P) and *Sibling* (S) relations. If there is a Child relation between nodes A and B, then there is a Parent relation between nodes B and A. Nodes with the same parent are siblings. If nodes A and B are siblings, there is a directed S-labelled edge between nodes A and B, and a directed S-labelled edge between nodes B and A. Note that in Figure 1(b), we do not show the Parent relations nor do we show all the Sibling relations since these relations are implied by the other relations shown.

Besides the containment relations (C, P, and S), there are non-containment relations. There are dependency relations between components such as Import (I), Export (E) and Use (U). These relations are also modelled by directed labelled edges as shown in Figure 1(b). As we shall see later in this paper, the rules of a particular architectural style impose constraints on these non-containment relations.

There may be multiple relations between components. For instance, component A may contain component B and at the same time, component A may export component B. This can be modelled as compound edge labels, where each label describes the list of types of relations. It can also be modelled as multiple edges. We have chosen to model this using multiple edges; thus the graph is *non-simple*.

We assume that the nodes of the graph representing a software design are labelled and attributed. The label and attributes encode the relevant information of a particular component. We use one node label, namely, "component". The attributes for the node encode the name of the component as well as the *level* or *depth* of the component in the containment tree. For example, the level of component A in Figure 1 is 0; the level of component G is 3.

3 Examples of Architectural Style

To illustrate what we mean by architectural style, here we describe four architectural styles:

Style 1: Import/Export Style
Style 2: Selective Import/Export Style
Style 3: Tube Style
Style 4: Buyer/Seller Export Style

These styles were identified and defined by Holt [7] and Mancoridis [11]. What follows is a summary of the constraints of these styles.

In the Import/Export Style, a component may export, import and use another component. X can export Y if Y is X's child. X can import Y if they are siblings or if X's parent imports Y's parent and Y's parent exports Y. X can use Y if they are siblings or if Y is an exported item (any number of levels of export) of X's sibling or of X's parent's imports. This style, as well as the next, is much like that used in various module interconnection languages (MILs) [13], as well as in languages such as Java [6] and Object-Oriented Turing [8].

The Selective Import/Export Style is similar to the first style. X can export Y if Y is a child of X or an exported descendent of X. X can import Y if they are siblings or if Y is an exported item (any number of levels of export) of X's sibling or of X's parent's imports. The conditions for which X can use Y are the same as for X importing Y, and hence usage of a component can be considered to be a special case of importing.

The tube style [9] consists of a single relation - the tube, which represents the combined concepts of importing, exporting and usage. In this style, a component can have a tube to its siblings, to components its parents have a tube to, to the children of components it has a tube to, and to the children of components its parent has a tube to. Because this style is quite simple, we will use this style to illustrate how a graph grammar can be used to perform style repair (Section 6).

The Buyer/Seller Export Style supports two types of exporting: export to buy, and export to sell. It also allows a component to use another component. This style has no concept of importing. X can export to buy Y if Y is a child of X or if Y is an exported (to buy) item of X's child. X can export to sell Y if Y is a child of X or if Y is an exported (to sell) item of X's child. X can use Y if Y is a sibling of any of X's ancestors so long as the ancestor exports to buy X. X can also use Y if Y is an exported (to sell) item of a sibling of X's ancestor so long as the ancestor exports to buy X.

Holt [7] illustrates how to use binary relational algebra (as defined by Tarski [15] and refined by Schmidt and others) to model these stylistic constraints. Mancoridis [11] shows how they can be formalized in a visual notation called

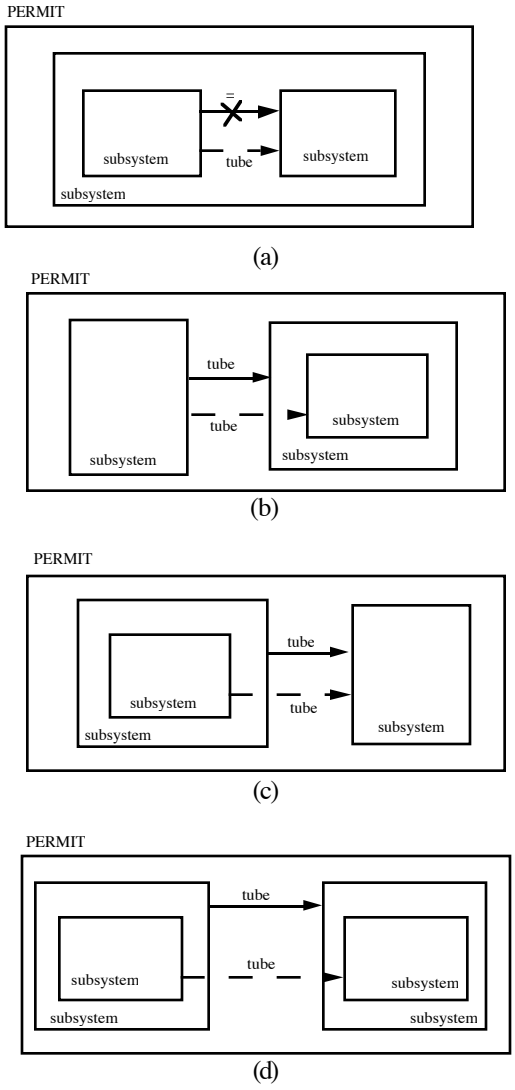


Figure 2: Mancoridis' visual notation for the Tube Style Constraints. A component can have a tube to:

(a) its siblings,
(b) the children of components it has a tube to,
(c) components its parents have a tube to, and
(d) the children of components its parent has a tube to.

ISF (Interconnection Style Formalism). Figure 2 illustrates Mancoridis' visual notation for the Tube Style. In this figure, the dotted lines represent edges that can be drawn if the solid lines are present.

Now that we have provided examples of architectural styles, we now turn to the problem of style repair.

4 The Style Repair Problem

The Style Repair Problem is defined as follows: how can we modify the architectural design of a software system to make it satisfy the constraints imposed by the designer? We choose to represent the architectural design as a directed graph (Section 2); hence repairing the style is equivalent to repairing the graph. The output is a "repaired" graph that conforms to the style; see Figure 3. In other words, the input graph undergoes transformations until it reaches a point where it becomes consistent with the constraints imposed by the architectural style.

The style repair algorithm produces a graph which represents a *legal* system. A system (or graph) is legal if all relations satisfy all the rules (i.e., all relations are legal). The system is *illegal* if at least one relation does not satisfy a constraint. Let us consider the following rule:

> *Module P can use module Q if P and Q are contained in the same subsystem of if module P's parent imports Q's parent and module Q's parent exports Q.*

If module P uses module Q yet they are not siblings and Q's parent does not export Q, then we consider that relation to be *illegal*.

In general, an illegal system can be repaired in a variety of ways. For instance, illegal relations may simply be deleted or the containment information may be altered. In our work, we place the following restrictions on our style repair algorithm:

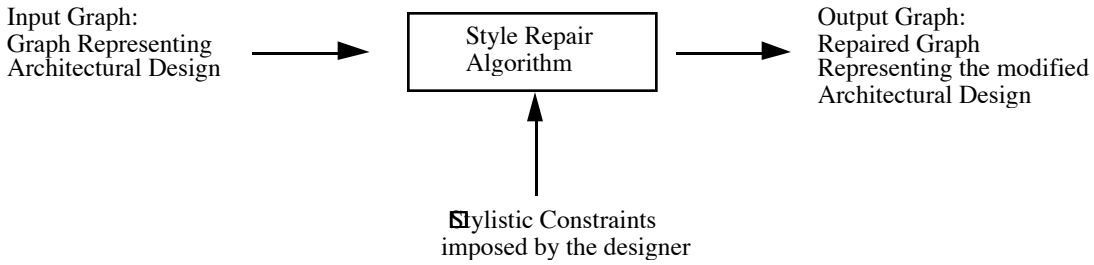


Figure 3: The input to the style repair algorithm is a graph representing the architectural design of a piece of software. It may or may not conform to the stylistic constraints imposed by the designer. How this input graph is derived is not a concern for our purposes; it may have been extracted from the source code by a parser, or it may simply be a sketch drawn by a software architect. The output of the algorithm is a "repaired" graph in which all stylistic constraints are satisfied.

(1) The style repair algorithm must not modify the containment tree of the input graph. In other words, (i) a node exists in the output graph if and only if it exists in the input graph, and (ii) there is a Child, Parent, or Sibling relation between nodes X and Y in the output graph if and only if there is a corresponding one in the input graph. The input and output graphs differ only in the non-containment relations.

(2) The style repair algorithm must not delete relations unless it is *irreparable*. An irreparable relation is an illegal non-containment relation which can never be legal.

(3) The algorithm can add only legal and *repairable* relations to the system. A *repairable* relation is an illegal non-containment relation which can be made legal by adding other non-containment relations to the system. These added relations must be either legal or repairable relations. In the example rule above, if module P uses module Q yet the constraint is not satisfied then we can satisfy the constraint by adding an import relation between P's parent and Q's parent, and/or an export relation between Q's parent and Q provided that such relations are not irreparable illegal relations.

Having defined the Style Repair Problem and having given the assumptions that our style repair algorithm makes, we now turn to a description of our graph grammar approach to style repair.

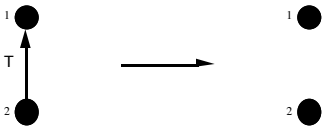
5 Graph Grammars and their Application to Style Repair

A *graph grammar* (or graph rewriting system) is a formalism in which the transformation of graph structures can be modelled and studied. There is an extensive literature on graph grammars; see [1] for an overview. A graph grammar is specified by a set of graph-rewrite rules, where the role of each rule is to replace one subgraph (given in the left-hand side of the rule) by another (given in the right-hand side of the rule). Specifically, each application of a graph-rewrite rule must (1) **identify** some pattern, and then (2) **transform** the host graph in some way based on that pattern.

In the rest of this section, we will show how graph grammars can be used to solve the Style Repair Problem. Systems are represented as graphs and the goal of a style repair algorithm is to (1) **identify** illegal relations, and then (2)

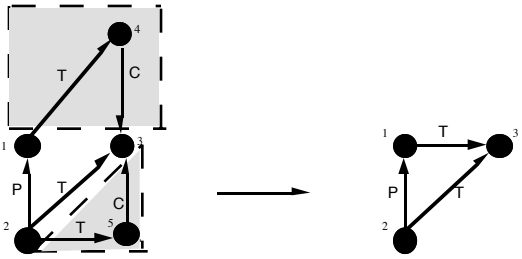
transform the graphs in such a way that leads to legal systems. What better way to specify the transformation of the input graphs than via graph-rewrite rules?

We will now introduce the features of graph grammars using examples from the tube style repair grammar. (The complete grammar is given in Figure 7 of Section 6.) Figure 4 shows two rules (T2 and T4), one which deletes an irreparable relation, and the other which repairs a repairable relation by adding a tube relation. Note that in the tube style, a tube from X to Y is irreparable if X and Y are the same component or X is the ancestor of Y or Y is the ancestor of X.



COND = Ancestor(1,2) or Ancestor (2,1)

(a) Rule T2



COND = ~Ancestor(1,3) & ~Ancestor(3,1) & ~Tube(1,3)

(b) Rule T4

Figure 4: Two sample graph rewrite rules from the tube style repair grammar. COND contains boolean functions, Ancestor(x,y) and Tube(x,y). The function Ancestor(x,y) returns true if there is a chain of Child-labelled edges between the nodes denoted x and y. The function Tube(x,y) returns true if there is a tube-labelled directed edge between nodes x and. y.

Rule T2, shown in Figure 4(a), illustrates a simple graph-rewrite rule. The rule states that if there is a tube between two distinct nodes, node 1 and node 2, and node 1 is either an ancestor or a descendant of node 2, then eliminate that tube; the tube is an irreparable illegal tube and cannot exist in a legal system. The "COND" is a boolean expression which must evaluate to true in order for the rule to be applied.

Rule T4, shown in Figure 4(b), illustrates a more complex rule. The rule models the repair of the following stylistic constraint: If there is a tube between X and Y then

(a) X and Y are siblings, or
(b) there is a tube from X's parent to Y, or
(c) there is a tube from X to Y's parent, or
(d) there is a tube from X's parent to Y's parent.

A tube requires repair if *none* of the conditions (a)-(d) given above are satisfied. Rule T4 states that if there is a tube between nodes 2 and 3, then add a tube between node 2's parent and 3 if one does not exist already. In other words, repair the graph such that condition (b) is satisfied. In this rule, COND states explicitly that node 2's parent cannot be the ancestor or descendant of node 3. (Implicitly, it states that nodes 2 and 3 are not siblings given that node 1 is the parent of node 2. Hence condition (a) above is not satisfied. If the nodes were siblings, then the tube between nodes 2 and 3 would not require repair.) COND must be satisfied or else the added tube may be an irreparable illegal tube. In other words, COND ensures that only legal or repairable illegal relations are added. Each shaded region (referred to as *conditional prohibited context* [4]) in the left-hand side of the rule specifies that if the graph in the shaded region exists in the context of the matched subgraph, then the rule cannot be applied. Here, we use the conditional prohibited context to ensure that indeed the tube being considered for repair requires repair. The top shaded region ensures that condition (d) given above is not satisfied; the bottom shaded region ensures that condition (c) is not satisfied. When we have more than one shaded region, the rule can be applied only if none of the graphs shown within the shaded region can be found in the context.

Figure 5 illustrates examples of graph-rewrite rule application, where Figure 5(a) shows the host graph to which the rules are applied. It contains one irreparable tube (between nodes 1 and 4) and two repairable tubes (between nodes 7 and 5 and between nodes 4 and 3). Figure 5(b) shows the resultant graph after applying the rule T2; the tube between nodes 1 and 4 is deleted. Figure 5(c) shows the resultant graph after

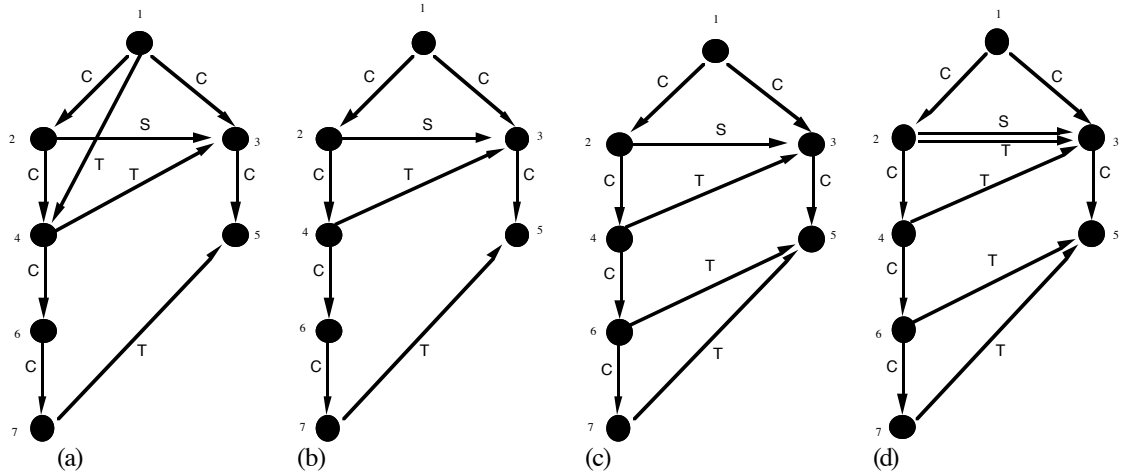


Figure 5: Examples of graph-rewrite rule application. The graph in (a) is the initial host graph. It contains one irreparable illegal tube, and two repairable illegal tubes. The graph shown in (b) is the result of applying rule T2 to the graph in (a). The graph shown in (c) is the result of applying T4 to the graph in (b). The graph shown in (d) is the result of applying T4 again to the graph shown in (c).

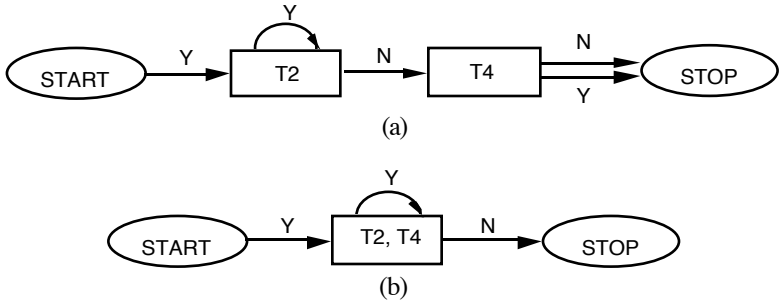


Figure 6: Examples of control diagrams.

applying the rule T4 to the graph in Figure 5(b); the tube between nodes 7 and 5 is repaired by adding a tube between nodes 6 and 5. Figure 5(d) shows the result of applying the rule T4 to the graph in Figure 5(c); a tube between nodes 2 and 3 is added to repair the tube between nodes 4 and 3. Note that due to the conditional prohibited context of rule T4, T4 cannot be applied to the tube between nodes 6 and 5. Because the graph shown in Figure 5(d) contains no illegal tubes, it represents a legal system.

Our approach to style repair relies on the ability to order the application of the graph-rewrite rules. We use a *control diagram* which specifies which set of rules should be tried next, conditional on whether the current rule was

applied successfully or unsuccessfully (Figure 6). The control diagram in Figure 6(a) states: apply rule T2 as often as possible, then apply T4 at most once. Given this control diagram, the rules T2 and T4 (Figure 4), and the initial host graph shown in Figure 5(a), a possible output graph is that shown in Figure 5(c). The control diagram shown in Figure 6(b) states: apply rule T2 and T4 in any order and as often as possible. Given this control diagram, the rules T2 and T4 (Figure 4), and the initial host graph shown in Figure5(a), a possible output graph is that shown in Figure 5(d). Note that no matter how the rules T2 and T4 are applied, the only output graph in this case is the one shown in Figure 5(d). This is not the case in general. Given an initial graph, a set of rules, and a control diagram, more than one output graph may be produced. This issue is discussed in detail in the next section.

We have used rules T2 and T4 to introduce the features of graph grammars, graph-rewrite rule application and control diagrams. These rules are only two of the five rules comprising the tube style repair grammar that is described in the next section.

6 The Tube Style Repair Grammar: A Case Study

In determining the appropriateness of using graph-rewriting rules to perform style repair, we have developed four graph grammars, each of which repairs one of the architectural styles presented in Section 3. To test the grammars, we have implemented and executed them on graphs requiring repair. The output graphs produced were legal. Specifically, all irreparable relations were deleted and legal relations were added to repair the repairable relations in the input graph.

This section uses the tube style, the simplest of the four styles, to show how graph grammars can perform style repair. This grammar consists of five rules: Rules T1 and T2 eliminate irreparable illegal tubes, while Rules T3-T5 repair repairable illegal tubes (Figure 7). (Note that rules T2 and T4 are also given in Figure 4 of Section 5.) Figure 8 shows possible output graphs of applying the graph grammar to the input graph shown in Figure 8(a). The fact that the graph grammar may produce more than one output graph for a given input graph is discussed in Section 6.3. The tube style repair grammar is used to highlight some issues such as efficiency (Section 6.1), termination (Section 6.2), non-determinism (Section 6.3), and optimal repair (Section 6.4).

6.1 Efficiency

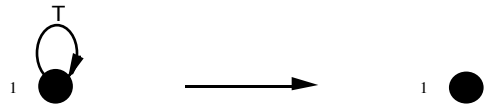
Subgraph matching is a potentially computationally expensive operation. Even after spending time to find a match, there is no guarantee that rule application will occur because COND may not be satisfied, or the conditional prohibited context may disallow the application. To reduce the chance of this happening, we have ordered the rules such that all irreparable relations are eliminated first before attempting to repair the repairable ones; refer to the control diagram in Figure 7(f). In doing this, we avoid wasting time trying to repair irreparable tubes. For the tube style repair grammar, this means that we apply rules T1 and T2 in any order as often as possible, and then we apply rules T3-T5 in any order and as often as possible. Since we guarantee, mainly by the COND, that the tube we add to repair an existing repairable tube is not an irreparable tube, we need not reapply the rules T1 and T2 upon successful application of any of the rules T3-T5. Note that the output graphs produced by this ordering are the same as those produced had we specified that rules T1-T5 are applied in any order and as often as possible.

6.2 Termination

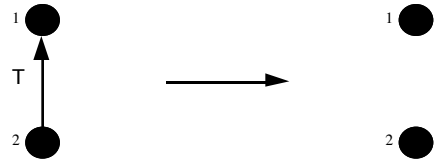
It is important to guarantee that the graph grammar system terminates. Termination can be proven since the tube style repair grammar has the following properties:

(1) An irreparable tube is never added.
(2) A legal tube is never deleted.
(3) A repairable tube is never deleted.
(4) Each graph-rewrite rule specifies:
    (i) the deletion of an irreparable tube (of which there is a finite number), or
    (ii) the conversion of a repairable tube to a legal one as well as the addition of either a legal or repairable tube.

Rule application continues so long as there exists irreparable or repairable tubes. Given the above properties and the fact that there can only be a finite number of legal tubes added to the graph, the grammar must terminate.

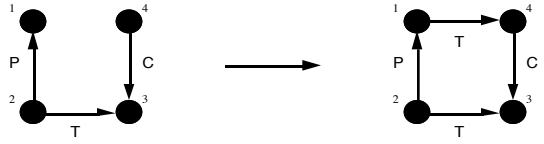


(a) T1: deletes an irreparable illegal tube



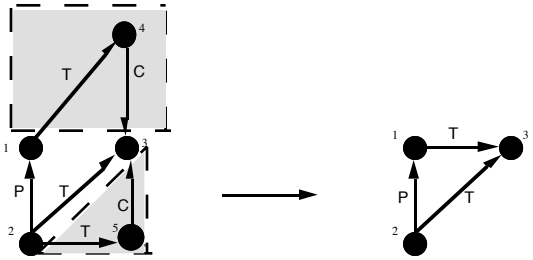
COND = Ancestor(1,2) or Ancestor (2,1)

(b) T2: deletes an irreparable illegal tube



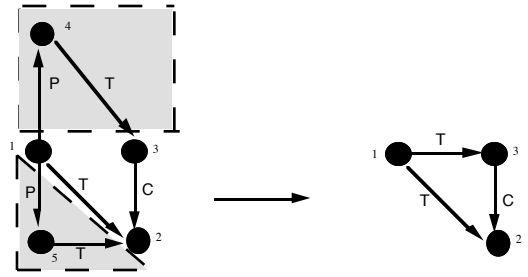
COND = ~Ancestor(1,4) & ~Ancestor(4,1) &
~Tube(1,3) & ~Tube(2,4) & ~Tube(1,4)

(c) T3: repairs a repairable illegal tube



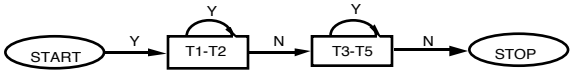
COND = ~Ancestor(1,3) & ~Ancestor(3,1)&
~Tube(1,3)

(d) T4: repairs a repairable illegal tube



COND = ~Ancestor(1,3) & ~Ancestor(3,1) &
~Tube(1,3)

(e) T4: repairs a repairable illegal tube



(f) The control diagram

Figure 7: Here we show the 5 rules ((a)-(e))comprising the graph grammar that performs Tube Style repair, as well as the control diagram (f).

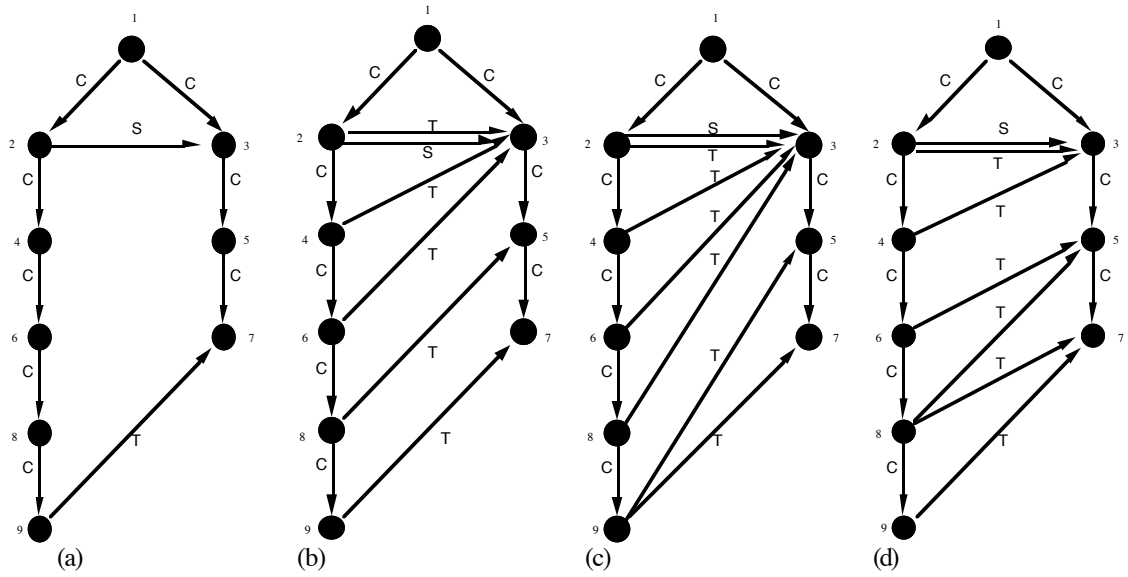


Figure 8: The graph in (a) is the input graph to which the grammar in Figure 7 is applied. It contains one repairable illegal tube. The graphs shown in (b), (c), and (d) are possible output graphs. Note that in all output graphs, there contains no (repairable) illegal tubes. The sequence of rules applied to the graph in (a) that results in the graph shown in (b) is: T3, T3, T4, and T4; the sequence of rules applied to the graph in (a) that results in the graph shown in (c) is: T5, T3, T4, T4, and T4; the sequence of rules applied to the graph in (a) that results in the graph shown in (d) is: T4, T5, T4, T3, and T4.

6.3 Non-determinism

Since there may be more than one way to repair an input graph, the graph grammar may produce various output graphs. Thus, the output is non-deterministic. In all of the output graphs, the same irreparable illegal relations are deleted from the graph. They differ in the added (non-containment) relations which are required to make the repairable illegal relations legal.

Referring to the tube style repair grammar, there are three ways (represented by the rules T3, T4, and T5) to repair a repairable illegal tube. For a given repairable illegal tube, at least one of the three rules will apply. This allows a tube to be repaired in more than one way. Additionally, the successful application of a repair rule may repair more than one repairable illegal tube. If we apply the tube style repair grammar to the graph shown in Figure 8(a), various output graphs can be produced. Figures 8(b)-8(d) illustrate three such graphs. Each output graph is guaranteed to contain no illegal tubes - repairable or irreparable.

6.4 Optimal Repair

If there can be more than one possible repair, it seems appropriate to define an *optimal repair*. There is more than one way to define optimality and most likely, it should be the software architect who would define it. Currently, we have defined it as one which causes the least amount of change to the input graph. In other words, we assume that the optimal repair is a *minimal repair*. Since all approaches delete the same irreparable illegal relations, the optimal repair is that which adds the least number of edges to the graph in order to make the repairable illegal relations legal. The number of added edges seems to be an adequate measure of how good a repair is, but it assumes that all relations are equal. For example, adding an import-

labelled edge is considered to be equivalent to adding an export-labelled edge.

As shown in Figure 8(d), even though COND and the conditional prohibited context constructs guarantee that indeed the tube being repaired requires repair, the grammar may still add *gratuitous* tubes. A gratuitous tube in a legal graph is one which can be deleted without causing the graph to become illegal. The graphs shown in Figure 8(b) and 8(c) contain no gratuitous tubes. Since the graph shown in Figure 8(b) had the least number of edges added to it, then given the way we have defined an optimal repair, this repair represents the best of the three repairs.

6.5 Summary

In this section, we have used the tube style repair grammar to illustrate how graph grammars can be used for style repair and we have also discussed various related issues. Section 6.1 discussed how we can order the graph-rewrite rules in order to make the grammar more efficient. The ordering of the rules is non-deterministic in that we can specify sets of rules (represented by rectangular nodes in a control diagram) whereby the rules in each set are applied repeatedly, in any order, until none of the set's rules apply. Regardless of the ordering of a set's rules, we can show that the grammar terminates. There may be more than one output graph which the grammar produces given an input graph. Since the grammar may repair a graph in more than one way, it is appropriate to define an optimal repair. We have defined it as one which causes the least amount of change to the input graph.

It is worth mentioning here that Holt and Mancoridis [9] have developed a linear-time tube style repair algorithm. Although there may be more than one way to repair the input graph, the algorithm produces one repair which is not necessarily minimal.

7 Conclusions and Future Work

For large software systems, there is a need to describe the interconnection of system components. We refer to this as the software architecture or the architectural design of the system and we can represent it as a directed graph (Section 2). Often system designers impose constraints on the design; these constraints comprise the architectural style of the system. Examples of architectural style were given in Section 3. If the architectural design of a system does not conform to the constraints imposed by the designers, how can we modify it such that it does? This is the Style Repair Problem which was presented in Section 4. In this paper, we proposed a graph grammar formalism which automates style repair (Sections 5 and 6). We have developed four graph grammars, each of which encodes the constraints of one of the following architectural styles: (1) import/export style, (2) selective import/export style, (3) tube style, and (4) buyer/seller export style. This paper focussed on the simplest of these grammars, the tube style repair grammar (Figure 7), which was used to discuss various issues such as termination and non-determinism.

To test the graph grammars, we have implemented and executed them using simulated illegal graphs as input. The grammars repaired the graphs such that the stylistic constraints were satisfied. In the future we plan to apply the grammars to existing software systems.

In our current approach, relations are only added to the system to make it legal; relations are only deleted if they are irreparable. The approach assumes that the containment information of the system remains unchanged during the repair. In the future, we plan to explore the possibility of allowing the containment information to change in the course of repair. In other words, in trying to obtain an optimal repair, the style repair algorithm may provide a "better" breakdown of the system.

The Style Repair Problem is a newly defined problem and we have introduced a novel approach involving graph grammars. Our experiments with graph grammars have facilitated the exploration of many style repair issues. We do not claim that our method is an efficient solution to the style repair problem, but rather it is an elegant one. In a sense, the graph-rewrite rules act as high-level, executable specifications for the implementation of style repair.

Acknowledgements

This work has been made possible by the first author's NSERC Postdoctoral Fellowship. This work has been supported in part by CSER (Consortium for Software Engineering Research)

and by ITRC (Information Technology Research Centre).

About the Authors

Dr. Hoda Fahmy is a Post-doctoral fellow in the Department of Computer Science at the University of Toronto. Her research interests include graph grammars and their application to various fields such as software engineering and document analysis and recognition. She received her Ph.D. degree in Computer Science from Queen's University in 1995. Hoda Fahmy is currently on leave from I.B.M. (Toronto Laboratory).

Dr. Ric Holt is a professor at the University of Waterloo and an adjunct professor at the University of Toronto in the Department of Computer Science. Holt received his Ph.D. degree in Computer Science from Cornell University in 1971. His research has included work in operating systems, compiler development, and software construction methods. He is an author of the Turing programing language. He recent work has concentrated on Software Landscapes, which provide a visual formalism for software architectures.

Dr. Spiros Mancoridis is an Assistant Professor in the Department of Mathematics and Computer Science at Drexel University in Philadelphia, USA. He received his Ph.D. degree in Computer Science from the University of Toronto in 1996. His research interests include: reverse engineering, module interconnection formalisms, and software visualization.

References

[1] Blostein, D., Fahmy, H., and Grbavec, A. "Issues in the Practical Use of Graph Rewriting." *Lecture Notes in Computer Science*, Vol. 1073, 1996, pp.38-55.

[2] Buxton, J. and McDermid, J. "Architectural Design." In *Software Engineer's Reference Book* (Chapter 17), CRC Press, Boca Raton, Florida, 1993, pp. 17/1-17/22.

[3] Dean, T.R., Cordy, J.R. "A Syntactic Theory of Software Architecture," *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995, pp. 302-313.

[4] Fahmy, H. "Reasoning in the Presence of Uncertainty via Graph-Rewriting," *Technical Report 95-382,* Department of Computing and Information Science, Queen's University, March 1995. (Ph.D. Thesis)

[5] 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).

[6] Gosling, J., Joy, B., and Steele, G. The Java Language Specification, Addison-Wesley, 1997.

[7] Holt, R. "Binary Relational Algebra Applied to Software Architecture," *CSRI Technical Report 345*, Computer Systems Research Institute, University of Toronto, June 1996.

[8] Holt, R., and West, T. Turing Reference Manual, 5th Edition, H.S.A. Inc., 1994.

[9] Holt, R. and Mancoridis, S. "Using Tube Graphs to Model Architectural Designs of Software Systems," *CSRI Technical Report 308*, Computer Systems Research Institute, University of Toronto, October, 1994.

[10] Le Métayer, D. "Software Architecture Styles as graph grammars." In *Proceedings of the 4th ACM SIGSOFT Symposium Foundations on Software Engineering*, (Nov. 1996), pp. 15-23.

[11] Mancoridis, S. "Customizable Notations for Software Design." In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, Madrid, 1997.

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

[13] Prieto-Diaz, R and Neighbors, J.M. "Module Interconnection Languages," *Journal of Systems and Software*, Vol. 6, 1986, pp.307-334.

[14] 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.

[15] Tarski, A. "On the Calculus of Relations," *Journal of Symbolic Logic* , Vol. 6, No. 3, 1941, pp. 73-89

[16] Wills, L.M. "Automated Program Recognition by Graph Parsing," *Technical Report 1358*, MIT Artificial Intelligence Lab., July 1992. (Ph.D. Thesis.)