Layout Merging with Relative Positioning Case Study : Concern Oriented Layout

Hyacinth Ali
McGill University
Montreal, Quebec
hyacinth.ali@mail.mcgill.ca

Gunter Mussbacher
McGill University
Montreal, Quebec
gunter.mussbacher@mcgill.ca

ABSTRACT

The advent of modeling in software engineering, like other engineering fields, has revolutionized the formalism and pace of software development. However, software applications are not built from scratch, instead, other existing software artifacts are reused and combined with new artifacts. This notion of software reuse has been in existence for decades. When structural models such as class diagrams are reused, the reusing and reused models often need to be merged and the result visualized to the modeler. However, state-of-the-art layout mechanisms such as GraphViz do not retain the original layout and rather arbitrarily layout the merged models. Therefore, important information conveyed by the specific layout is currently lost. This paper aims to establish a robust layout algorithm called rpGraph that retains the general layout of the reusing and reused models after merging. rpGraph uses the relative positioning of model elements to inform the positioning of merged model elements. Our findings are evaluated with 20 example model reuses from a library of reusable software model artifacts. A comparison of the merged layouts of rpGraph, GraphViz, and JGraphX shows that rpGraph performs better in terms of retaining the original layouts.

CCS CONCEPTS

• **Software and its engineering → System modeling languages**; **Reusability**;

KEYWORDS

Relative Positioning, Automatic Layout, Software Reuse, Model Composition, Class Diagram

ACM Reference Format:
Hyacinth Ali and Gunter Mussbacher. 2018. Layout Merging with Relative Positioning Case Study : Concern Oriented Layout. In *Proceedings of ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018).* ACM, New York, NY, USA, Article 4, 16 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

In software development, the reuse of existing software artifacts has been explored for decades [5, 18, 21], including but not limited to classes, components [13], design patterns [9], frameworks [14], Software Product Lines [22], and Reusable Aspect Models (RAM) [16]. Every programming language has numerous software libraries to facilitate reuse. Software reuse can increase the productivity, software quality and minimize development cost, time-to-market and schedule overruns [17]. In general, any software artifact can be reused. Although, there are numerous benefits of software reuse, the efforts required to reuse a software artifact should not outweigh those required to create the software system from scratch. Therefore, there is an imperative need to streamline the reusability of software artifacts.

Owing to the advent of model driven engineering [4, 25], virtually all software artifacts can be represented as models including reusable artifacts. While great efforts have been made to increase the reusability of software models, little or no work[1] has been done to improve the resulting layout when combining a reusing and reused model. For some modeling languages, the layout of the resulting model composed of the reusing and reused model does not pose a challenge. In behavioral models, the reusing model often simply references the reused model. For other models such as Feature Models [15], the layout is straightforward even for composed models. However, other modeling languages require the merging of model elements in the reusing and reused models as is the case for many structural models and particularly aspect-oriented techniques (e.g., Kompose [28], Theme/UML [6], and RAM [16]) and more generally model transformations with the intent [19] to merge structural models (e.g., MATA [30]).

State-of-the-art layout mechanisms used to layout models, e.g., GraphViz [8] and JGraphX [26], do not retain the original layout information and rather arbitrarily layout the merged models. However, the specific layout of a model often conveys important information [20], which is currently lost with the existing mechanisms. This paper addresses this problem by establishing a robust layout algorithm called rpGraph which aims to retain the layout information of reusing and reused models when they are combined. As a representative, we use RAM to demonstrate rpGraph in this work, but our findings are also applicable to other merge-based techniques. In RAM, class diagrams are used to describe artifacts which are combined during reuse. The aim is to retain the general topology of the individual class diagrams, i.e., the reusing and reused models, to maintain the *modeler's mental map* [20].

As a prerequisite for rpGraph, we assume that the reused and reusing models have a proper, desired layout formulated by the

MODELS 2018, October 2018, Copenhagen, Denmark

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

[1] A search in the IEEE Xplore and ACM Digital Libraries for publications at the MODELS conference and the search term "layout" resulted in only three relevant publications which are discussed further in Section 3.3. However, none of them addresses the layout of reused models.

modeler. The main contributions of this work are twofold. First, we present a layout metamodel which can be used to graphically represent models based on their relative positioning with the help of automatic layout mechanisms such as GraphViz. This can be applied not only in model reuses but also in model adjustment [12, 20, 24]. Second, we present a layout algorithm that largely retains the layout of models when merged during reuse.

Our findings are evaluated with 20 example model reuses from a library of reusable software model artifacts. In each case, two models with proper layout are combined and the layout of the combined model is compared against the original models. In this work, rpGraph, GraphViz (Graph Visualization Software), and JGraphX (the approach currently used by the RAM tool TouchCORE) layout visualizations are used to layout the merged models and the topology of each layout is compared against the others. Overall, rpGraph performs better in retaining the structure of the original models. The adjacent positions of nodes in each layout diagram are maintained after the diagrams are merged together while other layout mechanisms disorient the position of nodes after the merging.

In the remainder of this paper, we first present the motivation and background of the study in Section 2. Section 3 discusses related work while Section 4 presents the metamodel and the algorithm of rpGraph. In Section 5, we present and discuss the resulting layout of rpGraph and compare it against GraphViz and JGraphX. The conclusion in Section 7 summarizes the contributions of the study and presents future work.

2 BACKGROUND

This section presents a motivating example as well as background information on how models are reused with RAM and the TouchCORE tool.

2.1 Model Reuses with Concerns

Software reuse is defined as the process of creating new software systems using existing software artifacts rather than creating them from scratch [18]. A concern is a generic unit of reuse which groups related models (e.g., class diagrams, sequence diagrams, state machines) that cut across software application [3]. These related artifacts are grouped using the concept of Software Product Line [23]. Each artifact can be described using any software language such as class diagram, sequence diagram, use case diagram, or activity diagram. Examples of concerns are authentication, authorization, and logging. In this paper, we focus on concerns and in particular class diagrams specified with Reusable Aspect Model (RAM) [3] to describe the structural dimension of a concern. Each concern provides three interfaces to facilitate reuse: the variation, customization, and usage interfaces.

HA replaced: The variation interface describes provided features of the concern and their impact on system qualities. The customization interface allows adapting a chosen variation of the concern to a specific reuse context, while the usage interface defines how a customized concern may eventually be used [2]. with

2.1.1 Variation Interface: The *variation interface* specify the variabilities and commonalities in a family of software artifacts that a concern provides and the impact of each selection on system qualities. This helps a software designer to tailor a given concern to her specific needs via selection of the artifacts.

2.1.2 Customization Interface: The *customization interface* allows adapting a chosen variation of the concern to a specific reuse context. For example, a concept, "Authenticatable" in an authentication concern is instantiated as a User in a bank application.

2.1.3 Usage Interface: The *usage interface* defines how a customized concern may eventually be used [2]. It specifies the design structure and behaviour that the concern provides to the reusing application. In class diagram, the usage interface is the set of all public class properties, i.e., the attributes and the operations that are visible and accessible from the rest of the application.

Only the customization interface is relevant for rpGraph, because the merging of two models heavily depends on it. A concern modeler defines partial elements in the reusable model which are used to adapt the concern to a specific application or another concern. The full definition of a partial element depends on the reusing artifact where the partial element is completed. For example, an authentication concern has a partial definition of something that needs to be authenticated (|Authenticatable) and something that needs to be protected (|ProtectedClass). These two elements represent anything in the context of where the authentication is reused. Considering the model of authentication shown in Figure 1, the model elements |Authenticatable and |ProtectedClass are partially defined which is denoted by a bar (|) preceding the name. Assuming the bank application shown in Figure 2 reuses the authentication concern, a mapping of a partial element of authentication (|Authenticatable or |ProtectedClass) to an element in the bank application (User or AccountManager) defines how to adapt the concern to the context of the bank application. Correspondingly, the User and AccountManager classes define the full meaning of |Authenticatable and |ProtectedClass, respectively, in the context of the bank application. Each pair of mapped classes is merged into one class when the models are combined. The HA deleted default layout of this merged model, i.e., without any automatic layout algorithm, is shown in Figure 3. The lower mapped classes |Authenticatable and |ProtectedClass assume the position of the higher mapped classes User and AccountManager, respectively. As can be observed from the figure, several overlapping nodes and edges exist which clearly renders the model difficult to understand.

The HA deleted standard layout generated by GraphViz using dot layout tool as shown in Figure 4 is certainly better than the default layout in Figure 3, but it does not resemble at all the original models. While there are no overlapping elements, the compact layout of the original models is lost, significant white space exists, and model elements belonging to one original model are scattered across the merged models and interspersed with model elements of the second model.

These mappings, the merging process (which is called weaving in RAM), and the subsequent layout of the merged models is essential for the reuse of a concern. An automatic layout mechanism that retains the original layout information while avoiding overlaps of model elements after merging avoids time-intensive manual rearrangement of merged model elements and hence contributes to a more streamlined reuse process, which is the main goal of this work.

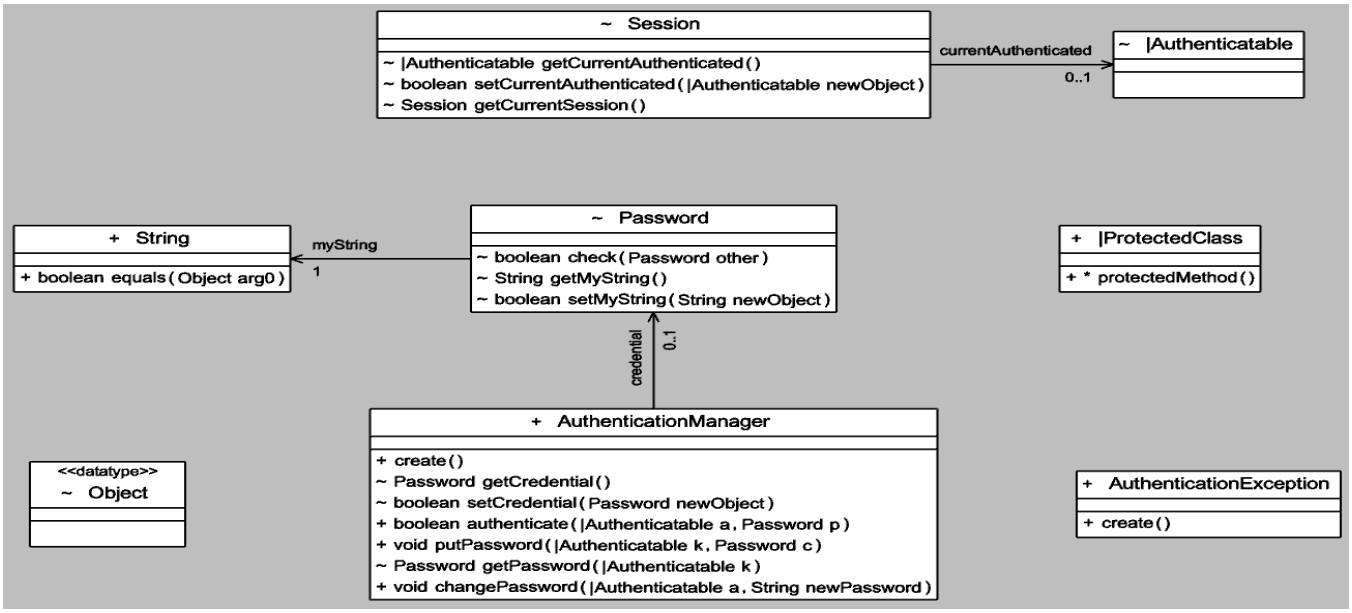


Figure 1: Authentication concern

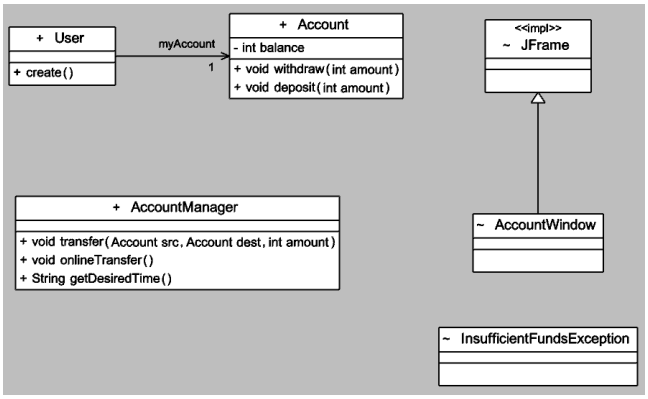


Figure 2: Bank application

2.2 TouchCORE

TouchCORE is a multitouch-enabled software tool used for software design modeling [29]. Its main objectives include but are not limited to developing flexible and reusable software design models [1]. It supports the main three interfaces of concerns: variation, customization, and usage interfaces. TouchCORE also provides the engine for combining different software models during reuses, *weaving* [1]. Tracing functionality in TouchCORE differentiates model elements after they have been combined as shown in Figure 3 and Figure 4. The pink color highlights all elements from the reused concern while the white color highlights the reusing concern.

The current layout mechanism in TouchCORE is based on JGraphX which we discuss in more detail in Section 3.2. TouchCORE is used for the implementation and evaluation of this work.

3 RELATED WORK

3.1 GraphViz

Graphviz is an open source graph visualization software. It draws graphs using graph description languages such as dot, neato, fdp and twopi [8]. GraphViz is the de-facto standard for layouting.

This work uses GraphViz as the back-end during layout and dot as the graph description language. Dot is a textual language used to draw directed graphs. It has essential features for placing nodes and splines of a graph to minimize or avert overlapping edges and nodes. Dot undergoes fours main steps for drawing graphs [10].

(1) **Ensure acyclic graph.** A cyclic graph exists when there is a complete loop of edges from one node back to the same node. The first step breaks any cyclic graph, if any, by reversing the direction of some edges of the graph.
(2) **Assign nodes to discrete ranks.** This entails positioning of nodes at a certain level either vertically or horizontally but not both at the same time. This feature constrains a node within a rank irrespective of its links with other nodes. In top-to-bottom drawing, rank determines the y-coordinate of a node while in left-to-right drawing, rank determines the x-coordinates of a node.
(3) **Position nodes within ranks.** The third step is to rearrange the positions of the nodes within a rank to avoid crossing of edges. In top-to-down drawing, the x-coordinates of the nodes are modified as the y-coordinates are kept constant. Similarly, the y-coordinates are modified in left-to-right drawing during the positioning of nodes within a rank. The coordinates set in this stage, either x or y, aim to keep the edges short.
(4) **Route edges.** The final step is to route the splines connecting different nodes in the graph.

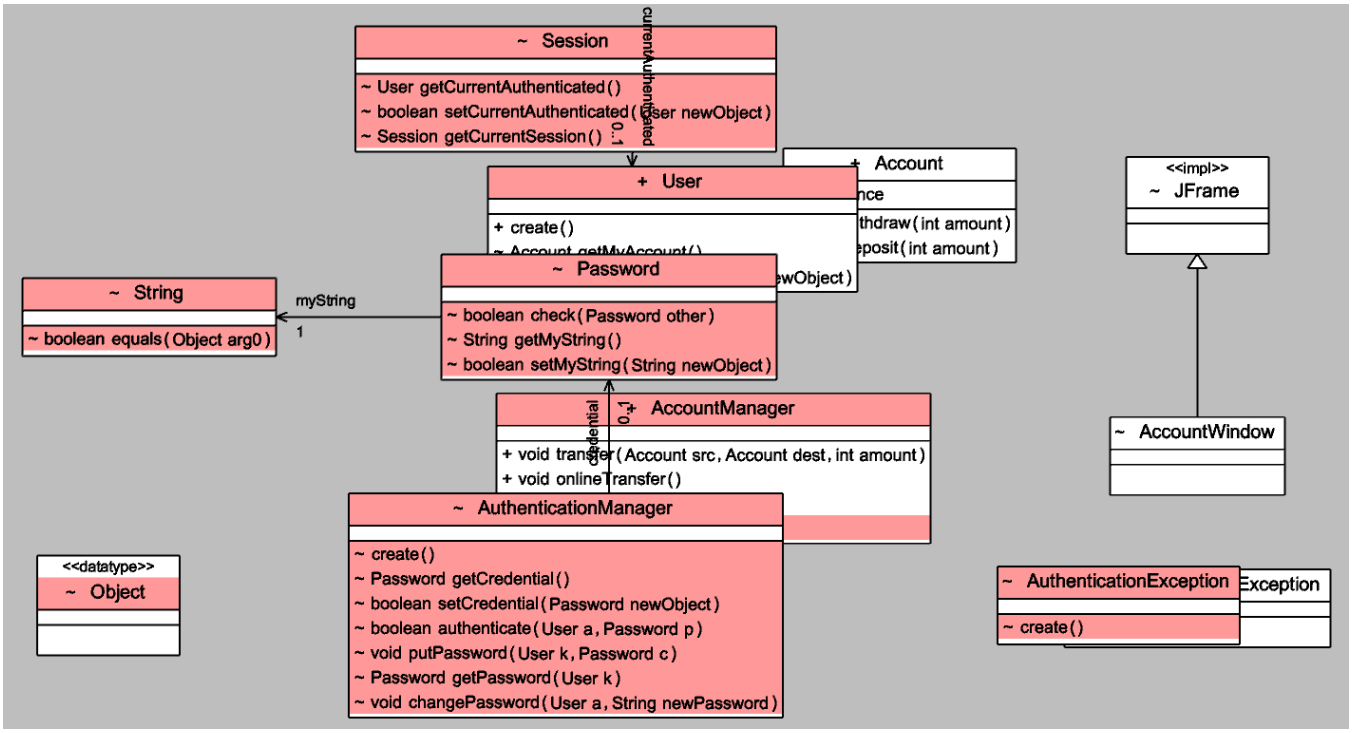


Figure 3: HA deleted Default layout of Merged model of bank and authentication

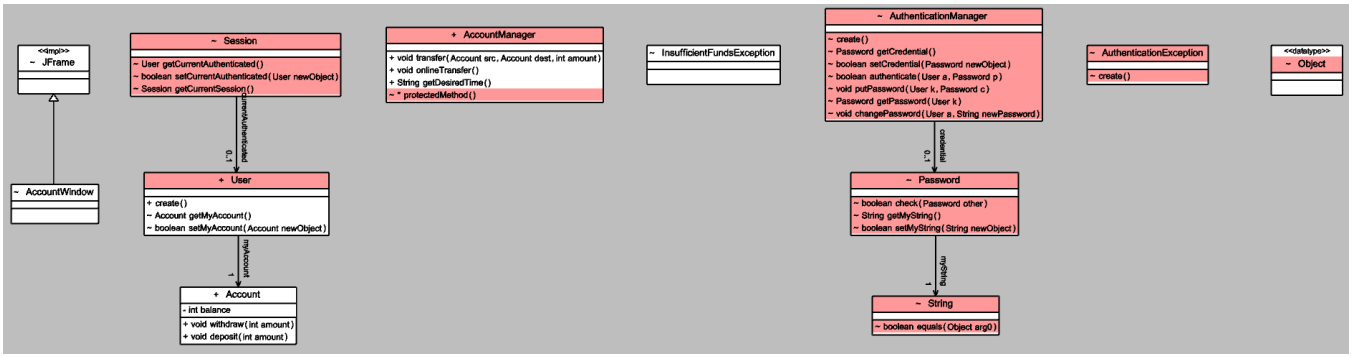
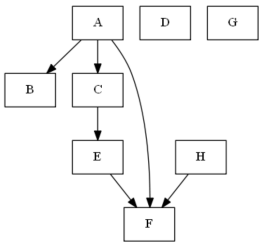


Figure 4: GraphViz layout of merged model of bank and authentication

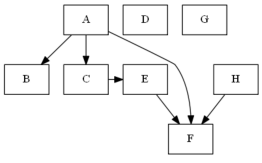
```
1. digraph A {
2.         node[shape=box];
3.         A -> B;
4.         A -> C;
5.         D;
6.         C -> E;
7.         E -> F;
8.         G;
9          H -> F;
10.        A -> F;
11. }
```



(a) Simple graph **(b) Layout of simple graph**

Figure 5: Simple dot specification with layout

```
1. digraph A {
2.         node[shape=box];
3.         A -> B;
4.         A -> C;
5.         D;
6.         C -> E;
7.         E -> F;
8.         G;
9          H -> F;
10.        A -> F;
11.        {rank = same; B, C, E, H}
12. }
```



(a) Simple graph with ranking **(b) Layout of simple graph with ranking**

Figure 6: Dot specification and layout with ranking

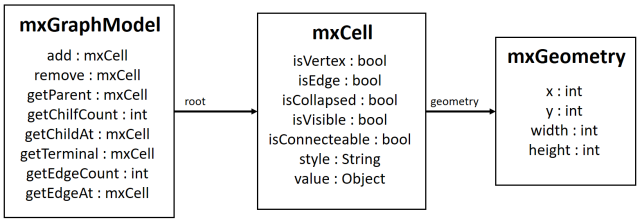


Figure 7: JGraphX design model [27]

```
padding = 7.0;
model.beginUpdate();
try {
    mxGraphLayout newLayout = new mxHierarchicalLayout(graph);
    ((mxHierarchicalLayout) newLayout).setIntraCellSpacing(padding);
    ((mxHierarchicalLayout) newLayout).setInterRankCellSpacing(padding);
    ((mxHierarchicalLayout) newLayout).setInterHierarchySpacing(padding);
    setEdgesOrthogonal(newLayout, graph);
    newLayout.execute(parent);
} finally {
        //update the display
    model.endUpdate();
}
```

Figure 8: JGraphX sample specification

Figure 5 illustrates a graph layout using GraphViz and the dot syntax. Line 1 defines name and type of the graph while line 2 sets the shape of the node to box (see Figure 5a). The following lines, 3 to 10, create nodes and edges and link them accordingly (e.g., A -> B). The visual representation of this graph is shown in Figure 5b. An example of a ranking constraint in a dot specification is depicted in Figure 6. Line 11 specifies a rank for nodes B, C, E, and H. This constraint forces the nodes to be at the same vertical or horizontal level. The default dot specification format is top-to-down drawing and thus the nodes have been constrained within a particular value of the y-coordinate. Close scrutiny of nodes C and E in Figure 6 reveals that C is drawn first before E, from left to right. This is because, the source of the connecting edge is C while the target is E and dot tries to position source node left of target node in the same rank.

3.2 mxGraph

mxGraph is a graph library which enables users to quickly create graphs used for analysis, visualization, interaction, and layout [26]. JGraphX is one of the graph library technologies embodied in mxGraph. mxGraph can be run in a web browser with a backend server or locally within a machine. The java library JGraphX of mxGraph is suitable for desktop applications while the Javascript implementation of mxGraph is preferred for web application [26]. This section focuses on JGraphX, because it is the one currently used by TouchCORE.

The layout algorithm of JGraphX assists users in a meaningful display of graph elements while avoiding overlap among the graph elements. The design model of JGraphX is shown in Figure 7. The design model describes the structure and behaviour of JGraphX elements. mxGraphModel contains several operations used for adding, removing, and manipulating diagram elements. mxCell represents any element of the diagram such as an egde or node. mxGeometry specifies the layout information of each cell which includes the x, y, width, and height values.

An example JGraphX snippet using hierarchical layout [11] is shown in Figure 8. After creating a new layout and setting the padding, the edges of the graph are set and the layout mechanism is executed. Finally, the model is updated to reflect the layout information. JGraphX supports various tree, force-directed, and hierarchical layouts [26].

3.3 Other Layout Mechanisms

This section discusses related layout mechanisms as identified by a search of the publications at the MODELS conference series (see footnote 1).

Balazs *et al.* [12] present a *textual diagram layout language* that allows modelers to describe the arrangement of model elements. The storing, editing, and versioning of models are better implemented in a textual language while graphical representation remains a preferred option for better comprehension. Balazs *et al.* do recognize that automatic layout mechanisms often invalidate some intents of the modeler by arbitrarily rearranging model elements. To this end, the *textual diagram layout language* is used to provide a concise layout description. Its basic idea is to define the position of nodes relative to others, e.g., *Above(Box1, Box2)* means that Box1 is above Box2. While the basic idea of relative positioning is similar to rpGraph, the *textual diagram layout language* does not address the layout of merged models during reuse. Furthermore, rpGraph infers the relative positioning of model elements from the existing layout while the *textual diagram layout language* requires modelers to explicitly specify the layout with the language.

Yosser and Cedric [7] argue that UML's graphical layout needs to be improved in order to enhance the communication among stakeholders. The color, orientation, and position of model elements, among others such as size, brightness, and grain/texture, are used to enhance UML models as a vehicle of communication. Although, this work does not address relative positioning and layout of merged models during reuse, its aim, similar to rpGraph, is to retain the mental map of the stakeholders about the models.

Evolution of software is spread across all software artifacts. Therefore, when a graphical modeling language is updated, migrating existing models may lead to improper layout such as overlap of nodes. Ult *et al.* [24] present a layout algorithm based on network simplex which rearranges the model elements to depict a proper layout while keeping the general framework of the model. In the context of this work, a proper layout retains the original positioning of model elements after model diagrams are merged or updated. This model adjustment keeps models synchronized with changes to software tools and language. However, the layout of a merged model during reuse is not addressed.

4 RPGRAPH LAYOUT ALGORITHM

As a possible solution to the challenges of determining a proper layout of merged model elements, this work proposes an algorithm to retain the original layout information while avoiding edges and

elements overlap. The algorithm operates on the proposed rpGraph metamodel which establishes relative relationships among model elements. This section details the metamodel first and then explains how we intend to use it to solve the problem of proper layout of merged models after reuse.

4.1 Layout Metamodel of rpGraph

This section describes the abstract syntax and semantics of our metamodel shown in Figure 9. The LayoutModel contains a set of Nodes and a set of Edges, with each Edge going from a source Node to a target Node. The position relationships covered by Edge in the metamodel are Above, DirectAbove, Left, and DirectLeft. The Above relationship exists between two Nodes (i.e., rectangles) where the source Node is above the target Node when visualized. The same relationship applies to Left: the source Node should be at the left of the target Node when visualized. Additionally, DirectAbove is when the source Node is HA deleted fairly directly above the target Node, i.e., a vertical line extending the left or right side of the source Node touches the target Node. As shown in Figure 2, Session is directly above Password while Session is Above |ProtectedClass. Similarly, a horizontal line extending the top or bottom sides of the source Node touches the target Node in the case of DirectLeft. These relationships are intended to be used while visualizing the model to make sure that their relative positions are maintained, and overlaps of nodes and edges are minimized. The following points highlight additional features of the relationships.

- **Above:** An Above relationship is established for all Nodes below a Node. If Node A is above Node B which is above Node C, then three relationships are created: A above B, A above C, and B above C. This approach is preferred over an approach that exploits transitivity, because it simplifies processing of the LayoutModel later on.
- **DirectAbove:** In some cases, a target Node that is directly below the source Node behaves differently when visualized HA replaced: because there is no Left relationship between the two Nodes. The target may hence be placed freely left or right of the source Node. with especially when it does not have source Node based on Left or DirectLeft relationship. The target Node may hence be placed freely far left of the above source Node especially when the node is from a model diagram that needs to be positioned right of another model diagram after merging. To solve this problem, we introduced DirectAbove as shown in Figure 9. This relationship ensures that the target node is restrained from moving left of the source above node.
- **Left:** Similar to Above, a Left relationship is established for all Nodes right of a Node (if Node A is left of Node B which is left of Node C, then three relationships are created: A left of B, A left of C, and B left of C).
- **DirectLeft:** Similar to DirectAbove, we use DirectLeft to restrain Nodes from being positioned freely above or below the source Node even though it should be situated right next to the source Node.

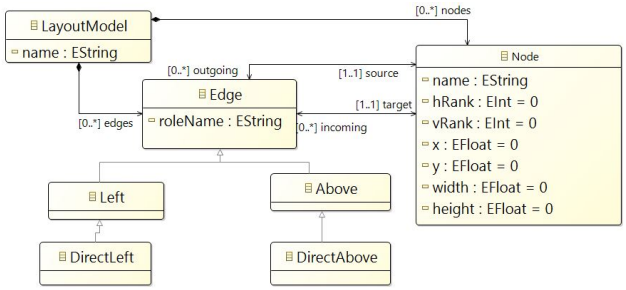


Figure 9: Layout metamodel

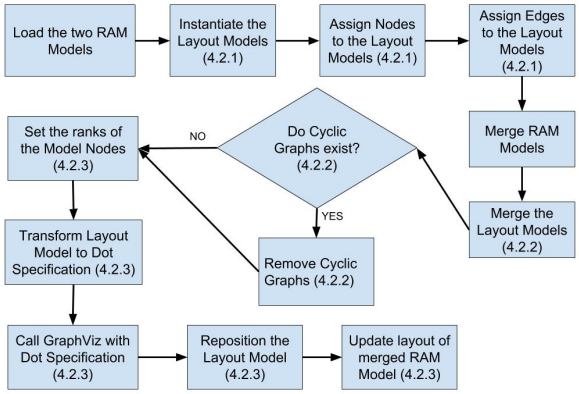


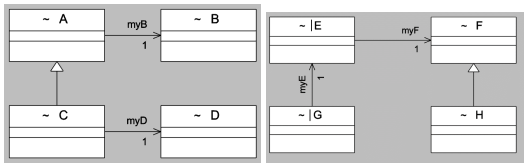
Figure 10: rpGraph algorithm

4.2 The rpGraph Layout Algorithm

This section describes the rpGraph layout algorithm of a merged model. This algorithm is based on the metamodel (see Section 4.1), employs GraphViz (see Section 3.1) as the backend, and is implemented in the TouchCORE tool (see Section 2.2). The following subsections detail the algorithm while an overview of the algorithm is shown in Figure 10.

4.2.1 Create layout models. The RAM models of the reused and reusing models are maintained by the TouchCORE tool. The x-coordinate, y-coordinate, width, and height values of each model element are used as parameters to instantiate the Node class in the metamodel (see Figure 9). Prior to creating instances of Node, two LayoutModels are instantiated, one corresponding to the reusing (higher) model and one to the reused (lower) model. Following these instantiations, each Node instance is added to its corresponding LayoutModel instance.

Afterwards, we create and assign Edge instances to nodes based on their attributes. At this juncture, we evaluate each node using its x-coordinate, y-coordinate, width, and height to establish its position with respect to all other nodes in the model. The evaluation of relative positions of each element is based on its original model. Exactly one DirectAbove relationship, or exactly one DirectLeft relationship, or exactly one pair of Above and Left relationships is created for each pair of two nodes in a model.



(a) Reusing model **(b) Reused model**

Figure 11: Cyclic graphs

4.2.2 Merge layout models. Having transformed the two models to layout models, we merge the two using the weaving information [3] provided by TouchCORE, i.e., after TouchCORE has merged the two RAM models (see Figure 10). During the layout merging, each mapped element from the lower model, i.e., the reused model, is removed while its edges are copied into the corresponding higher mapped element, i.e., the reusing model elements. The result of this merge can lead to a cyclic graph because of the assigned edges of the nodes.

HA replaced: For example, a lower element E may be mapped to a higher element D and another lower element G may be mapped to higher element B and E is above G while B is above D as shown in Figure 11. with For example, a lower element E may be mapped to a higher element D and another lower element G may be mapped to higher element B. In reused model, E is above G while B is above D in the reusing model as shown in Figure 11. During the edge assignment, an Above edge is assigned to E and G, where E is the source and G is the target. Similarly, an Above edge is assigned to B and D, where B is the source and D is the target. However, due to the merging of the two layout models, class E and D have been combined into one class D, and class G and B have equally been combined into one class B. This implies that B has two above relationships where D is the source element in one and target element in the other. This leads to a cyclic graph. In order to avoid infinitum runtime during ranking of model elements which uses recursion algorithm HA deleted apply GraphViz (which does not support cyclic graphs) as the backend of the rpGraph layout algorithm, we remove the relationship HA replaced: belonging to the lower model (the reused model), with established between E and G because we give higher priority to the higher model (the reusing model) as the lower model is merged into it. The next section explains how we use GraphViz to reposition the layout model elements based on relative position information.

4.2.3 Reposition the layout with GraphViz. Having created the merged layout model with its nodes and edges (Above, DirectAbove, Left, and DirectLeft), we aim to use the attributes of the layout model to reposition its nodes accordingly.

In the first place, prior to the transformation of the model to dot specification, we assign ranks to each model element as shown in the attributes of the Node class in the metamodel (see vRank and hRank in Figure 9). We assign two ranks per element, vertical and horizontal ranks. Vertical ranking is top-down while horizontal ranking is left-right starting from number one. We use the edges of a node to determine its rank recursively. The pseudo-code of the ranking algorithm is shown in Figure 12. The diagram show how vertical ranks are determined which is similar to horizontal ranks

```
1.   Operation determineVRank(Node node) {
2.   //The rank is 1 when there is no incoming edge
3.   If node.incomingEdges = empty {
4.      return = 1
5.   } Else {
6.     //Create list of nodes to store above nodes
7.   DECLARE nodes : LIST of Node
8.   For each edge : Edge in node.incomingEdges {
9.      if edge = Above or edge = DirectAbove {
10.             Node n = edge.getSourceNode
11.             nodes.add(n)
12.             }
13.    }
14.    rank = 0;
15.    For each n : N in nodes {
16.             //Recursive call
17.             call : nRank = determineVRank(n)
18.             if (rank < nRank) {
19.                rank = nRank
20.             }
21.    }
22.    rank = rank + 1
23.    return rank
24.  }
25.  }
```

Figure 12: Nodes ranking

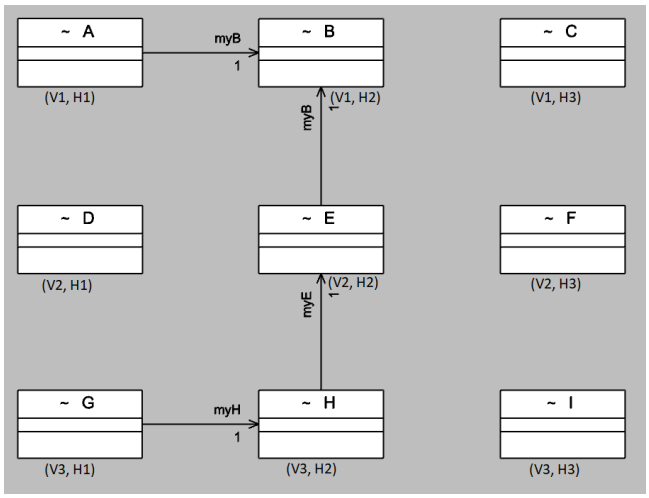


Figure 13: Ranking of nodes (vertical and horizontal ranks are indicated in parentheses next to a node)

except that Left and DireftLeft edges are used. The algorithm first checks if the node has any incoming edge else the rank is 1, lines 3 to 5. For example, the rank of node A in Figure 13 is 1. However, if there are incoming edge, this shows that there are above nodes and these nodes collected and stored, lines 7 to 13. Each stored node is processed to determine its rank using recursive method, lines 14 to 23.

Figure 13 shows an annotated model to illustrate the ranking of nodes, V1 is vertical rank 1 and H1 is horizontal rank 1 and so on.

```
1. maxVRank = 0
2. For each node : Node in nodes {
3.      node.determineVRank()
4.      node.determineHRank()
5.      If maxVRank < node.vRank {
6.              maxVRank = node.vRank
7.      }
8. }
9.  // lines 1 to 5
10. print header
11. // line 6
12. For (i=1; i < maxVRank; i++) {
13.     print i ->
14. }
15. print maxVRank ;
16. // lines 7 to 15
17. For each node : Node in nodes {
18.     print node.name [pos="" width=" node.width ", height=" node.height "];
19. }
20. // lines 16 to 24
21. For each node : Node in nodes {
22.     print {rank = same; node.vRank ; node.name }
23. }
24. // lines 25 to 30
25. For node1 : Node in nodes and node2 : Node in nodes {
26.     If node1.hRank = node2.hRank AND node1.vRank - node2.vRank = -1 {
27.             print node1.name -> node2.name ;
28.     }
29. }
30. // lines 31 to 36
31. For node1 : Node in nodes and node2 : Node in nodes {
32.     If node1.vRank = node2.vRank AND node1.hRank - node2.hRank = -1 {
33.             print node1.name -> node2.name ;
34.     }
35. }
36. // line 37
37. print }
```

Figure 14: Layout model to dot transformation (the comments // include references to the lines in Figure 15)

```
1. digraph G{
2.      newrank = true;
3.      node[shape = box];
4.      nodesep = 1.2;
5.      ranksep = 1.5;
6.      1 -> 2 -> 3;
7.      F[pos="", width="2.08", height="1.138"];
8.      A[pos="", width="3.93", height="1.47"];
9.      B[pos="", width="2.08", height="1.138"];
10.     C[pos="", width="2.08", height="1.138"];
11.     D[pos="", width="2.08", height="1.138"];
12.     E[pos="", width="3.93", height="1.47"];
13.     G[pos="", width="3.96", height="1.47"];
14.     H[pos="", width="3.93", height="1.47"];
15.     I[pos="", width="2.08", height="1.138"];
16.     {rank = same; 1; A}
17.     {rank = same; 1; B}
18.     {rank = same; 1; C}
19.     {rank = same; 2; D}
20.     {rank = same; 2; E}
21.     {rank = same; 2; F}
22.     {rank = same; 3; G}
23.     {rank = same; 3; H}
24.     {rank = same; 3; I}
25.     A -> D;
26.     B -> E;
27.     C -> F;
28.     D -> G;
29.     E -> H;
30.     F -> I;
31.     A -> B;
32.     B -> c;
33.     D -> E;
34.     E -> F;
35.     G -> H;
36.     H -> I;
37. }
```

Figure 15: Dot specification of the sample model

As shown, node A is above node D and node D is above node G. It implies that node A, D, and G have a vertical rank of V1, V2, and V3, respectively. Similarly, node A is left of node B and node B is left of node C. This also implies that node A, B, and C have horizontal rank of H1, H2, and H3, respectively. The rpGraph layout algorithm sets the vertical ranks using Above and DirectAbove edges while Left and DirectLeft are used for the horizontal ranking (lines 3-4 in Figure 14). These rank values are used in addition to other attributes of the nodes during the transformation to dot. As explained in Section 3.1, nodes can be ranked vertically or horizontally in GraphViz but not both at the same time. In this work, we apply the vertical ranks to constrain a set of nodes within a distinct level (lines 1, 5-7, 11-15, and 20-23 in Figure 14; lines 6 and 16-24 in Figure 15). On the other hand, both the horizontal and vertical ranks are used during the connection of the nodes with the -> operator (lines 24-29 for vertical ranks and 30-35 for horizontal ranks in Figure 14; lines 25-30 and lines 31-36, respectively, in Figure 15). For example in Figure 13, the same vertical rank V1 ensures that A, B, and C are constrained within the first level. Horizontal ranks are used to position the nodes within a vertical rank. Since node A is left of node B while A and B are on the same vertical rank, the corresponding dot specification is A -> B. This approach is applied to all nodes to obtain the corresponding specification in the dot notation. Figure 15 shows the corresponding dot specification for Figure 13. The resulting dot specification is fed to GraphViz to obtain the new x and y coordinates of each node. The new coordinates are consequently used to update the merged model accordingly.

4.3 GraphViz Layout Algorithm

In this work, the layout models of rpGraph are compared with the layouts of GraphViz using dot language. Hence, we present in this section the step by step processes followed to produce the GraphViz layout models. In the first place, the width, and height values of each RAM model elements are used as parameters to create node and its attributes in the dot notation (lines 1-3 in Figure ...). These nodes are created arbitrarily as TouchCORE does not provide the RAM model elements in any order. TouchCORE tool also provides the existing associations among the model elements, each association is used to create a corresponding link in the dot notation, (lines 4-6 in Figure ...), where the source node represents the source of the association and the target node represents the target association. For generalization, a dot link is created for each inheritance, the source node of the link represents the super class while the target node stand for the sub class. The GraphViz algorithm used in this work top down and thus, representing the super as the source node of the link makes it to be drawn above the sub class. The next step is to feed the dot specification, example shown in Figure ., to GraphViz which produces new x and y coordinates of each node. The new coordinates are consequently used to update the RAM model elements accordingly. The idea behind the GraphViz layout algorithm is produce a graph which has the same structural relationship as the input RAM model.

4.4 Examples

In this section, we illustrate the layout problem with two examples, show how our algorithm stands to address the issue, and visually compare our results with related layout solutions, i.e., the de-facto layout standard GraphViz and JGraphX, which is the layout mechanism currently used in the TouchCORE tool. Figure 16 depicts two models with the desired layout. Model 1 reuses Model 2 with class

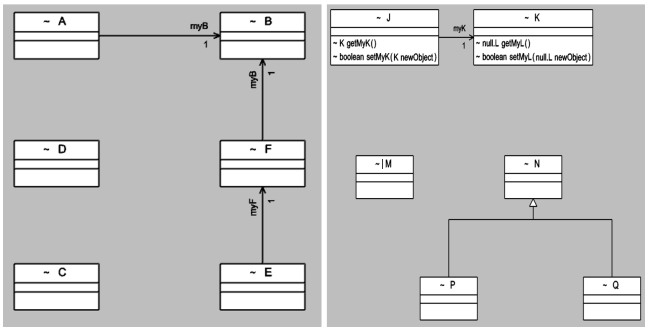


Figure 16: Reuse maps M in model 2 to F in model 1

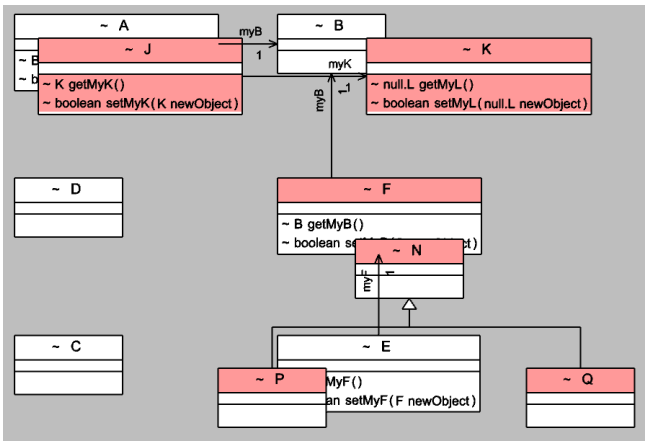


Figure 17: Default merged model

M in Model 2 being mapped to class F in Model 1, i.e., M and F are combined in the merged result.

When these models are merged together without the application of any automatic layout algorithm, the model elements overlap one another as they retain their original positions as shown in Figure 17. The result of the automatic layout approaches GraphViz, JGraphX (TouchCORE), and rpGraph are depicted in Figure 18 with the reused model highlighted in pink color and the reusing model shown with white background. Close inspection of the models in Figure 18 reveals that rpGraph maintains the relative location of model elements of each model while keeping the model dimension ratio fairly constant and avoiding additional free space in the merged model. On the other hand, the GraphViz and JGraphX solutions reorder the model elements arbitrarily which leads to loss of important information conveyed by the original model layout. For example, the original reuse model shows that B is above F while F is above E but this ordering is reversed in the merged model using GraphViz or JGraphX. Another important loss of information is placement of B under A instead of right of A, as shown in the merged model of GraphViz or JGraphX. As another example, consider the authentication and bank application models from Figure 1 and Figure 2, respectively, as well as the default merged layout shown in Figure 3. |Authenticatable and |ProtectedClass in authentication are mapped to User and AccountManager in the bank application, respectively. The results of the automatic layouts of GraphViz, JGraphX (TouchCORE), and rpGraph are shown in Figure 19. HA replaced: and similar observations can be deduced from the models. with Similarly, rpGraph keeps the original relative positions of the model elements while GraphViz and JGraphX reorder the layout to the extent that most of the original relative positions are lost. Examples, the layout mental map of the bank application is completely lost as the model elements are scattered in the merged model of both JGgraphX and GraphViz. Although, the model elements of the authentication are more close in both cases (JGraphX and Graphiz), the relative positions of the model elements are not maintained.

In the next section, the three automatic layout algorithms are compared with each other more formally with the help of a set of metrics characterizing the layout of the reused model, the reusing model, and the merged result.

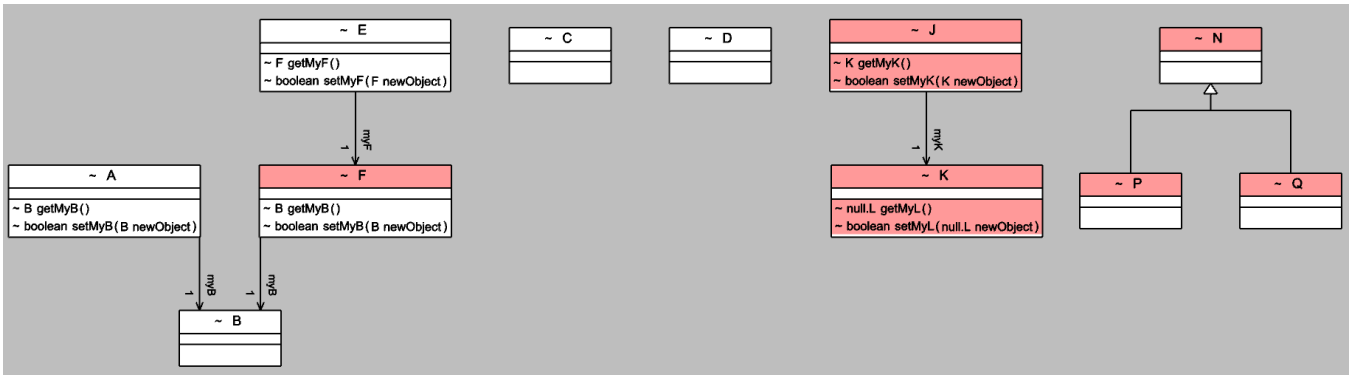
5 RESULTS AND DISCUSSION

In this section, we evaluate our rpGraph algorithm by comparing its layout of merged models with two notable layout algorithms, GraphViz (GV) and JGraphX (GX). In this evaluation, we denote our algorithm as Relative Positioning (RP). We use existing reusable RAM models (class diagrams) including authentication, access control, and workflow from the TouchCORE library of reusable models. In total, we selected 20 reusable artifacts and each is reused in a specific context such as bank, door, or accounting applications (see Table 1). The number of classes in each reusing model ranges from 2 to 17 while the reused models have between 1 to 20 classes each. The number of classes in each merged model spans from 2 to 35. The number of pairs of mapped classes used in the evaluation ranges from 1 to 4 for each model reuse. All the original and merged models used in this evaluation are available in the rpGraph GitHub repository[2].

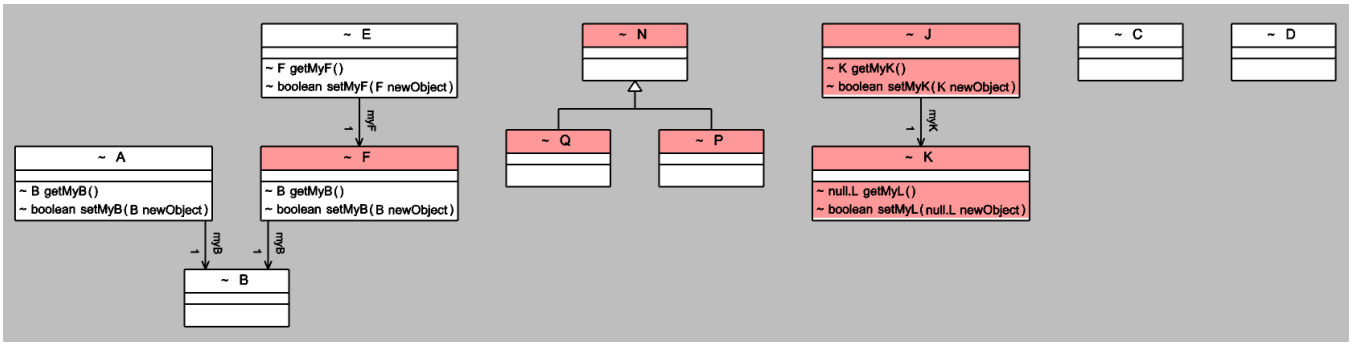
In the remainder of this section, we present the metrics and discuss the results of the layout algorithms. The evaluation based on metrics is rooted in the assumption that the modeler's mental map of an original model is least disrupted, if the layout does not change. This involves, among others, the retention of the original positions of the model elements after combining the models. The metrics hence measure changes to the layout from various angles. Whether the resulting layouts in fact disrupt the modeler's mental map can only be answered with an empirical study involving human subjects, which is out of scope for this work and needs to be done in future work. The focus here is on demonstrating that the rpGraph layout algorithm retains the original model layouts.

The first evaluation is based on the number of overlapping model elements. The second metric assesses the difference of model dimension ratios between the merged and the original models. The next metrics measure the concentration of the original model elements in the merged model as well as the free spaces in the merged model

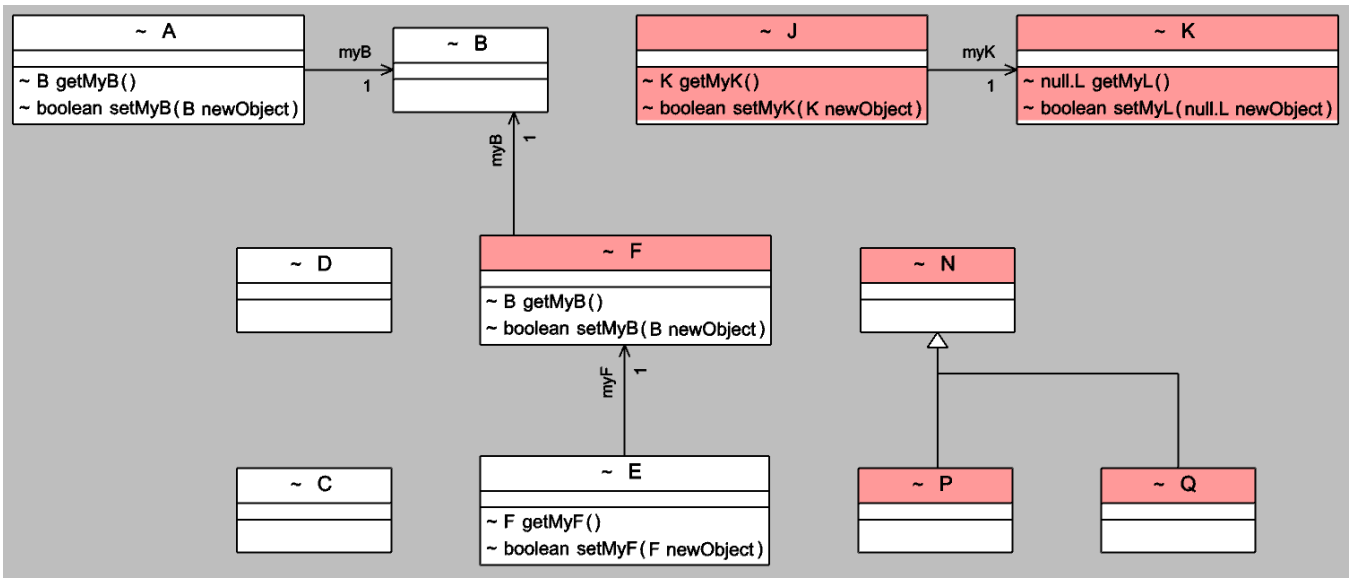
[2]The individual and merged models, merged model of rpGraph, GraphViz, and JGraphX can be found in GitHub, https://github.com/Hyacinth-Ali/AutoLayoutDataset/tree/master/LayoutImages. Pink color elements are reused elements while white elements are reusing elements. The name of the models rx, gv, and gx correspond to the layout of rpGraph, GraphViz, and JGraphX, respectively.



(a) GraphViz



(b) JGraphX

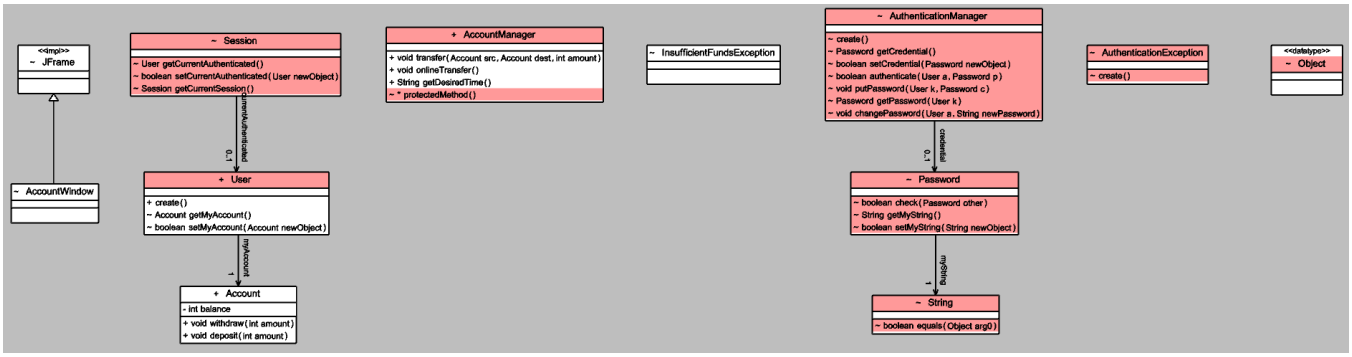


(c) Relative positioning (rpGrah)

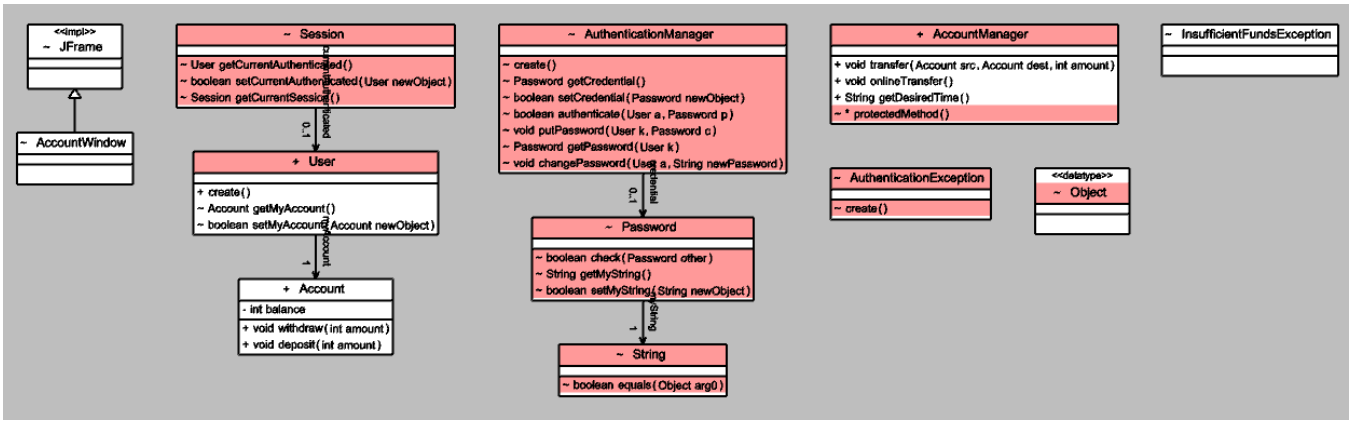
Figure 18: Comparison of layout solutions of merged model 1 and model 2

compared to the original models. The final evaluation looks at the number of pairs of model elements that still have the same relative positioning (above/below and right/left) in the merged model compared to the original models.

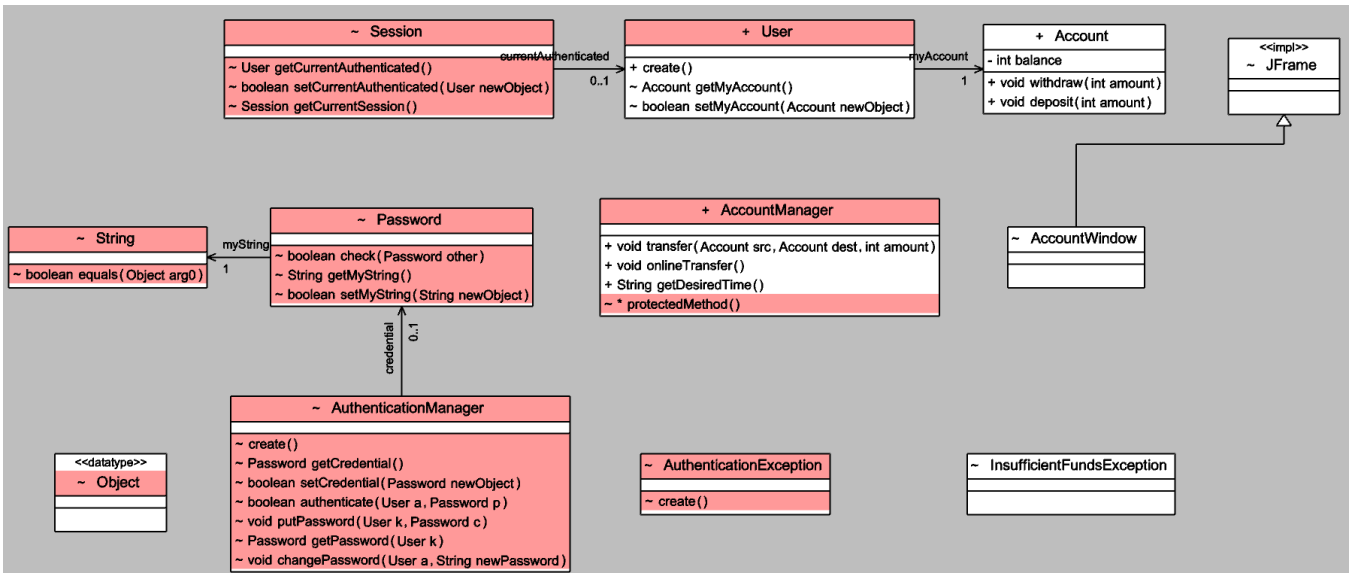
Overlapping model elements. This metric counts the overlaps among edges, and between nodes and edges of each merged model. Hence, 0 is the optimal result for this metric. Figure 20 shows the box plot of the number of overlaps which summarizes its distribution. EE (Edge/Edge) denotes the number of overlaps among edges while



(a) GraphViz



(b) JGraphX



(c) Relative positioning (rpGraph)

Figure 19: Comparison of layout solutions of merged bank and authentication

Reusing Model	Reused Model
Bank	Authentication
Demo	Minueto
Door	Access Control
Workflow	Copyable
Input Output	Singleton
Input	Name
Input Output	Command
Conditional	Named
Command	Named
Inpath	Named
Checkpoint	Copyable
Traceable	Access
Accounting	Authentication
Accounting	Mortgage
Accounting	Minueto
Accounting	Workflow
Mortgage	Minueto
Accounting	Command
Mortgage	Command

Table 1: Table of reusing and reused models

NE (Node/Edge) denotes the number of overlaps among nodes and edges. While the median number of overlaps among all the layout mechanisms is 0, the third quartile of NE and EE using GV is 1 and the third quartile of NE and EE using GX is 2, both having outliers from 4 to 7. In contrast, RP has a smaller number of overlaps for both NE and EE as the values are predominantly between 0 and 0.3, while in GV and GX, the values are spread wide, from 0 to 5. The outliers of RP are also smaller than the outliers of GV and GX, hence indicating better performance of rpGraph.

Difference of model dimension ratios. The second metric we use in this evaluation is the model dimension ratio of a model, that is, the ratio of the width to the height of a model. In this evaluation, we aim to determine how the model dimension ratio of each original model is retained after merging. Hence, the difference between the "before" and "after" ratio of each algorithm is calculated. This difference is 0 for an optimal layout of the merged model that is most similar to the original layouts.

Figure 21 shows the distribution of the difference in model dimension ratios. The small gap between first and third quartiles of RP compared to GV and GX indicates that RP better maintains the model dimension ratio of each model. In detail, the median of the difference using RP is about 0 while the median of the other ranges from 1 to 3. The extreme values for RP are 2 and -4.3 compared to 8 and -8 for GV and 8 and -6 for GX. Overall, rpGraph outperforms the other layout mechanisms in retaining the dimensions of the layouts of the original models.

Element concentration. Furthermore, we evaluate the concentration of the individual model elements from one original model in one place in the merged model. The objective of this is to determine how model elements of one original model are still positioned close to one another after merging. In this regard, we aim to calculate

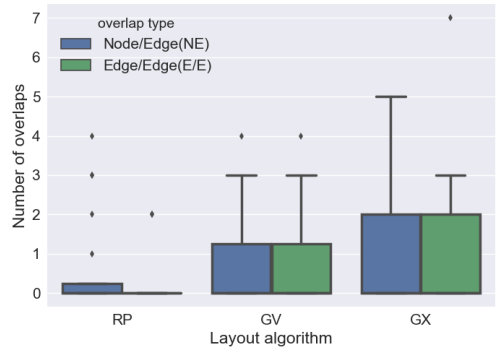


Figure 20: Overlapping of model elements

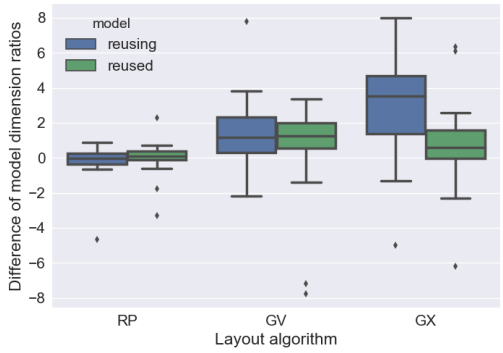


Figure 21: Difference of model dimension ratios

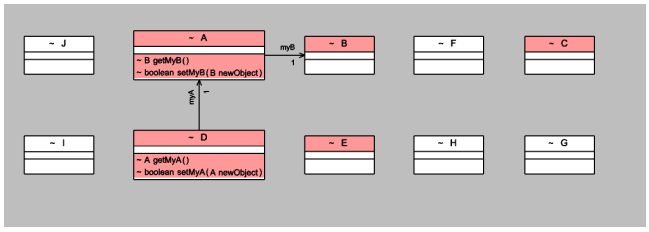


Figure 22: Element concentrations in merged model

the highest number of adjacent nodes, i.e., vertically or horizontally adjacent nodes in each merged model. Equation 1 shows the formula we employ for this metric.

$$EC = \frac{Highest\ No.\ of\ Adjacent\ Nodes}{Total\ No.\ of\ Nodes} \quad (1)$$

For illustration (see Figure 22), the pink colored elements (A, B, C, D, and E) are from the reused model (m2) while the white nodes (E, F, G, H, I, and J) are from the reusing model (m1). Note that node E is a mapped node, therefore, it belongs to both m1 and m2. The optimal value of this metric is 1. Thus, by applying Equation 1, the HA replaced: adjacent nodes of m1 in the merged model are E, F, H,

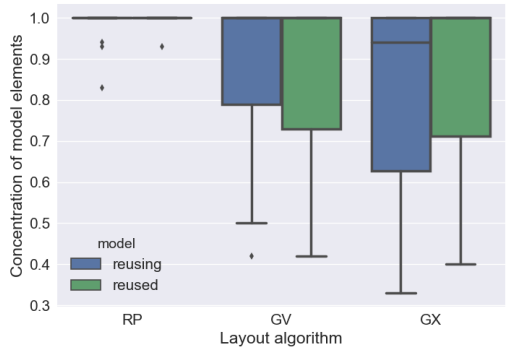


Figure 23: Concentration of model elements

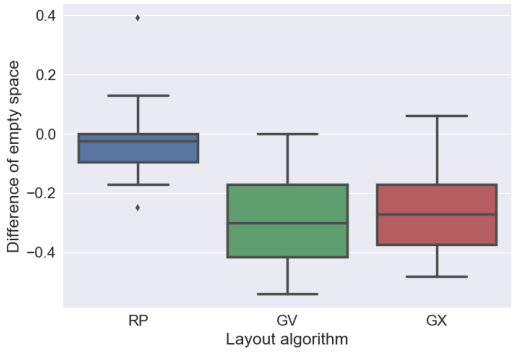


Figure 24: Difference of empty space of models

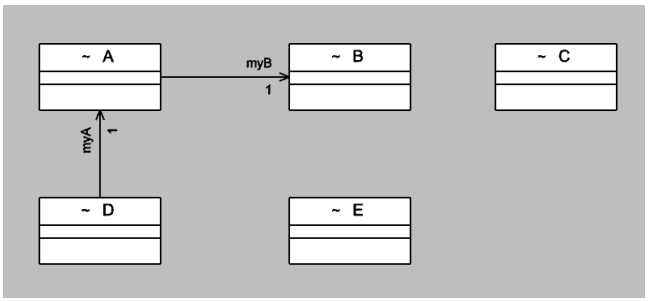


Figure 25: Empty space in model

and G but not I and J, (i.e., the result is $\frac{4}{6} = 0.67$) while the adjacent nodes of m2 in the merged model are A, B, D, and E but not C (i.e., the result is $\frac{4}{5} = 0.8$). with highest number of adjacent nodes of m1 is 4, i.e., (E, F, H, and G) but not 2 (I, and J). Therefore, the result is $\frac{4}{6} = 0.67$) while the highest number of adjacent nodes of m2 in the merged model is 4, i.e., (A, B, D, and E) but not 1 (C). Hence, the result is $\frac{4}{5} = 0.8$).

Figure 23 shows the result of this evaluation using the 20 model reuses. The figure shows how the concentration of elements in RP is typically around 1 unlike GV and GX where the concentration is spread over a much wider range. Although, there are outliers in RP with a lowest value at 0.83, this value is still higher than the first quartile of GV and GX. The lowest value in RP is 0.83 while it is just above 0.4 in GV and just above 0.3 in GX, showing that rpGraph keeps the model elements of a model largely close to each other even after they have been combined with other models.

Difference of empty spaces. The objective of the empty space metric of a model is to determine how much white space is introduced in the merged model (e.g., an original model placed at the top left while the other original model is at the bottom right with the top right and bottom left quadrants fairly empty). Equation 2 shows the formula we use to calculate the empty space.

$$ES = \frac{No.\ of\ Nodes}{Absolute\ No.\ of\ Nodes} \tag{2}$$

This metric is illustrated in Figure 25, where the number of nodes is 5 while the absolute number of nodes is 6 (the empty space below C and to the right of E is counted as another node), resulting in an empty space metric of $\frac{5}{6} = 0.83$. The absolute number is hence the number of nodes that would make the model look like a square or rectangle with all parts filled in with nodes. A node may be counted as two (three, four, ...) if its width or height spans across two (three, four, ...) other nodes. This implies that the maximum value of the metric is 1 with higher values signifying more compact layout.

In this evaluation, we compare the empty space of a merged model with the empty space of the original models. The average of the original models is used, i.e., the sum of the empty space of reusing and reused models divided by 2. Therefore, 0 means that there is no difference in empty spaces, a positive/negative value indicates that empty space has been reduced/increased in the merged model, respectively. The result of this evaluation is shown in Figure 24. The smaller box size of RP whose third quartile is still at 0 indicates the small degree of difference between the compactness of the RP layout and the original layouts compared to GV and GX, whose boxes are wider with their third quartiles close to -0.2. The largest value of RP is 0.15 with an outlier of 0.4, while the maximum value of GV and GX is approximately at 0. The lowest outlier of RP is still higher than the median of GV and GX, with the lowest values of GV and GX going well below -0.4. This evaluation further indicates the robustness of rpGraph in retaining the original layout of models after they have been merged.

Similarity of relative positioning. The last metric verifies the basic premise of the rpGraph algorithm, i.e., the relative positioning of a pair of model elements should not change from the original models to the merged models. Equation 3 shows the formula we use to calculate the similarity of relative positioning.

$$SRP = \frac{No.\ of\ Pairs\ with\ Similar\ Relative\ Positioning}{Total\ No.\ of\ Pairs\ of\ Nodes} \tag{3}$$

Therefore, each pair of nodes in the merged model is compared against the corresponding nodes of the original models. If at least one of the above/below or right/left designations changes from the original model to the merged model, then this pair is not counted for the dividend in Equation 3. As can be seen from Figure 26, RP largely retains the relative positioning whereas GV and GX significantly change the relative positioning. Note that the outliers

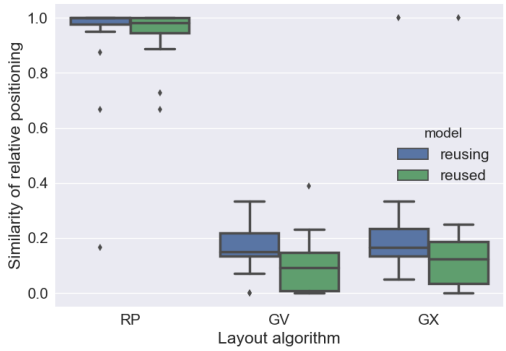


Figure 26: Similarity of relative positioning

for RP can be explained by anomalies due to a higher number of mappings between model elements of the reusing and reused model. Nevertheless, rpGraph clearly outperforms GraphViz and JGraphX in terms of retaining the original layout.

In summary, additionally to indicating that our rpGraph algorithm largely retains the relative positions of the original model elements, these evaluations also demonstrate that it outperforms GraphViz and JGraphX with regards to overlapping model elements, the retention of model dimension ratios, the continued concentration of individual model elements from the original model, and the occurrence of empty space in the merged models.

6 CONCERN ORIENTED REUSE LAYOUT USING RPGRAPH

Concern-Oriented Reuse (CORE) is a software reuse paradigm that promotes the use of concern its main artifact during software development. A concern groups related models (e.g., class diagrams, sequence diagrams, state machines) that cut across software application, Section 2.1. During software reuse, different concern models are *woven/composed* to generate a specific application required for a given a context. In this section, we detail how we use rpGraph to layout the combined models of a concern. In the first place, we give an structural overview of a concern and then present the algorithm of concern models using rpGraph.

6.1 Concern Structure

The basic structure of a concern is shown in the excerpt of the metamodel in Figure 27. A concern (COREConcern) groups related models (COREModel) together, with at least one model by default. A COREModel is an abstract class whose implementation is defined by classes that subclass the COREModel, this is called *corification*. Several model including class diagram, sequence diagram, state machine and aspects can be corified. In this work, we use work, we use corified Reusable Aspect Model (RAM) to demonstrate the applicability and layout of concerns. A concern is composed of one feature model (COREFeatureModel), a subclass of COREModel. A feature model groups the commonality and variations of a family of software products such as authentication, logging, and authorization. Each software artifact in the family is represented as a

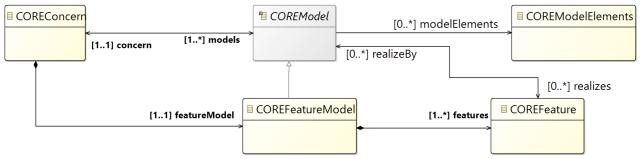


Figure 27: Basic Structure of a Concern

feature (COREFeature). In addition, a feature is realized by one or more COREModels while a COREModel can realize one or more core features.

6.2 CORE Feature Model and Weaving Algorithm

Feature Model is a tree diagram that specifies the commonalities and variabilities of a set of features belonging to a particular domain. The main objective of feature model is to enhance reuse during development. A simple example of a feature is an authentication as shown in Figure 28. The feature model has a root feature, authentication, with three children, namely: Biometric, Password, and Access Card. This feature model grouped different authentication means with password being compulsory. Based on feature selection (Configuration), different authentication means can be constructed from the model, e.g., password and biometric, password and access card, or password, biometric and access card. A feature model represents the variation interface of a concern. Each feature (e.g., password) has a realization model as depicted in the metamodel, Figure 27. The realization model can be a class diagram, sequence diagram, or aspects. Based on a particular configuration, the realizing model(s) of each selected feature is combined with other realizing models (*Weaving*). In CORE, bottom-up weaving algorithm is used, i.e., the composition of the models start from the the following configuration : Facial Recognition, Biometric, Password, Access Card, Magnetic, Bar Code, and Authentication; Facial Recognition, Bar Code, and Magnetic are weaved first in any order. It can be first Facial Recognition and Magnetic, and then "Facial Recognition"/Magnetic and Bar Code. During each weave, two models are combined to produce the merged model which is used to combine with the next model. The merged model of FR, BC, and Magnetic is weaved with either Biometric, Password, or Access Card. In each case, a merged model is produced and the trends continues until all the realizing models of the configuration are woven together. The composed model of the seven(7) artifacts produces the complete authentication application. The layout of the woven model is paramount to retain the original information which is currently lost with current automatic layout mechanisms. In subsequent section, we show how we use rpGraph to address this challenge.

6.3 Concern Layout Algorithm

This section describes the layout of a concern, i.e., woven model, using rpGraph. This algorithm is based on the weaving techniques of CORE, (see Section 6.2 in addition to the formalisms provided by the rpGraph (see Section 4.2, and is implemented in the TouchCORE

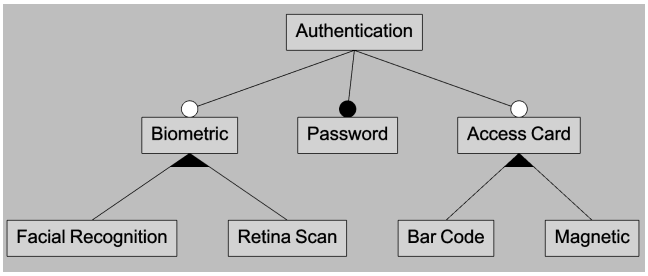


Figure 28: Basic authentication feature model.

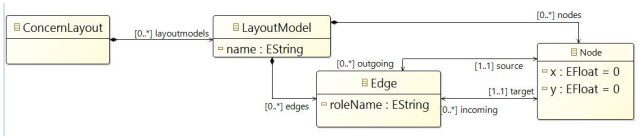


Figure 29: Concern layout metamodel (Excerpt shown)

tool (see Section 2.2). The following subsections detail the algorithm.

6.3.1 Create Concern Layout. The layout metamodel has been modified to support the Concern-Oriented Layout, excerpt shown in Figure 29. During each weaving iteration, two RAM models are supplied by the TouchCORE tool. These models are used to create Layoutout Model, (see Section 4.2.1, except that the edges are not assigned at this juncture. The Concern Layout is updated after each weaving iteration. This involves the modification of existing model elements and addition of new ones. For example, if the first weaving using the authentication feature model (see Figure 28 is Facial Recognition, i,e., Facial Recognition woven to empty authentication application. This weaving does involves only one model and hence no merging, but the Layout Model is added to the Concern Layout after the iteration. Next iteration could be authentication application, which currently contains only Facial Recognition, against Magnetic. The Layout Models of the authentication application instantiated as the reusing model while the Magnetic model is the reused model. Because the Concern Layout already contain the layout information of the authentication application (Facial Recognition Layout Model), the application model is updated to reflect the new status of the model elements after the second weaving. However, the layout of the Magnetic Model only added to the Concern Layout after the merge. The next section details the layout algorithm after the updated version of the Concern Layout.

6.3.2 Reposition the Concern Layout with rpGraph. In this section, we detail how we reposition the layout information of the concern models. The overall algorithm is shown in Figure 30.

6.3.3 Merge Concern Layout Models. It is evident at this juncture that the number of Concern Models increases as the weaving iteration increases. In this section, we describe how we merge the models to retain its original layout information using rpGraph. The operation "mergeopncernLayout" handles the repositioning algorithm. It has two parameters : m1 and m2, which are the merged two models after each weave. According the weaving techniques

```
1.  Operation mergeConcernLayout(LayoutModel m1, LayoutModel m2) {
2.      LayoutModel newM2 = m2
3.      LayoutModel copyM2 = m2.copy()
4.      merged = true
5.      //Reverse iteration of concern layouts, i.e., original models of m1
6.      For each newM1 in reverse : LayoutModel in ConcernModels {
7.        If (newM1.mappedElement != empty) {
8.            Update mappedElement
9.        }
10.       If (merged is true ) {
11.           //call rpGraph
12.           newM2 = rpGraph(newM1, newM2)
13.       } Else {
14.           //call rpGraph
15.           newM2 = rpGraph(newM1, copyM2)
16.       }
17.       DECLARE copy1 : LayoutModel, copy2 : LayoutModel
18.       If (newM2.mergeddElement = empty) {
19.           merged = false
20.           For each n : Node in newM2 {
21.           If (m1.contains(n)) {
22.               copy1.add(n);
23.           } Else {
24.               copy2.add(n);
25.           }
26.           }
27.           If (concernModels.previousLayoutModel != empty) {
28.               concernModelsCopy.add(copy1);
29.           } Else {
30.               concernModelsCopy.add(copy1);
31.               concernModelsCopy.add(copy2);
32.           }
33.       } Else {
34.           merged = true
35.           copyM2 = newM2.copy()
36.           If (concernModels.previousLayoutModel = empty) {
37.               concernModelsCopy.add(newM2);
38.           }
39.       }
40.     }
41.     If (concernModels == empty) {
42.       concernModels.add(m2);
43.     } Else {
44.       concernModels.clear();
45.       for each m : LayoutModel in concernModelsCopy {
46.             concernModels.add(m);
47.          }
48.         concernModelsCopy.clear();
49.      }
50.  }
```

Figure 30: Concern Layout Algorithm

in ToucCORE, the result of weaving is used as a model for the next weaving and so on. In order to retain the original information of these models, we treat merged model of the resultant weaving differently. In the first place, we store m2 as newM2 and create a copy of it, lines 2 and 3, because parameters needs not be edited during the execution. The variable concernModels stores the layout models of the concern. We iterated this variable in reverse order, in each section, we obtain two models that are merged using the rpGraph, lines 6 to 16. The result of the merging always returns a model but we separated it depending on the merging information. If the two models do not have direct relationship such as mapped elements or parent/child relationship, we do not merge their models, instead, we keep them separately. On this regard, if there is not relationship between them, we separate the model as "copy1" (m1 elements) and "copy2", lines 17 to 26. With the two separate models, we check to find out if concern models has previous model to be used for the merging. If there is next merging, we store the "copy1" in a list variable "concernModelsCopy", lines 27 to 29, because this model, copy1, will not be used for any subsequent mergings. However, if there is no more models to be merged, the two models : copy1 and copy 2 are stored in concernModelsCopy, lines 29 to 32. This concernModelCopy is used to update the concernModels after all merging. On the other hand, the result might have generated a model with merged elements or parent/child models are involved in the merge. In this case, we do not separate the resultant model, instead, we add it to concernModelsCopy if that is the last merge.

We also update copyM2 to the merged model as this is required in subsequent merging where the original m2 is required. The idea is that once there is a merged model with mapped elements, the result becomes the original models of m2. This iteration is continued until all the model in concernModels have been utilized.

The next step is to update the concern model before the layout is repositioned. During the first weaving, the m2 is weaved in empty concern, therefore, the concern model is empty. Thus, no model is merged, instead, we just add the m2 to concern models, lines 41 to 43. However, if the merging followls second weaving and beyond, the elements in concern layout are removed and the recent model elements are added, lines 43 to 49. Finally, the layout of concern model elements are repositioned to reflect the changes.

7 CONCLUSION AND FUTURE WORK

Software reuse is a fundamental technique to increase productivity, software quality and minimize development cost, time-to-market, and schedule overruns. This also applies to model-driven engineering, where models are the primary development artifact. For a certain class of modeling languages, reuse means merging two models into one and visualizing the merged result for the modeler. This is true for structural models and particularly aspect-oriented techniques and more generally model transformations with the intent to compose models based on a merge operator. However, state-of-the-art layout algorithms such as GraphViz do not consider the carefully crafted layouts of the original models and rather arbitrarily arrange the layout of the merged model, which leads to loss of information conveyed by the original layouts. This work proposes a new layout algorithm called rpGraph, which aims to retain the original layout information by taking the relative positioning of model elements into account.

rpGraph's layout metamodel is introduced, rpGraph's layout algorithm is explained with the help of example models, and rpGraph's resulting layouts of 20 model reuses from a library of reusable software models are assessed against the layouts created by GraphViz and JGraphX based on a set of layout metrics. These metrics measure the similarity of the relative positioning of model elements in the merged and original models, the number of overlapping model elements, the retention of the original model dimension ratios, the continued concentration of individual model elements from the original model, and the occurrence of empty space in the merged models. Based on these metrics, rpGraph outperforms GraphViz and JGraphX for the visualization of merged models.

In future work, we will verify our assumption that retaining the layout of the original models does not disrupt a modeler's mental map of the original models with an empirical study involving human subjects. Furthermore, we will study the effects of applying rpGraph to the visualization of reuses in large reuse hierarchies where many consecutive model merges have to visualized.

REFERENCES

[1] Wisam Al Abed, Valentin Bonnet, Matthias Schöttle, Omar Alam, and Jörg Kienzle. 2012. TouchRAM: A Multitouch-Enabled Tool for Aspect-Oriented Software Design. In *5th International Conference on Software Language Engineering - SLE 2012 (LNCS)*. Springer, 275 – 285.

[2] Omar Alam. 2016. *Concern oriented reuse: a software reuse paradigm.* Ph.D. Dissertation. McGill University, Montreal, Canada.

[3] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. 2013. Concern-Oriented Software Design. In *Model-Driven Engineering Languages and Systems*, Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 604–621.

[4] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. *Model-driven software engineering in practice.* Morgan & Claypool Publishers.

[5] CL Braun. 1994. Nato standard for the development of reusable software components, vol. 1/3. *NATO Communications And Information Systems Agency* (1994).

[6] Andrew Carton, Cormac Driver, Andrew Jackson, and Siobhán Clarke. 2009. Model-driven theme/UML. In *Transactions on Aspect-Oriented Software Development VI*. Springer, 238–266.

[7] Yosser El Ahmar, Sébastien Gérard, Cédric Dumoulin, and Xavier Le Pallec. 2015. Enhancing the communication value of UML models with graphical layers. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*. IEEE, 64–69.

[8] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. 2001. GraphViz open source graph drawing tools. In *International Symposium on Graph Drawing*. Springer, 483–484.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1 ed.). Addison-Wesley Professional.

[10] Emden Gansner, Eleftherios Koutsofios, and Stephen North. 2006. Drawing graphs with dot. (2006). https://graphviz.gitlab.io/_pages/pdf/dotguide.pdf

[11] Carsten Görg, Peter Birke, Mathias Pohl, and Stephan Diehl. 2005. Dynamic Graph Drawing of Sequences of Orthogonal and Hierarchical Graphs. In *Graph Drawing*, János Pach (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 228–238.

[12] Balázs Gregorics, Tibor Gregorics, Gábor Ferenc Kovács, András Dobreff, and Gergely Dévai. 2015. Textual diagram layout language and visualization algorithm. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*. IEEE, 196–205.

[13] OK Harsh and ASM Sajeev. 2006. Component-Based Explicit Software Reuse. *Engineering Letters* 13, 1 (2006), 30–39.

[14] Ralph E Johnson. 1997. Frameworks=(components+ patterns). *Commun. ACM* 40, 10 (1997), 39–42.

[15] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study.* Technical Report. Carnegie-Mellon University, PA, USA.

[16] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. 2009. Aspect-oriented multi-view modeling. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*. ACM, 87–98.

[17] Yongbeom Kim and Edward A Stohr. 1998. Software reuse: survey and research directions. *Journal of Management Information Systems* 14, 4 (1998), 113–147.

[18] Charles W Krueger. 1992. Software reuse. *ACM Computing Surveys (CSUR)* 24, 2 (1992), 131–183.

[19] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan MK Selim, Eugene Syriani, and Manuel Wimmer. 2016. Model transformation intents and their properties. *Software & systems modeling* 15, 3 (2016), 647–684.

[20] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. 1995. Layout adjustment and the mental map. *Journal of Visual Languages & Computing* 6, 2 (1995), 183–210.

[21] G. Mussbacher and J. Kienzle. 2013. A vision for generic concern-oriented requirements reuse RE@21. In *2013 21st IEEE International Requirements Engineering Conference (RE)*. 238–249. https://doi.org/10.1109/RE.2013.6636724

[22] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media.

[23] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media.

[24] Ulf Rüegg, Rajneesh Lakkundi, Ashwin Prasad, Anand Kodaganur, Christoph Daniel Schulze, and Reinhard von Hanxleden. 2016. Incremental diagram layout for automated model migration. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 185–195.

[25] Douglas C Schmidt. 2006. Model-Driven Engineering. *IEEE Computer* 39 (2006), 41–47.

[26] JGraph website. 2018. JGraphX Manual. (2018). https://jgraph.github.io/mxgraph/docs/manual_javavis.html#1.6.3

[27] JGraph website. 2018. mxGraph Tutorial. (2018). https://jgraph.github.io/mxgraph/docs/tutorial.html

[28] Kermeta website. 2018. Kompose : A generic model composition tool. (2018). http://www.kermeta.org/mdk/kompose/

[29] TouchCORE website. 2018. TouchCORE. (2018). http://touchcore.cs.mcgill.ca/

[30] Jon Whittle, Praveen Jayaraman, Ahmed Elkhodary, Ana Moreira, and João Araújo. 2009. MATA: A unified approach for composing UML aspect models based on graph transformation. In *Transactions on Aspect-Oriented Software Development VI*. Springer, 191–237.