VISUAL ABSTRACTION IN THE VISUAL DESIGN PROCESS

S. K. Chang, W. Hua and C. W. Yoo Department of Computer Science University of Pittsburgh, Pittsburgh, PA 15260 USA {chang, cwyoo, hua}@cs.pitt.edu

Abstract: We investigate the properties of syntactically well-formed, semantically valid and pragmatically admissable visual diagrams for the visual design process, supporting transformations among visual diagrams at different levels of abstraction. With this characterization of the visual design process, sound software engineering principles can be applied to the visual design process to effectively and efficiently accomplish the design objectives. This merger of visual design process with software engineering may also lead to the new discipline of visual software engineering.

Keywords: Visual abstraction, visual diagrams, software engineering principles for visual design, visual software engineering

1. Introduction

In many fields of science and engineering, designers rely on visual diagrams and their refinements to design products. For example in designing the relay circuit for a railroad control system, the designer will first draw a circuit diagram which is then transformed into logic equations and finally implemented by a microprocessor-controlled system. Although the designer can directly specify the logic equations, it is much more convenient to draw the relay circuit diagrams first. With CAD tools the creation and display of such visual diagrams is no longer a problem. However, visual diagrams are often much larger than the display screen, therefore the designer has to scroll the visual diagrams in order to view them. Moreover, the current CAD tools are inadequate in supporting the designer to visualize and/or conceptualize the visual diagrams, because it is usually difficult to provide various levels of abstractions for the visual diagrams. Since there is no adequate theoretical framework for visual abstraction, documentation and maintenance of visual diagrams then become a problem. On the other hand, adequate design tools should support multiple views in the design environment [7]. We are therefore faced with the following challenges: (a) how to formulate a theoretical framework for visual abstraction, (b) how to design software tools to provide various levels of abstraction and dynamic visualization, and (c) how to ensure the tools are adequate in supporting the designer's visual design process.

Diagram is a popular representation for visual languages. Daniel D. Hils [4] had a survey of data flow visual languages. Some research have been done to define the syntax of the visual language. In his papers [2, 3], Eric J. Golin uses the picture layout grammar to define the syntax of visual languages. A. Papantonakis and P. J. H. King [6] give the attribute grammar and semantics of their graphical query language. Joe Marks [5] also describes the syntax and semantics of the network diagram. But none of the above works considers visual abstraction. In this paper, our objective is to formally define visual diagrams at various levels of abstraction, which are syntactically well-formed, semantically valid and pragmatically admissable, so that techniques and guidelines can be developed to analyze and synthesize such visual diagrams. With such techniques and guidelines, we can apply sound software engineering principles to help a designer in his visual design process to effectively and efficiently accomplish the design objectives. In other words, design can be done in less time with fewer mistakes and is easier to understand by other designers.

2. A Conceptual Framework for the Visual Design Process

Our viewpoint is that *every instance of the visual diagram that the designer creates or sees on the display screen is a part of a visual program*. Thus the designer is constantly "programming" through continuous interaction with the visual diagram - although the designer may not know he (she) is "programming". This instance of the visual diagram is called a *visual sentence* [1]. Moreover, the designer does not merely interact with the visual diagram at a fixed level of abstraction. The visual sentence can be transformed into other visual sentences. Thus our viewpoint leads to a *transformational framework.*

Given a visual sentence, the designer can transform it into another visual sentence using a *vistractor* (*vis*ualization-and-abs*trac*tion opera*tor*) to accomplish different effects of visualization and abstraction. In the visual design process, the designer will first create a visual sentence as an initial design. By applying a vistractor, the designer can visualize the initial design at a different abstraction level. The initial design is improved to the detailed design and eventually led to the final design. Correspondingly, the abstracted initial design, detailed design and final design can also be constructed and visualized. This collection of visual sentences used in the visual design process is called a *visual design configuration*. Similar to a software configuration, the visual design configuration provides detailed documentation of the visual design process for verification, validation, maintenance and many other purposes.

In order to formally characterize the visual design process, in the following sections we will define the visual diagrams and the abstraction of visual diagrams using the aggregation vistractor.

3. Visual Diagrams

A *diagram* G is a graph that includes nodes N and arcs A. Generally the diagram has a hierarchical structure. A portion of the diagram can be compressed into an *aggregate node*. If the node is an aggregate node there will be a subdiagram inside this node. Conversely an aggregate node can also be expanded to a subdiagram.

There are two different types of nodes: A *complex node* s_c is the node that includes a diagram inside it, and a *port node* n_{port} is the node that does not include a diagram. Thus $N = N_c \cup N_{port}$ where N_c and N_{port} are disjoint subsets of complex nodes and port nodes, respectively.

Those complex nodes that include diagrams inside and are expandable to lower level diagrams are called *aggregate nodes*. A *regular* node is a complex node which has a self-contained diagram but its internal structure can not be visualized or expanded any more, while an aggregate node is expandable. The union of the set of aggregate nodes N_a and the set of regular nodes N_r is N_c.

A *port* n_{port} is the interface between the inside and the outside of the diagram G.

A port can be either an *input port* n_{in} or an *output port* n_{out}, and $N_{port} = N_{in} \cup N_{out}$. Since an aggregate node is a compressed diagram, and a regular node has a self-contained diagram inside, there are also ports in a complex node.

An *arc* is the link from an input port of a diagram or a complex node, to an output port of a diagram or another complex node.

Finally, each node is associated with a *label* (the name of the node) together with other attributes, and each arc is associated with the name of the arc together with other attributes. In general such labels may be strings of arbitrary length.

As an example, Figure 1(a) illustrates the visual diagram which may be used in designing a relay circuit, and Figure 1(b) illustrates a visual diagram which may be used in specifying a finite-state transition diagram. The two visual diagrams have the same graph so the syntax is the same, but they are used for different purposes so the semantics are different.

The following is the formal definition of a diagram G:

Definition: $G = (N, N_a, N_r, N_{in}, N_{out}, A, f_N, f_A)$, where

N is the set of nodes, $N = N_a \cup N_r \cup N_{in} \cup N_{out}$;
N_a is the set of aggregate nodes;
N_r is the set of regular nodes;
N_{in} is the set of input ports;
N_{out} is the set of output ports;
A maps *N* into the set of all subsets of N: $N \dashrightarrow 2^N$;
f_N is a mapping from nodes N to attribute space $K_{N_1} \times K_{N_2} \times \ldots \times K_{N_m}$;
f_A is a mapping from arcs A to attribute space $K_{A_1} \times K_{A_2} \times \ldots \times K_{A_n}$.

The pair (n_i, n_j) with n_j `in` A n_i is called an *arc* of G. The node n_i is the *initial node* and the node n_j is the *final node* of the arc (n_i, n_j). The node n_i is the *predecessor* of n_j, and n_j is the *successor* of n_i.

A *path* is a node-sequence $(n_1,n_2,\ldots,n_k)$, where n_{i+1} `in` An_i, for i=1 to k-1.

As a notational convenience, instead of $f_N(n_i)$ we will write $(k_{n_i}{}^1, \ldots, k_{n_i}{}^m)$. Similarly, instead of $f_A(a_j)$ we will write $(k_{a_j}{}^1, \ldots, k_{a_j}{}^n)$.

$f_N(n_i).name_j$ stands for $k_{n_i}{}^j$, j'th attribute of the node n_i, and $f_A(a_i).name_j$ stands for $k_{a_i}{}^j$, j'th attribute of the arc a_i,

N_c is the set of complex nodes and $N_c = N_a \cup N_r$.

N_{port} is the set of port nodes and $N_{port} = N_{in} \cup N_{out}$.

In Figure 2(a), for example, $N_r = \{n_1,n_2, n_4,n_5\}$, $N_a = \{n_3\}$, $N_{in} = \{n_0\}$, and $N_{out} = \{\ \}$. In this figure regular nodes are drawn as single circles, aggregate nodes as double circles and port nodes as squares.

If the initial or final node of an arc a_i is a complex node, then more specific information about interface ports is provided in the attribute space $f_A(a_i)$. For example, $f_A(a_i).initial$ and $f_A(a_i).final$ denote the internal port of initial node and the internal port of the final node in arc a_i, respectively.

An aggregate node n_a has the same definition as that of diagram G.

4. Abstraction of Visual Diagrams

Given a visual diagram, a user can manipulate it to obtain a new one, or to transform it into another visual diagram. In this section we describe the syntax of the diagram to formally characterize such a transformation by a vistractor called *aggregation*.

By convention G - N is the intersection of G and -N, where -N means the complement of N, and |G| represents the number of nodes in the diagram G. In what follows we will use the definition G=(N,A) or G=(N) for a diagram instead of formal one, as long as we do not need any other components of G.

If a node doesn't have any predecessor in G, it is called *input node* of the diagram. We define N_{input} to be the non-empty set of input nodes of G. An input port is a kind of input node.

The *inverse* of A is a function A^{-1}: $N \dashrightarrow 2^N$ defined as:

$$A^{-1}\ n_i = \{\ n_j \mid n_i \texttt{ in } A\ n_j\}$$

We define A^2, A^3, ... , A^+ and A^* as:

$A^2\ n_i = A\ (\ A\ n_i\ )$
$A^3\ n_i = A\ (A\ (\ A\ n_i\ )\ )$
...

$A^+ n_i = A\ n_i \cup A^2\ n_i \cup ...$
$A^* n_i = n_i \cup A\ n_i \cup A^2\ n_i \cup ...$

A node n_j is *reachable from* n_i if n_j is in $A^* n_i$. A node n_j is *accessible* if it is reachable from one input node in N_{input}, and it is *inaccessible* otherwise.

The *connectivity function* C: *N* --> 2^N is defined as:

$C\ n_i = A\ n_i \cup A^{-1}\ n_i$

Lemma 1: For every two nodes n_i and n_j, if n_i `in` C n_j, then n_j `in` C n_i, so the connectivity function C is symmetric.

Proof: If n_i `in` C n_j, then by the definition n_i `in` ($A\ n_j \cup A^{-1}\ n_j$). Case-1: if n_i `in` $A\ n_j$, then n_j `in` $A^{-1}\ n_i$, so n_j `in` C n_i. Case-2: if n_i `in` $A^{-1}\ n_j$, then n_j `in` $A\ n_i$, so n_j `in` C n_i. In both cases, n_j `in` C n_i, so the function C is symmetric. **Q.E.D.**

Two nodes n_i and n_j are *adjacent* if n_i `in` C n_j or n_j `in` C n_i.

C^* is the reflexive and transitive closure of C.

A diagram G is *connected* if for every two nodes n_i and n_j, n_i `in` C^* n_j.

A diagram G is *well-formed* if the following two conditions hold:

1) it is connected, and
2) every node is accessible

Given a diagram G and a node n_i, we define SCR(n_i) as the maximal strongly connected subdiagram which includes the node n_i. A diagram is *cycle-free* if it does not contain any strongly connected subdiagram. A node n_i is *self-cyclic* if n_i `in` A n_i.

Given a well-formed diagram G=(N,A) and a subdiagram G'=(N',A'), there can be an arc between inside and outside of the subdiagram. In other words, inside of the subdiagram, there will be a interface node whose predecessor or successor is in the outside of the subdiagram. We define N'_{entry} to be the *entry nodes* of G', and N'_{exit} the *exit nodes* of G' as follows:

$N'_{entry} = \{ n_i\ in\ N' \mid A^{-1}\ n_i$ `is not subset of` $N' \}$
$N'_{exit} = \{ n_j$ `in` $N' \mid A\ n_j$ `is not subset of` $N'\}$

Given a well-formed diagram G=(N,A), a subdiagram G' is *aggregatable* if the following four properties hold:

1) G' is connected,
2) every node in N' is reachable from at least one node in $N'_{entry} \cup N'_{input}$,
3) G'-($N'_{entry} \cup N'_{input}$) is cycle-free, and
4) for every node n_i in N'-($N'_{entry} \cup N'_{input}$), $A^{-1}\ n_i$ is a subset of N'

By this definition, if a subdiagram G' is aggregatable then all incoming arcs are through these entry nodes, and all cycles within G' must contain at least one entry node n'_{entry} in N'_{entry}.

Given a diagram G and a subdiagram G', we use the notation G/G', where the G' is aggregatable, as the *aggregated diagram* in which the aggregatable subdiagram G' is replaced by an aggregate node.

In the visual design process, a subdiagram can be identified by clicking and dragging interactively as in Figures 2(a) and 4, where the dotted rectangle represents a dragged subdiagram. If a given subdiagram is aggregatable, the subdiagram can be compressed into an aggregate node.

In Figure 2(a), for example, the whole diagram G=($\{n_0,n_1,n_2, n_3,n_4,n_5\}$,A) is well-formed, because it is connected and every node is reachable from the input node n_0. The subdiagram G_{234} is aggregatable, as it is connected, every node of G_{234} is reachable from the entry node n_2, G_{234}-$\{n_2\}$ is cycle-free, and all the predecessors of the nodes n_3 and n_4 are inside the subdiagram. Figure 2(b) is the aggregated diagram, where the subdiagram G_{234} has been compressed into the aggregate node n_{234}. Figure 3 shows the internal diagram for the aggregate node n_{234}, with the input/output ports added.

For a well-formed diagram G=(N,A) and an aggregatable subdiagram G'=(N',A'), the *aggregation process* X(G,G'), which produces an aggregated diagram G/G', is defined as belows:

1) Assume that the aggregatable subdiagram is compressed into an aggregate node n_a, the nodes of X(G,G') = (N-N') $\cup$ $\{n_a\}$.

2) All arcs (A-A') belong to the arcs of X(G,G'). If there is an arc whose initial or final node is in N', then it's initial or final node is replaced by the aggregate node n_a.

3) The syntax and attributes of the aggregate node n_a are inherited from the subdiagram G'.

In the above definition, the syntax and attributes of the aggregate node are preserved at the attribute space of the node to keep the same characteristics as one of the subdiagram compressed. The preserved characteristics are used for an expansion or de-aggregation of the diagram.

Theorem 1: Given a well-formed diagram G and an aggregatable subdiagram G', the aggregated diagram G/G' is also well-formed.

Proof: Let G_1 be the aggregated diagram G/G' and n_a the aggregate node of G_1 which is compressed from G'. If G=G', then $|G_1|=1$, and n_a is the unique node of G_1. Therefore, the G_1 is connected and accessible. We will now prove the case of G not equal to G'. Since the diagram G is connected, for every two nodes n_i of G' and n_j of (G-G'), n_i in C^* n_j, and also n_j in C^* n_i by symmetry. And by the definition of aggregation process, every arc (n_i,n_k) and every arc (n_k,n_i) of G, such that n_i in G', are replaced by (n_a,n_k) and (n_k,n_a), respectively. In G_1, therefore, for every node n_i of $(G_1-\{n_a\})$, n_a in C^* n_i and also n_i in C^* n_a by symmetry. For every two nodes n_i and n_j, n_i in C^* n_a and n_a in C^* n_j. So n_i in C^* n_j by transitivity, and also n_j in C^* n_i by symmetry. Therefore the aggregated diagram G_1 is connected. Since G is accessible, every node of G is reachable from one input node of G. Let $p=(n_1,n_2,...,n_{k-1},n_k)$ be a path from one input node n_1 of G to the node n_k of G', such that every node n_i in G for i=1~k-1, and n_k in G'. By the definition of aggregation process, the arc (n_{k-1},n_k) is replaced by (n_{k-1},n_a). So in G_1, there exists a path $(n_1,n_2,...,n_{k-1},n_a)$, which means that n_a is reachable from one input node n_1. Assume that every node n_i in $(G_1-\{n_a\})$ of G_1 is reachable from both n_a and one input node of G_1. Hence, every node n_j in $A^{-1}n_i$ is also not reachable from one input node, a contradiction to the accessibility of the well-formed diagram G. Therefore every node n_i in $(G_1-\{n_a\})$ of G_1 is reachable from either n_a or one input node of G_1. If n_i in A^*n_a, then the node n_a is always reachable from one input node of G_1, and n_i is also reachable by transitivity. Hence the diagram G_1 is accessible. **Q.E.D.**

A *series of aggregations* can be applied to a diagram. In Figure 4, for example, the three diagrams G_0, G_1 and G_2 constitute a series of aggregations. The diagram G_0 of Figure 4(a) can be aggregated to G_1 of Figure 4(b), and then finally to G_2 of Figure 4(c). The nodes n_2, n_3 and n_4 are aggregated to the node n_{234}, and all the nodes of G_1 are aggregated to the node n_{123456} of G_2. Conversely, nodes n_{123456} of G_2 can be expanded into the diagram G_1.

Once a subdiagram of G is given by an user and then checked whether the subdiagram is aggregatable or not, the aggregation process is applied to the diagram. The size of the subdiagram and the order of aggregation processes are determined through the user's interactions. An undisciplined user may not find a proper series of aggregations, and sometimes a user may not give a unique series of aggregation to generate a certain aggregated diagram. For example, in Figure 4, the two nodes n_2 and n_3 can be aggregated first, and then to n_{234} with n_4 next. This shows that there exists another series of aggregations which has a different level and depth, to produce the same aggregated diagram as the one of Figure 4. This kind of abstraction level and style may vary from designer to designer.

The following algorithm accepts an aggregatable subdiagram G' of G and produces another aggregatable subdiagram G" such that |G"| >= |G'| and G' is subset of G".

Algorithm 1 (EXTEND_AGGREGATABLE_SUBDIAGRAM):
Input: a well-formed diagram G=(N,A) and an aggregatable subdiagram G'=(N',A')
Output: an extended aggregatable subdiagram of G'
Method:

```
N" = ENTRY_NODES(G,G');
A" = A';
change = true;
while (change)
        change = false;
        for each node ni in N'
                for each node nj in Ani
                        if (nj is not in N") and (A-1nj is subset of N")
                                N" = N" ∪ {nj};
                                nj is also in A"nj;
                                change = true;
return(N",A");
```

Algorithm 1 iterates through successor nodes of the entry nodes of G', applying the four properties of aggregatable diagram until no more changes take place. For example, in Figure 4, if G'={n_2,n_3}, then Algorithm 1 will extend it to the set {n_2,n_3,n_4}. So the algorithm adds as many nodes as possible to the given aggregatable subdiagram G' if it satisfies the above mentioned four properties, that is, the algorithm produces a submaximal and unique aggregatable subdiagram including G'. Therefore, the algorithm will make it easy to find a series of aggregation and to reduce the aggregation steps.

Lemma 2: If a diagram G is made up of only one node which is self-cyclic, then the diagram is aggregatable.

Proof: The diagram G is clearly connected and has only one entry node, which is accessible by the definition of aggregatable diagram. G-{n} is empty, cycle-free, and is of no predecessors. Hence the diagram is aggregatable. **Q.E.D.**

Lemma 3: Given a well-formed diagram G which is not a degenerated graph of a single aggregate node, there exists a subdiagram G' of G which is aggregatable.

Proof: Since the diagram G is connected, we can find a topological order $(n_1,n_2,...,n_{k-m},...,n_k)$ for all nodes of G such that n_{k-m} in $A^{-1}n_k$ and n_{k-m+1} *is not in* $A^{-1}n_k$ for i=1~m-1, that is, n_{k-m} is the nearest predecessor of n_k. If the node n_k is self-cyclic then the subdiagram G' which is made up of only one node n_k is aggregatable by Lemma-2. If the node n_k is not self-cyclic then we can construct the subdiagram G' which is made up of two node n_{k-m} and n_k. Now we will show that G' is aggregatable. Case-1: if n_{k-m} is the only predecessor of n_k, then by the topology the node n_{k-m} will be either an entry node or an input node of G', as the subdiagram G' is well-formed and the node n_{k-m} is reachable from one input node of G. Since G' is connected, both nodes n_{k-m} and n_k are reachable from n_{k-m}, and G'-$\{n_{k-m}\}$ is cycle-free by the assumption such that the node n_{k-m} is not self-cyclic. And $A^{-1}n_k$ is a subset of G', so the subdiagram G' is aggregatable. Case-2: if there exists a predecessor of n_k outside of G', then n_k will be an entry node of G' by the definition of entry node. And by the topology, the node n_k will be either an entry node or an input node of G', as mentioned above Case-1. Since G' is connected, both n_{k-m} and n_k are reachable from the node n_{k-m}, and G'-$\{n_{k-m},n_k\}$ is empty and cycle-free, so the subdiagram G' is aggregatable. **Q.E.D.**

Theorem 2: If a diagram G is well-formed, then the diagram can be compressed into one aggregate node.

Proof: By Lemma 3, there exists an aggregatable subdiagram G' of the well formed diagram G. And by Theorem-1 its aggregated diagram G/G' is also well-formed. So there exists a series of aggregations $(G_0,G_1,...,G_n)$ such that G_0=G, G_i=X(G_{i-1}, $G'^{(i)}$) and G_i is not equal to G_{i-1} for i=1~n. And by the definition of the aggregation process, $|G_i| >= |G_{i-1}|$ for i=1~n. Since number of arcs and nodes of G are finite, we can conclude that $|G_n|$=1 and n < K for a certain constant K. **Q.E.D.**

Definition: A diagram is *completely aggregatable if there exists a series of aggregations $(G_0,G_1,...,G_n)$ such that $|G_n|$=1 and n For example, Figure 4 shows that the diagram G_0 is completely aggregatable, as it can be compressed finally into one aggregate node with the series of aggregation (G_0,G_1,G_2).*

Theorem 3*: A diagram G is well-formed if and only if it is completely aggregatable.*

Proof*: If the diagram G is well-formed, then by Theorem 2 the diagram is completely aggregatable. If the diagram G is completely aggregatable, then by the definition of aggregatable diagram it should be connected. Assume that G is not accessible. Since there exists a node n which is not reachable from one input node of G, we can not find any aggregatable subdiagram which includes the node n. Since the diagram G can not be aggregated any more, it is not completely aggregatable, a contradiction.* ***Q.E.D.***

The above lemmas and theorems show that well-formedness and aggregatability are equivalent concepts.

5. Valid and Admissable Visual Diagrams

A well-formed diagram is syntactically correct, but additional constraints must be satisfied so that the diagram can be semantically valid. We now describe the constraints for a diagram to be semantically valid.

V1. Arcs and nodes must be compatible. If the nodes n_{i1} and n_{i2} have attributes x_{i1} and x_{i2}, respectively, and arc a_i =(n_{i1}, n_{i2}) has attributes y_i, then (x_{i1}, x_{i2}, y_i) is in constraints set R.

V2. Every input port in N_{in} can not be used as a final node of an arc, and every output port in N_{out} can not be used as an initial node of an arc.

V3. The intersection of N_{in} iN_{out} is empty.

V4. The visual diagrams must be unambiguous. Let a_i = (n_{i1}, n_{i2}), a_j = (n_{j1}, n_{j2}) be two arcs of G. If $f_N(n_{i1}) = f_N(n_{j1})$ and $f_N(n_{i2}) = f_N(n_{j2})$, then $f_A(a_i)$ is not equal to $f_A(a_j)$.

V5. Every interface port of a complex node can not be used for both input port and output port of the complex node. Let a_i = (n_{i1}, n_{i2}), a_j = (n_{j1}, n_{j2}) be two distinct arcs of G, where n_i,n_j in N_c. If $f_N(n_{i1}) = f_N(n_{j2})$, then $f_A(a_i)$.initial is not equal to $f_A(a_j)$.final. Similarly if $f_N(n_{i2}) = f_N(n_{j1})$, then $f_A(a_i)$.final is not equal to $f_A(a_j)$.initial.

V6. If an interface port of a complex node is used for an input inside, then its indegree should be no more than one. That is, if a_i = (n_{i1}, n_{i2}), a_j = (n_{j1}, n_{j2}) be two distinct arcs of G, where n_{i2},n_{j2} in N_c, and if $f_N(n_{i2}) = f_N(n_{j2})$, then $f_A(a_i)$.final is not equal to $f_A(a_j)$.final.

In addition to the above described semantic constraints, there are also pragmatic constraints, because we must put some constraints on the diagram drawn by the user in order to obtain a pragmatically admissable diagram.

A1. |N| < MAX_NODE_NUM

|N| means the cardinality of the set N, and this constraint means the number of nodes in one diagram should not exceed a maximum number.

A2. constraints on node degree

$f_N(n)$.degree < MAX_NODE_DEGREE

$f_N(n)$.degree means the degree attribute of the node n.

Assume n in N_a, we define the diagram which describes n as G_a=lookdown(n)=(N^a, $N_a{}^a$, $N_r{}^a$, $N_{\{in\}}{}^a$, $N_{\{out\}}{}^a$, A^a, $f_N{}^a$, $f_A{}^a$).

G and G_a should satisfy:

a) $|N_{\{in\}}{}^a \cup N_{\{out\}}{}^a| < MAX\_NODE\_DEGREE$.

b) $f_N(n).degree=|N_{\{in\}}{}^a \cup N_{\{out\}}{}^a|$

c) $f_N(n).degree\_in=|N_{\{in\}}{}^a|$

d) $f_N(n).degree\_out=|N_{\{out\}}{}^a|$

e) $f_N(n).degree\_in>0$, for all n in $N - N_{\{in\}}$

f) $f_N(n).degree\_out>0$, for all n in $N_{\{in\}}$

A3. $|N_{\{in\}}| > 0$, $|N_{\{out\}}| >= 0$

Every diagram should have at least one input port, which allows the digram to be connected to the other diagram or the environment. This constraint is also a connectivity constraint.

In addition to the above pragmatic constraints, we may also add other informal constraints to incorporate certain design guidelines (see Section 7).

6. The Visual Design Process

In the previous sections we defined well-formed, valid, admissable visual diagrams. We can develop analysis algorithms for deciding whether a visual diagram is well-formed, valid and admissable. Only such diagrams will be in the collection of diagrams maintained by the system to form a visual design configuration. The analysis algorithms can be formal or informal. For syntactic checking, formal algorithms that are checked automatically are readily available:

***Algorithm 2 (CHECK_WELL_FORMEDNESS)**:*
Input: a diagram G=(N,A)
Output: "yes" if well-formed, "no" otherwise
Method:

```
1) if not CONNECTED(G) return("no");
2) for each node n_i in N
       if not REACHABLE(n_i,N_input)
              return("no");
3) return("yes");
```

***Algorithm 3 (CHECK_AGGREGATABILITY)**:*
Input: a well-formed diagram G=(N,A) and a subdiagram G'=(N',A')
Output: "yes" if the subdiagram G' is aggregatable, "no" otherwise.
Method:

```
1) if not CONNECTED(G') return("no");
2) N'_entry = ENTRY_NODES(G,G');
3) for each node n_i in N'
       if not REACHABLE(n_i, N'_entry) return("no");

4) if not CYCLE_FREE(G'-N'_entry) return("no");
5) for each node n_i in (N'-N'_entry)
       if PREDECESSORS(n_i) is not a subset of N'
              return("no");
6) return("yes");
```

It is also straightforward to develop an algorithm to check whether a visual diagram is valid and admissable. Thus the visual design configuration can be documented and maintained following sound software engineering principles.

As mentioned above, the aggregate node is based upon the concept of abstraction. With aggregate nodes, the designer can work on the design problem at any desirable level of abstraction. We can apply the above described analysis algorithms to help the designer synthesize visual diagrams correctly. The aggregation vistractor enables the designer to generate diagrams at different levels of abstraction. Informal or heuristic rules can also be incorporated into the analysis algorithms, so that they serve as design guidelines for the designer.

7. Discussion

Visual diagrams are only one type of visual languages which are becoming increasingly important in human-computer interaction. Compared to traditional programming languages, in general visual languages are simpler to learn and use, avoid the need for

experienced user, interact more effectively, and are more appealing to the average user. Even though visual languages are rapidly gaining importance, there is yet to be developed a standard set of criteria that measures the quality of visual languages. In [1] S.K. Chang and P. Mussio proposed a practical set of design criteria for visual languages, based upon the theory of visual sentences as programs. These criteria were used to rate the quality of nine popular software and the visual programming software developed by the Visual Computer Laboratory at the University of Pittsburgh. The results show a trend of improved ratings from older software to more recent ones. Also, the ratings help us understand how practical the proposed criteria are in measuring the quality of visual languages. These criteria are based upon the theory of visual languages with emphasis on adequate communication between human and computer or between human and human. Thus it is a principled approach. We will further refine the criteria based upon the above framework of visual abstraction, aided by further experiments and empirical observations. These evaluation criteria will lead to the formulation of constraints (the DO's and DONT's) for the designer to follow, to be incorporated into the "guidebook" of the visual design process.

Conversely, the visual design process may also lead to a visual software design process and become useful to software design in general. Traditionally, visual design is used only at the initial design phase in the software design process. A refined visual design process may facilitate the incorporation of visual design in every phase of software design, leading to what we tentatively call visual software engineering. The exploration of visual software engineering will become increasingly important, aided by the merger of software engineering principles with the visual design process.

References

[1] S. K. Chang and P. Mussio, "Customized Visual Language Design", Proc. of Eighth Int'l Conference on Software Engineering and Knowledge Engineering, June 10-12, 1996, Lake Tahoe, Nevada, 553-562.

[2] Eric J. Golin, "Parsing Visual Languages with Picture Layout Grammars", Journal of Visual Languages and Computing, Vol. 2, No. 4, pp. 371-393, 1991.

[3] Eric J. Golin and Steven P. Reiss, "The Specification of Visual Language syntax", Journal of Visual Languages and Computing, Vol. 1, No. 2, pp. 141-157, 1990.

[4] Daniel D. Hils, "Visual Languages and Computing Survey: Data Flow Visual Programming Languages", Journal of Visual Languages and Computing, Vol.3, No. 4, pp. 69-101, 1992.

[5] Joe Marks, "A Formal Specification Scheme for Network Diagrams That Facilitates Automated Design", Journal of Visual Languages and Computing, Vol. 2, No. 4, pp. 395-414, 1991.

[6] A. Papantonakis and P. J. H. King, "Syntax and Semantics of Gql, a Graphical Query Language", Journal of Visual Languages and Computing, Vol. 6, No. 1, pp. 3-25, 1995.

[7] P. Roesch, "User Interaction in a Multi-View Design Environment", IEEE Symposium on Visual Languages, Sept 3-6, 1996, Boulder, 316-323.

***Technical Report TR 97-02**, February 1997, Department of Computer Science, University of Pittsburgh.*

Figures: Figure-1a, Figure-1b, Figure-2a, Figure-2b, Figure-3, Figure-4a, Figure-4b, Figure-4c.