SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 95-28

Visualisation and Software Development: Analysing the requirements

T. Jones, D. Carrington, W.Allison,
L. Stewart-Zerba, G. Watson and J. Welsh

June 1995

Phone: +61 7 365 1003
Fax: +61 7 365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from ftp.cs.uq.edu.au in the directory /pub/SVRC/techreports.

Visualisation and Software Development: Analysing the requirements

T. Jones, D. Carrington, W. Allison,
L. Stewart-Zerba, G. Watson and J. Welsh.

Email: {tsj, davec, warwick. larry, gwat, jim}@cs.uq.oz.au

Software Verification Research Centre
Department of Computer Science
University of Queensland
Queensland, Australia 4072

May 1995

Key Words: visualisation, graph drawing requirements, software development tools.

Abstract

As part of the software development process, software engineers produce numerous documents, ranging from initial specification documents to documents associated with implementation. Some of these documents are diagrammatic in nature, such as traditional design documents. Others are textual such as program source code. Textual software engineering documents, particularly source code, contain relational structures which are most naturally presented diagrammatically. This paper investigates the types of diagrams that are commonly used by software engineers throughout the life of software. To establish the requirements of such diagrams in a generic manner, a set of review criteria was created. Our goal is to establish display requirements for general-purpose tools that will provide software engineers with diagrammatic views of these documents.

1 Introduction

Diagrammatic presentations are one of the most powerful methods of human communication. With the ever increasing power of the underlying hardware, diagrammatic representation of information is becoming a predominant mechanism for communicating complex ideas for a large range of software applications. This is true for tools and environments that support software development.

Software engineers have for some time taken advantage of the power of diagrammatic presentations through the use of various diagramming techniques during the early phases of the software development. Such techniques are supported by a variety of Computer Aided Software Engineering (CASE) tools [Ber93, Fug93]. However with the increasing complexity and size of software, programmers and analysts are turning to tools that provide diagrammatic representations of relationships that exist within and between the documents that comprise the software they are developing [Amb89]. In this paper we identify two distinct areas in which diagrams can be used by software engineers during development. They are: modelling diagrams used in the early phases of software development and program visualisation.

Modelling diagrams representing program structure, control flow, and data have always been used by programmers to record design decisions about a program. Such diagrams are usually drawn before the program is implemented. Program visualisation tools make it possible to generate diagrams that are similar in nature to modelling diagrams from the source code itself, therefore providing a basis for comparing *what is* and *what was meant to be*.

Originally software development tools were stand alone applications. As such, the programmers chose those tools that were most appropriate for their situation. Recently there has been a trend to integrate software development tools into single software development environments. Although this practice has benefits, it confines programmers to particular styles and methodologies regardless of how applicable they are. To address this, many environments allow configuration of some aspects by the end user. Such configuration varies from setting previously defined options at run-time, to a more complex configuration performed before a project is started or when the software is installed. The next generation of software development environments will be almost completely generic in nature, allowing document types, interaction and presentation styles, and associated tools to be specified. This paper is based on a more extensive working paper [Jon95] that reviews the requirements for diagrammatic presentations in generic software development environments.

To establish the diagrammatic presentation requirements in a generic manner, a set of review criteria is described in *section 2*. In [Jon95], a taxonomic classification was used to establish a selection of modelling diagrams and program visualisations, to which the review criteria were applied. The classifications and the results from the more interesting examples are described for modelling diagrams in *section 3* and for program visualisations in *section 4*.

2 Analysis approach

This section describes the review criteria that were applied to modelling diagrams and program visualisations. We define four criteria:

- Graph theoretic class;
- Conformance;
- Presentation; and
- Interaction.

In the four sub-sections, each criterion is defined and a brief description is given describing its importance to the presentation and manipulation of modelling diagrams and program visualisations in software development environments.

2.1 Graph Theoretic Class

The first criterion is the type(s) of graph that forms the structure of the diagram. This is important for establishing the basic requirements for any drawing package that might be used to present such diagrams. We have chosen a subset of graph types that we feel are relevant for this task.

A simple graph is a set of *nodes* N and a set of *edges* E where each edge is a pair of distinct nodes in N. The set of simple graphs can be subdivided in several different ways.

- *Directed* and *undirected*: A simple graph is directed if every edge is an ordered pair of distinct nodes. For an undirected graph, each edge is an unordered pair.
- *Cyclic* and *acyclic*: A *path* is an ordered list of nodes n_1 to n_k such that (n_1,n_2), $(n_2, n_3)...(n_{k-1}, n_k) \in E$. The length of the path and the number of edges in that path are given by *k-1* and the number of nodes is *k*. A *cycle* is a path where $n_1 = n_k$ and $k>1$. A graph that contains a cycle is called cyclic.
- *Connected* and *disconnected*: An undirected graph is connected iff, for every pair of nodes in the graph, a path exists between them. A directed graph is connected if its undirected equivalent is connected.
- *Planar* and *non-planar*: A graph that can be drawn on a plane such that no edges cross is said to be planar. The planarity of a graph is a key issue for its presentation.

These four attributes are used to classify graphs.

A *tree* is a specialised form of connected graph where if any edge is removed, the resulting graph is disconnected. For this reason they are also known as minimally connected graphs. Trees are always connected, acyclic and planar and may be directed or undirected.

A *rooted tree* is a tree with a specific node designated as its root. A rooted tree has an implicit direction associated with its edges and hence is a directed graph. Rooted trees are always directed, connected, acyclic and planar.

A *multigraph* is a generalisation of the simple graph concept to allow edges from a node to itself and multiple edges between any pair of nodes. Multigraphs may or may not be directed, cyclic, connected and planar.

A *compound graph* is a graph that consists of distinct sub-graphs, connected or disconnected. This category is not drawn from graph theory but is included because of the hybrid nature of many diagrammatic techniques used in software engineering. These distinct sub-graphs are often connected by a sub-graph which is a tree.

2.2 Conformance

The second criterion considers the syntax and semantics that affect the layout of a graph that are not captured in its type description. Simple graph layout mechanisms produce diagrams based on a set of aesthetic goals only. They do not consider, nor do they have any knowledge of, the syntax and semantics of the diagram being drawn. Regardless of the layout approach chosen, it is necessary to provide this information so a resulting *view* of a diagram conforms to the syntax and semantics of that diagram. We identify two key criteria, geometric relations and absolute positioning.

2.2.1 Geometric Relations

Geometric relations are constraints that specify geometric relationships between nodes. There are two type of geometric relations:

- Relative positioning; and
- Clustering.

Relative positioning constrains a node's position in relation to another node. For example, we may constrain a node X to be to the left/right/above/below node Y. Such constraints may be general like those just listed, or more specific such as 'node X must be the left neighbour of node Y'.

Clustering groups nodes within a defined geometric region. As with individual nodes, a cluster can have a relative positioning constraint. That is, a node or a cluster may be geometrically constrained in relation to another node or cluster. Similarly nodes within a cluster may also be subject to relative positioning constraints within the cluster itself. Clustering is hierarchical in the sense that clusters and nodes may be grouped together within a defined geometric region to form a new cluster.

2.2.2 Absolute Positioning

Absolute positioning constrains a node or cluster to a fixed position within an assumed coordinate system. For example, a diagram, which is a rooted tree, may require the root node be placed at the top of the diagram, centred left to right.

2.3 Presentation

The third criterion considers the presentation attributes of the visual objects in a diagram. Presentation attributes do not contain information that affect the initial layout of the graph,

rather they are the visual encoding of the information being presented in a diagrammatic form. These attributes are either geometric or appearance attributes.

2.3.1 Geometric Attributes

Geometric attributes describe physical geometric properties of a visual object. They may or may not require fine tuning of the layout of a diagram. Examples of geometric attributes are:

- *shape of nodes* - are the nodes round, square etc.?
- *size of nodes* - what are the dimensions of nodes?
- *type of arc* - what is the shape and format of arcs?
- *connection of arcs to nodes* - where do arcs connect to nodes?
- *annotation of nodes* - how and where are nodes annotated? and
- *annotation of arcs* - how and where are arcs annotated?

2.3.2 Appearance Attributes

Appearance attributes describe the physical appearance of a visual object. They have no geometric properties and as a result do not influence layout. Examples of appearance attributes are:

- *colour* - what colour are nodes and arcs?
- *line width* - how wide are the lines that make up nodes/arcs?
- *#lines* - how many lines are drawn for each node/arc shape?
- *filled* - are nodes/arcs filled? and
- *texture* - what texture are nodes and arcs?

2.4 Interaction

The fourth and final criterion considers what role a view of a diagram may play in interaction between the user and the development environment. Interaction can be considered to belong to one of the following two classes:

- *Manipulative* - the view can be used to manipulate (i.e. create, cut, paste, copy etc.) both the diagram structure, and/or the underlying document structure.
- *Non-manipulative* - the view can be used only to call another view, invoke a tool or both.

Four types of visual roles are considered, views with no interaction, interactive views that do not allow manipulation, views that allow restricted manipulation, and views that allow all manipulation activities.
View-only views are straight diagrammatic presentations with no interactive properties. Such views can be called and dismissed, but do not allow the user to interact with the diagram itself. *Interactive views* allow the user to interact with the diagram, but do not change the document structure directly. Interactive views allow non-manipulative interactions.

Restricted manipulative views allow manipulative interaction in some cases and may also allow non-manipulative interaction. Manipulation in this class may be restricted because either the underlying diagram structure is affected by an operation that breaches an established constraint, or the role of the view in question does not support certain manipulation. An allowed manipulation will be reflected in both the document and diagram structures if it does not violate an existing constraint.
Unrestricted manipulative views do not place any restriction on what manipulations are allowed and may also allow non-manipulative interaction. A manipulation on this type of view can change all related document and diagram structures.

3 Classification of Modelling Diagrams

In the early phases of software development, analysts and programmers use a variety of modelling diagrams to describe the problem domain of a system, and eventually the design of the system itself. Over the years, a large number of such diagrams have evolved, each with a diverse set of notations and presentation styles. For this reason, we only review a subset of such diagrams in this paper. To select an appropriate subset, we consider a classification of modelling diagrams.

3.1 A Classification of Modelling Diagrams

Modelling diagrams are used in software engineering at several levels. Rock-Evans and Engelien [Roc89] provide a taxonomy of modelling diagrams at the analysis level. Martin and McClure [Mar85] define a taxonomy for both analysis and design. The modelling diagrams described in these texts are similar despite the modelling level at which they are used. Both taxonomies differentiate between diagrams on the basis of the software engineering issues that they address. Both identify the need to model data and activities or functions separately. Other common modelling techniques are techniques for modelling the detailed logic of a program and techniques for modelling the changes in a system's state.

We have identified four categories of modelling techniques which play an important role in analysis and design. They are data modelling, activity modelling, program logic modelling and state-transition modelling. These four modelling techniques tackle important issues present in software development.

For each of the categories, it was appropriate to define a sub-classification. The sub-classifications reflect the purpose and detail of the models. Data modelling diagrams include analytical diagrams, design and object-oriented diagrams. Activity-modelling diagrams include data-flow diagrams, activity-decomposition diagrams and activity-dependency diagrams. State-transition modelling diagrams include simple state diagrams and state-charts. Program logic diagrams include those drawn with only lines, only boxes, and boxes and lines.

3.2 Analysis

To illustrate how the review was conducted, this section describes the extent of the review of modelling diagrams, provides an example of how the review criteria were applied to each diagram reviewed, and summarises the results of the full review.

3.2.1 Extent of the survey

Data modelling is performed at two different levels during software development. Firstly, *analytic modelling* is concerned with describing relationships between real world objects/ entities. Secondly, *design-oriented* modelling is concerned with describing how data is structured in the implementation of the program. A recent modelling technique, known as *object modelling* provides a smooth join between these two phases. When considering analytic modelling, we reviewed Entity-Relationship (ER) diagrams [Che76, Elm89]. When considering design-oriented modelling we reviewed Jackson data diagrams [Jac83]. When considering object modelling, we reviewed the Object Modelling Technique's (OMT) [Rum91] object diagrams.

Activity modelling explores the relationships within and between activities. By classifying activity modelling diagrams by the type of information they convey, it is possible to identify classes of activity modelling diagrams. *Data-flow* diagrams (DFDs) [DeM79] model the transition of data between activities. *Activity-decomposition* diagrams such as structure charts [Ken88] model the hierarchical breakdown of activities in more and more detail. *Activity-dependency* diagrams model logical ordering of activities based on the requirements of each activity to proceed.

State diagrams convey temporal information about a system. Simple state-transition diagrams only describe states and events. State charts [Har87] are normally used to model systems with complex temporal constraints. We reviewed OMTs state diagrams as an example of state charts.

Program logic diagrams model flow of control and the conceptual structure of algorithms at a level closely related to implementation. Many varieties of program logic diagrams exist, all of which represent basically the same information with different presentation styles. Based on presentation style, three classes can be identified: those containing only lines, those containing only boxes, and those containing both boxes and lines [Tri88]. When considering line only diagrams, we reviewed Structure Program Diagrams (SPDs) [Aoy89]. When considering box only diagrams, we reviewed Nassi-Shneiderman diagrams [Nas73]. When considering box and line diagrams, we reviewed flow charts.

3.2.2 Method

Our example applies the review criteria to data-flow diagrams.

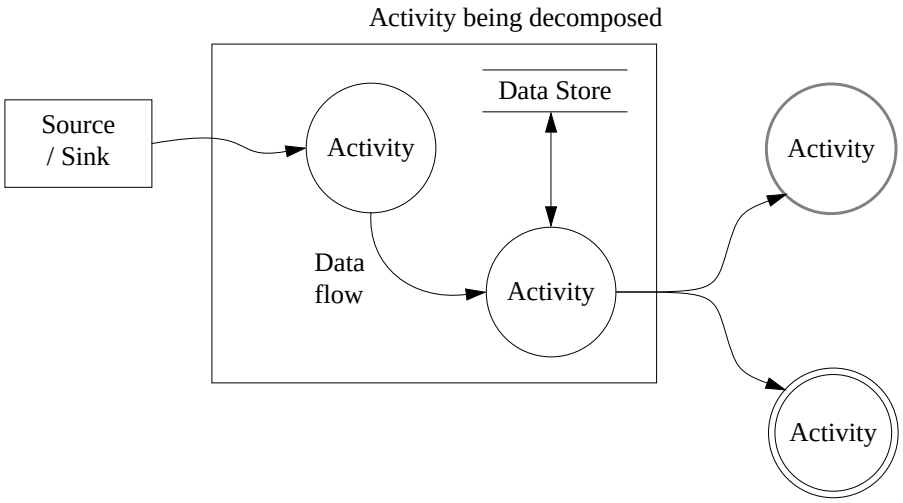

Figure 1. Components of DFDs

Data-flow diagrams are unconnected compound graphs. Each decomposition is a directed multigraph that may or may not be planar and/or cyclic depending on the design created by the user. The collection of decompositions form a rooted tree.

A single data-flow diagram is a decomposition of some activity. All nodes that participate in the decomposition are clustered together within a box which is annotated in the top-right corner to indicate the activity that is being decomposed.

Data-flow diagrams contain three primary types of node: sources and sinks which are rectangles, activities which are circles, and data stores which are two parallel lines. There are two special activity nodes, one is hashed, representing an activity outside the current decomposition and one is double lined, indicating an activity outside the scope of the analysis. Labels in nodes are centred. To conform to our definitions of graph types, it is necessary to define a fourth type of node which is placed at each split and join on a data-flow. Such a node has no visual representation.

The direction of a data-flow is given by an arrow on the edge. Data-flows may be bidirectional in which case only one arc is drawn with two arrows, one for each direction. In all cases, arrows are placed on data-flows at the point where they connect to the node to which the flow is directed.

The rooted tree that forms the collection of decompositions is not usually represented, however if it was, it would provide an ideal *interactive view* for navigation between decompositions, or simply act as a *view-only* reference map.

3.2.3 Results

A summary of applying the graph theoretic class criteria to modelling diagrams is in *table 1*. For each diagram reviewed, the table highlights the graph type(s) found and describes their attributes indicating whether they are: directed, cyclic, connected, and/or planar. For compound graphs, a description of which part forms the sub-graph is given.

Due to the complex notations used in most modelling diagrams, the resulting graph types are quite complex. The results of this survey indicate that several of the diagrams are compound graphs consisting of sub-graphs which are rooted trees. Some of the diagrams are directed graphs and some are undirected graphs, and all except for DFDs are connected graphs. Diagrams whose graphs are not rooted trees have graphs for which planarity can not be guaranteed. Except for state transition diagrams, all diagrams whose graphs are not rooted trees, are possibly cyclic. In addition to the graph types defined in *section 2.1*, it was found that software development environments need to support the presentation of other representations such as tables.

Table 1: Graph theoretic classes for modelling diagrams

	Graph Type	Compound graph description	Directed	Must be Acyclic	Must be Connected	Must be Planar
ER / EER diagrams	Simple graph		No	No	Yes	No
	Rooted tree	Generalization and categorisation hierarchies	Yes	Yes	Yes	Yes
Jackson data diagrams	Rooted tree	N/A	Yes	Yes	Yes	Yes
OMT object diagrams	Multigraph		No	No	Yes	No
	Rooted tree	Generalization and aggregation hierarchies	Yes	Yes	Yes	Yes
Data-flow diagrams	Multigraph		Yes	No	No	No
	Rooted tree	Decomposition sequences	Yes	Yes	Yes	Yes
Structure charts	Simple graph	N/A	Yes	Yes	Yes	No
SPDs	Rooted tree	N/A	Yes	Yes	Yes	Yes
Nassi-Shneiderman diagram	Rooted tree	N/A	Yes	Yes	Yes	Yes
Flow charts	Simple graph	N/A	Yes	No	Yes	No
State-transition diagrams	Multigraph	N/A	Yes	No	Yes	No
State tables	Table	N/A				
State charts	Multigraph		Yes	No	Yes	No
	Rooted tree	Generalization hierarchies	Yes	Yes	Yes	Yes

A summary of applying the conformance criteria to modelling diagrams is in *table 2*. For each appropriate diagram reviewed, the necessity for absolute positioning, relative positioning and clustering is given. Except for ER diagrams and state transition diagrams, all diagrams require support for one or more of the conformance constraints. For ER diagrams, it was found that relative positioning and clustering is desirable to convey the intentions of the author. No conformance constraints was found to be necessary for state transition diagrams. There is no consistent correlation between graph types and the necessity of the three constraints.

Table 2: Conformance criteria for modelling diagrams

	Absolute positioning	Relative positioning	Clustering
ER / EER diagrams	-	Desirable	Desirable
Jackson data diagrams	Required	Required	-
OMT object diagrams	-	Required	Desirable
Data-flow diagrams	-	-	Required
Structure charts	Required	Required	-
Structured program diagrams	Required	Required	-
Nassi-Shneiderman diagrams	-	Required	-
Flow charts	Required	Required	-
State-transition diagrams	-	-	-
State charts	-	-	Required

Modelling diagrams were widely used before there were tools to automate their development. As a result, use of appearance attributes such as colour, texture and line width was not feasible. Thus geometric attributes dominated as the mechanism for distinguishing concepts. As automated tools became available, diagrams started to incorporate simple appearance attributes such as line width, however, technology limited the application of more visually complex attributes such as colour and texture. The results of the survey are consistent with this. It was found that there is limited use of appearance attributes, which is compensated for by use of a large variety of geometric attributes.

Modelling diagrams are user-generated, and are often informal in nature. As a result, they have limited semantic connection to other documents in the system. Thus, constraints on manipulation of such diagrams are only dependent on the semantics and interrelationships of a single type of diagram. However, if a uniform paradigm is used during modelling, consistency is an issue. For example, activities identified in the activity modelling phase should be included in the data model if an object-oriented approach is chosen. Equally, activities referred to in a state-chart should exist in the set of activity models. Such consistency issues may restrict the manipulation of a document [Wel94] depending on the consistency strategy chosen. Navigation between diagrams is dependent on relationships between diagrams, which

are defined by the development paradigm being used. Navigation within diagrams is enhanced by mechanisms that provide opportunities to navigate through various sub-diagrams, such as decomposition levels in data-flow diagrams.

4 Program Visualisation

Program visualisation is the process of visually enhancing some aspects of either a program's source code or its execution. Baeker [Bae86] defines program visualisation as "*the use of the technology of interactive computer graphics and the crafts of graphic design, typography, animation and cinematography to enhance the presentation and understanding of computer programs*". This definition provides that such presentations may be static or animated [Bro88]. However, this review only considers static representations. As with modelling diagrams, a large variety of program visualisations may be used. To select an appropriate subset, we consider a classification of program visualisations.

4.1 Classification of Program Visualisation Views

Program visualisation is a relatively young field which attracted significant interest in the late 1980's. No widely accepted visual conventions exist. Existing systems that provide visually enhanced representations of programs all differ in the visualisations they provide, the notations used in the visualisations and the emphasis placed on the visualisation within the overall system.

Several taxonomies of program visualisation have been proposed by various authors, but all are associated with papers that address issues that are not considered in this paper. An early review of visual programming techniques by Raeder [Rae85] describes four classifications for diagrams representing programs:

- control-flow;
- data-flow;
- data-structure; and
- program structure.

He further notes the need for visually representing debugging information, performance information and information about entity types and the importance of preserving the abstraction mechanisms naturally found in programming languages.

Myers [Mye86, Mye90] and Roman and Cox [Rom92, Rom93] provide two variations of a taxonomy for the purpose of reviewing program visualisation systems. Myers' taxonomy is the simplest. He describes two criteria: aspect and display style. The aspect criterion indicates which aspect of a program a visualisation illustrates. An aspect may be one of the following:

- code;
- data; or
- algorithm.

Each of these aspects is then described by a display style which is either static or dynamic. Roman and Cox provide a more complex taxonomy that considers five criteria: scope; abstraction; specification method; interface; and presentation. Scope is similar to Myers' aspect criterion; it considers which aspect of a program is to be visualised. Scope may be a program's:

- code;
- data-state;
- control-state; or
- behaviour.

The abstraction criterion considers the level of abstraction of the information contained in a visualisation. Roman and Cox define three levels of abstraction as follows:

- direct representation;
- structural representation; and
- synthesised representation.

The specification, interface and presentation criteria are directed towards classification of systems, not diagrams so we do not describe them. It should be noted that this taxonomy is generally biased towards systems that provide dynamic views.

Shu [Shu85, Shu88] provides a categorisation of visual programming which includes visual environments that provide program views. This categorisation is not relevant to this paper as it is too broad in scope, but it is worth noting that he makes a distinction between a program's source code and its execution.

None of the taxonomies described above provide a classification suitable for the discussion of the presentation of program visualisations within the framework of the review criteria described in *section 2*. As previously mentioned, in this review we are only interested in static representations of a program and/or its execution. Within this restriction we define a classification scheme which is used in *section 4.2*. The primary classification of program visualisations is based on the source of the visualisation. That is, a visualisation is either derived from a program's:

- source code; or
- its execution

Visualisations derived from a program's source code are referred to as program abstraction and browsing visualisations. Visualisations derived from the execution of a program are referred to as program execution visualisations.

4.2 Analysis

This section describes the extent of the review of program visualisations, and summarises the results of the review.

4.2.1 Extent of the survey

Program abstraction and browsing visualisations allow the programmer to peruse a program in a manner that is closely related to the way a human conceptualises a program. They allow a programmer to visualise complex relations within a large system, enhancing both a programmer's comprehension of a program and increasing a programmer's efficiency during difficult and time-consuming tasks such as program maintenance. Four classes of program abstraction and browsing visualisations were identified: source code visualisations, program dependency visualisations, program logic and control flow visualisations, and static metric visualisations. Source code visualisations are inherently textual and as a result were not reviewed.

Program dependencies are relationships between a set of program elements that are dependent either syntactically or semantically on another set of program elements. The review of program dependencies was limited to those found in conventional imperative programming languages. When considering such languages, we identify four classes of program dependencies, where each class contains elements of the previously listed classes. The classes are:

- data-type dependencies;
- data-item dependencies;
- procedure/function dependencies; and
- module dependencies.

Only the first, third and fourth of these classes are associated with interesting and desirable visualisations. The second class, data-item dependencies, form complex graphs which are useful for establishing program slices [Wei81] and performing ripple effect analysis [Yau84].

Program logic diagrams were reviewed as a mechanism of modelling algorithms in terms of their control flow. This notion is equally applicable to presenting the final program. We did not review the diagrammatic requirements again as they do not differ from the previous description.

Static metrics provide useful measures of software quality and evolution. Visual representations can be used to contrast changes in functions, modules and the system as a whole, over time. Such visual representations are examples of statistical graphs, such as Kiviat diagrams, bar graphs, pie charts, and line graphs.

Program execution visualisations allow the programmer to visualise information concerning the run-time behaviour and performance of a program. Among other things, this type of visualisation can be used as an aid in: locating and resolving run-time errors; optimising program source code and assessing test coverage.

We reviewed diagrams from four types of program execution visualisations: data-content visualisations, execution performance visualisations, coverage visualisations and execution stack visualisations. These are not the only diagrams a programmer may desire, but they illustrate how information generated by debugging and performance evaluation tools can be

presented, and are representative in scope and nature of program execution visualisations in general.

4.2.2 Results

A summary of applying the graph theoretic class to program visualisations is in *table 3*. The information is presented in the same way that it was for modelling diagrams. Not all views form graph types defined in *section 2.1*. Static metric visualisations form more traditional presentations which should also be supported. For data content visualisations, it is not possible to determine what type of graph will be produced until run-time. None of the visualisations reviewed are compound graphs. For the appropriate visualisation types, all graphs are directed, and possibly cyclic. Some are possibly unconnected and those which are not rooted trees are possibly non-planar. It should be noted that as program visualisation becomes more widely used, new notations will evolve which may constitute more complex graph types.

Table 3: Graph theoretic classes for program visualisation views

	Graph Type	Compound graph description	Directed	Must be Acyclic	Must be Connected	Must be Planar
Source code views	No graphical view					
Data type dependencies	Rooted tree	N/A	Yes	No	Yes	Yes
Data item dependencies	Same as those for base view on which the additional information is included					
Procedure / function dependencies - Call graph	Multigraph	N/A	Yes	No	No	No
Module dependencies	Multigraph	N/A	Yes	No	No	No
Program logic and Control flow views	Same as either flow charts, Nassi-Shneiderman diagrams or Structured program diagrams depending on the representation chosen					
Static metric views	Bar graphs, Pie charts, line graphs etc.	N/A				
Data contents views	All	N/A	Yes	No	No	No
Execution performance views	Same as those for base view on which the additional information is included					
Coverage views						
Execution stack views						
Abstract syntax tree	Rooted tree	N/A	Yes	No	Yes	Yes

A summary of applying the conformance criteria to program visualisations is in *table 4*. The information in this table is presented in the same way that it was for modelling diagrams. Conformance constraints were found to be very important in the presentation of program

visualisations. All of the applicable visualisations required at least one constraint, and for all it is, at a minimum, desirable to exercise at least two constraints. Again there is no consistent correlation between graph type and conformance criteria.

Table 4: Conformance criteria for program visualisation views

	Absolute positioning	Relative positioning	Clustering
Data type dependencies	Required	Required	-
Procedure / function dependencies - Call graph	Required	Desirable	Desirable
Module dependencies	-	Desirable	Required
Data contents views	Dependent on graph type		
Abstract syntax tree	Required	Required	-

As previously mentioned, program visualisation is a relatively young field. As such, it has been able to benefit from recent technological advances in hardware and software. This has resulted in the use of a large range of appearance attributes such as colour, texture and line width. It was found that although geometric attributes played an important role in differentiating information, appearance attributes are the dominant mechanism. As technology progresses, we expect that this trend will continue.

Program visualisations are generated from concrete information, that is a program's source code or its execution. As a result, manipulation of elements in a visualization have strong semantic and syntactic consequences for the underlying source code. For example, if a node is added to a call graph, an appropriate declaration of the function/procedure can be made in the source code. However, if an edge is added to reflect a call to one procedure/function from another, there is a problem reflecting this change in the source code. The problem is that there is no predetermined position in the calling function/procedure to add the code for the call. Finally, the relationships within and between program visualisations, provide excellent means for navigation within and between these visualisations and also to the source code.

5 Further Work

Software documents are generally inter-related in some way. For example, parts of specification, analysis and design documents may be related to parts of the program source code or to parts of each other. Examples of such relationships are described by Welsh and Han in [Wel94]. The number and complexity of such relationships will generally be dependent on the software development paradigm that is employed. Further work is required to establish the relationships that should be represented, and define how they are best presented diagrammatically. Finally, these diagrams should be reviewed, allowing their requirements to be incorporated into a generic software development environment.

6 Conclusions

In this paper we have described the results of reviewing a representative set of diagrams that may be used during software development. This review aimed at providing the requirements for presentation of such diagrams in a generic software development environment. A taxonomic classification was presented which was used to establish the set of review diagrams.

The survey found a diverse range of graph types in the diagrams reviewed as well as other visual presentation styles, and extensive use of conformance constraints. This indicates that a generic software development environment must provide a wide range of layout facilities, for both diagrams, tables and other appropriate visual representations. In general, modelling diagrams relied on geometric attributes to communicate information, while program visualisations had the potential to utilize both geometric and presentation attributes. As a result, a wide range in presentation attributes was found, indicating that generic environments must be flexible in the definition and use of such attributes. Interaction with diagrams should incorporate a range of navigational facilities, both within and between diagrams. Finally, it was shown that manipulation of most diagrams is constrained by either the semantics of the diagram, or its relationships to other documents. To support this, generic software development environments must provide flexible mechanisms for defining allowable interactions for each type of visual representation that it is capable of presenting.

7 Acknowledgements

This work was conducted in association with the Graphical Presentation and Manipulation project, which is supported by a research grant from the Australian Research Council. The principal researchers are Prof. Jim Welsh and Dr. David Carrington both at The University of Queensland, and Prof. Peter Eades at The University of Newcastle.

References

[Amb89] Ambler A. L., Burnett M. M. Influence of visual technology on the evolution of language environments. *IEEE Computer*, 22(10):9–22, October 1989.

[Aoy89] Aoyama M., Miyamoto K., Murakami N., Nagano H., Oki Y. Design specification in Japan: Tree-structured charts. *IEEE Software*, pages 31–7, March 1989.

[Bae86] Baecker R. M. An applications overview of program visualization. *Computer Graphics*, 20(4):325, August 1986.

[Ber93] Bergin T. J. *Computer-aided software engineering: Issues and trends for the 1990s and beyond*. Idea Group Publishing, Harrisburg, Paris, 1993.

[Bro88] Brown M. H. Perspectives on algorithm animation. In *Proceedings, CHI '88:Human Factors in Computing Systems*, pages 33–8. ACM Press, 1988.

[Che76] Chen P. The entity relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, pages 9–36, March 1976.

[DeM79] DeMarco T. *Structured Analysis and System Specification*. Prentice-Hall, Englewood Cliffs, N.J., 1979.

[Elm89] Elmasri R., Navath S.B. *Fundamentals of database systems*. Addison-Wesley world student series. Benjamin/Cummings Publishing Company, Redwood City, California, 1989.

[Fug93] Fuggetta A. A classification of case technology. *IEEE Computer*, 26(12):25–38, December 1993.

[Har87] Harel D. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8(3):231–74, 1987.

[Jac83] Jackson M. A. *System Development*. Prentice Hall, Inc., Englewood Cliffs, 1983.

[Jon95] Jones T. Diagrammatic presentation of software documents. Technical Report SVRC TR95-6, The University of Queensland, St. Lucia, Queensland 4072, February 1995.

[Ken88] Kendall K. E., Kendall J. E. *Systems analysis and design*. Prentice-Hall International, Englewood Cliffs, N.J., 1988.

[Mar85] Martin J., McClure C. *Diagramming techniques for analysts and programmers*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[Mye86] Myers B. A. Visual programming, programming by example, and program visualization: A taxonomy. In *Conference Proceedings, CHI '86: Human Factors in Computing Systems*, pages 59–66. ACM Press, 1986.

[Mye90] Myers B. A. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.

[Nas73] Nassi I., Shneiderman B. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8), August 1973.

[Rae85] Raeder G. A survey of current graphical programming techniques. *Computer*, 18(8):11–25, August 1985.

[Roc89] Rock-Evans R., Engelien B. *Analysis techniques for CASE: A detailed evaluation*, volume 1. Ovum, London, England, 1989.

[Rom92] Roman G., Cox K. C. Program visualization: The art of mapping programs to pictures. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 412–20, Melbourne, Australia, May 1992. ACM Press.

[Rom93] Roman G., Cox K. C. A taxonomy of program visualization systems. *IEEE Computer*, 26(12):11–24, December 1993.

[Rum91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W. *Object-oriented modelling and design*. Prentice Hall, Englewood Cliffs, N.J., 1991.

[Shu85] Shu N. C. Visual programming languages: A perspective and a dimensional analysis. In *International Symposium on New Directions in Computing*, pages 326–334, Trondheim, Norway, August 1985. IEEE Computer Society.

[Shu88] Shu N. C. *Visual Programming*. Van Nostrand Reinhold, New York, New York, 1988.

[Tri88] Tripp L. L. A survey of graphical notations for program design–An update. *ACM SIGSOFT Software Engineering Notes*, 13(4):39–44, 1988.

[Wei81] Weiser M. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–49, San Diega, California, March 1981. IEEE Computer Society.

[Wel94] Welsh J., Han J. Software documents: Concepts and tools. *Software–Concepts and Tools*, 15(1):12–25, January 1994.

[Yau84] Yau S. S. Methodology for software maintenance. Technical Report RADT-TR-83-262, Rome Air Development Centre, Griffis Air Force Base N. Y., February 1984.