VISUALISATIONS OF LARGE OBJECT-ORIENTED SYSTEMS

P. HAYNES
OO Pty. Ltd., P.O. Box 1826, Nth. Sydney, NSW, Australia;
philip@oose.com.au

T.J. MENZIES
Software Development, Monash University, Melbourne, VIC, Australia;
timm@insect.sd.monash.edu.au

R.F. COHEN
Computer Science, Newcastle University, Newcastle, NSW, Australia;
rfc@cs.newcastle.edu.au

The use of *ternary diagrams* to represent *normalised call graph directions* permit the succinct visualisations of object-oriented (OO) systems. Important features of such diagrams include (i) the ability to compare different object-oriented applications; and (ii) the potential ability to make value judgments about partially completed systems. Ternary diagrams also permit an overview of very large graphs. For example, we present here a visualisation of five OO applications comprising 1,643 vertices and 194,451 edges.

1 Introduction

A *call graph* is a directed graph whose vertices represent basic data values and whose edges represent how those basic data values are passed to subroutines. An *anonymous call graph* is a call graph where all the vertices have been changed to anonymous variables (e.g. *class0023*) and the source of the call graph is not recorded with the graph. Call graphs offer a uniform view for a variety of programming systems. For example, the dependency network within propositional expert systems, the patterns of functions calls in a procedural program, or OO systems can all be viewed as call graphs. If we can develop methods for making value judgments about software systems based on their call graphs, then we will have a widely applicable software engineering technique.

Many systems, particularly those which present relational information, include an information visualization function. Examples include CASE tools [13], idea organizing systems [18], reverse engineering systems [19] and software design systems [6]. Such systems have motivated a great deal of research on algorithms and systems for visualizing relational information (see the survey [3] for over 250 references). In this paper, we focus on call graphs from OO systems.

The problem with visualising call graphs is that they can be very large (e.g. thousands of vertices and hundreds of thousands of edges). Classical

graph models tend to become insufficient for visualising graphs this size. We are limited either by the size of a window, or by the number of pixels available to represent an entire large graph [23]. Some more powerful graph formalisms for representing information have been introduced, e.g. hypergraphs [4], compound digraphs [22], cigraphs [14], higraphs [11], and clustered graphs [7,8]. These graph types propose various ways of summarizing or hiding information via recursive grouping of vertices. The problem of automatically drawing these types of graphs appears difficult. To date only a few algorithms have been proposed [7,8,22,20].

Call graphs (in the general case) are large digraphs, without a meaningful way to determine a recursive grouping structure. Rather than displaying all the details of such large graphs, we have sought a summary visualisation which permits an overview of the entire application. We call this summary representation *ternary diagrams*.

Section 2 describes OO call graphs and the problems we have visualising them. Section 3 describes our novel visualisation technique. Using our ternary diagrams, we can visualise a call graph with thousands of vertices and hundreds of thousands of edges. Section 4 describes our experimental results. Conclusions and directions for future work are discussed in Section 5.

2 OO Call Graphs

For OO systems, we define a call graph vertex to be a class. Each method M_j^i of a class C^i contains:

- References to instance variables I_k^i that are either instance variables defined in C^i or its parent classes.
- Calls to methods defined in some class X.

We add to our call graphs an edge for each such call and reference.

In terms of software engineering metrics research, call graphs have certain useful properties:

- Call graphs can be generated by automatic tools; i.e. they require little effort on the part of an organisation to generate them.
- We have argued previously for more experimentation in software engineering research [17]. One pragmatic restriction to such experimentation is that most software development takes place behind closed (and probably locked) doors. Organisations are reluctant to publish the details of their

work. The time taken to prepare a publication may be better spent on developing more software. Further, such publications may give competitors hints regarding how to improve their market share. Hence, we propose that organisations share anonymous call graphs. Organisations can place such anonymous call graphs in the public domain, without compromising their commercial standing.

There are two practical issues with using call graphs. Firstly, the class X of the methods that are called by method M_j^i must be computed. This is called *type inferencing*. Using parsing tools, type inferencing in strongly-typed languages (e.g. Eiffel, C++) is a relatively simple matter since the type can be inferred from the type declarations for the variables and parameters used by the methods. In un-typed languages (e.g. Smalltalk), no such type clues can be found in the source code. For such languages, heuristic type inferencing is required. While sophisticated schemes exist for detailed type inferencing [21,2], we have found that a simple "brute-force" approach suffices. Our *butterfly heuristic* [12] relies on certain Smalltalk-specific methods that return the list of method names used in a method. The butterfly algorithm:

- Collects lists of method names that are only defined in one class. Such unique names can be used to quickly determine the class type X for the method calls to the uniquely named methods.
- Parses for the special special Smalltalk variables *self* and *super* in order to detect calls back into the current hierarchy.
- Uses a hand-built library of commonly-used method names to decide where ambiguous edges points to [a].
- Assigns the class type X for any remaining unresolved calls heuristically using the distributions computed from the above techniques.

Experimentally, we cannot detect any statistical differences between such automatically generated call graphs and call graphs built manually by programmers reading the code.

The second, previously unresolved, issue with call graphs is visualising them. Such graphs can be very large. For example:

- In one off-the-shelf Smalltalk system, there were 216 classes whose messages called 18,614 other messages (i.e. at least 18,614 edges in the call graph).

[a] For example, any call to "*" is taken to be a call to the most general numeric class "Number"; all calls to conditions (e.g. "ifTrue:", "ifFalse:") are taken to be a call to the most general conditional class "Boolean".

- In the studies presented below, we are working with a call graph with 1,643 classes/vertices and 194,451 calls/edges.

The rest of this paper explores the use of ternary diagrams as a visualisation technique for call graphs.

3 Ternary Diagrams

3.1 Call Graph Edge Categories

There is a directed edge in the call graph for each method call and instance variable reference. The in-coming edges can be in one of six categories:

1. I_v (*local variable reference*): A method references an instance variable defined in its own class.

2. P_v (*parent variable reference*): A method references an instance variables defined in a superclass.

3. E_v (*external variable reference*): A method references an instance variable defined in another hierarchy via a *get method*. A get method is a simple accessor that only returns the value of an instance variable.

4. I_m (*internal method call*): A method calls a method in its own class.

5. P_m (*parental method call*): A method calls a method in one of its superclasses.

6. E_m (*external method call*): A method calls a method in another hierarchy.

Our analysis ignores out-going edges as well as a class calling a method or referencing a variable in a descendant class. Our intuition was that:

- The in-coming edges give the same information as the out-going edges;

- Calls down the hierarchy are a rare construct.

Elsewhere, we have experimentally confirmed these intuitions[15].

3.2 Problems with Finding External Variable References

Categories I_m and P_m are simple to compute within a hierarchy (a parser need only search for the instance variable name string). The situation is much more complicated, however, when we consider external variable references. If we could recognise get methods, then we could distinguish external method calls into (i) "real" method calls and (ii) "get" calls. However, it is hard to automatically recognise a get method. A common construct is to extend a get method such that if a variable is not found, it is initialised. Is this still a get method? Or is it a "real" method call?

Because of this problem we reduce the number of call graph edge categories to three, and consider the following values:

1. $I = |I_v \cup I_m|$ - The number of *Internal* method calls and instance variable references.
2. $P = |P_v \cup P_m|$ - The number of *Parental* method calls and instance variable references.
3. $E = |E_v \cup E_m|$ - The number of *External* method calls and instance variable references.

We combine categories this way to avoid the following potential problem: If we ignore get methods that access instance variables defined in other hierarchies, are we adding an asymmetry into our analysis(i.e. we measure references to local and parent variables, but ignore external variables)?

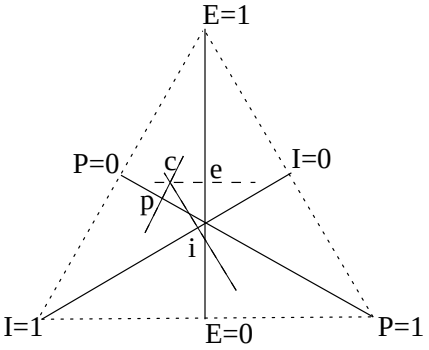
This is a complex issue which, for the moment, we will defer.

3.3 Normalised Call Patterns

We normalise I, P, and E by expressing them as ratios of the total number of calls. Once normalised: $I + P + E = 1$. We can visualise these normalised call patterns using *ternary diagrams*, a well-known technique in Chemistry. Consider Figure 1. As we move from (e.g.) $P = 1$ to $P = 0$, we are moving from the point where *all* the call graph directions are to parent classes (and $E = I = 0$) towards the point where *none* of the call directions are to parent classes. A perpendicular to this line at x is the space of all numbers with $P = x$. To locate the triple $< i, e, p >$ on the ternary diagram, we find the intersection cof the perpendiculars to the points $I = i$, $E = e$, $P = p$.

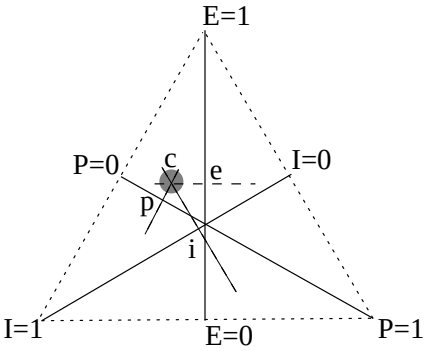
Suppose an application $\mathcal{APP}_x$ contains N classes. For each class $C^w \in \mathcal{APP}_x$, we can compute $< i_w, e_w, p_w >$. Once this is known, we can compute the mean $< i', e', p' >$, standard deviation $< i_\mu, e_\mu, p_\mu >$ and standard error

Figure 1: Locating the point $< i, e, p >$ in a ternary diagram



of the mean $< i_\sigma, e_\sigma, p_\sigma >$ [b] for $\mathcal{APP}_x$. We call $< i', e', p' >$ the centroid c_x for $\mathcal{APP}_x$. Let $\max_\sigma$ be the maximum of$i_\sigma, e_\sigma, p_\sigma$. We can then visualise the N classes of $\mathcal{APP}_x$ as follows. We draw the centroid $c_x =< i', e', p' >$ on a ternary diagram as a circle whose radius is $\max_\sigma$ and whose centre is c_x. For example, in Figure 2 we see that $\max_\sigma = \frac{1}{20} = 0.05$.

Figure 2: Displaying a centroid plus its associated standard error



When viewing ternary diagrams, it is useful to add *tenths lines* to assist in reading off the position of a point. For example, in Figure 3, we see a centroid at $< 0.4, 0.45, 0.15 >$ with $\max_\sigma$ of 0.05.

We can contrast numerous applications by drawing on the same diagram the centroids and $\max_\sigma$. In Figure 4, we see three applications, labeled $\mathcal{APP}_1$,

[b] Recall that $\sigma = \frac{\mu}{\sqrt{N-1}}$.

Figure 3: $c =< 0.4, 0.45, 0.15 >$, $\max_\sigma = 0.05$

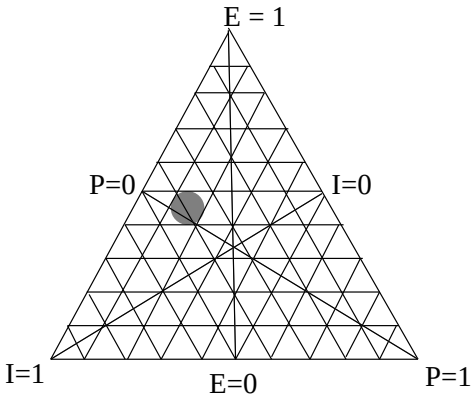
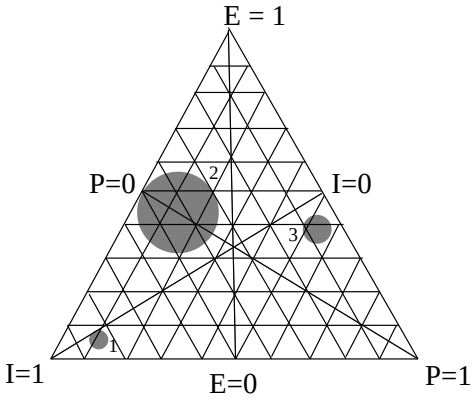


Figure 4: Comparing 3 applications



$\mathcal{APP}_2$, $\mathcal{APP}_3$. Looking at the diagram, we can say that:

- $\mathcal{APP}_1$ has the smallest $\max_\sigma$. $\mathcal{APP}_1$ appears to make little use of inherited features or services defined in other hierarchies (i.e it makes most of its calls to itself).
- $\mathcal{APP}_2$ uses equal amounts of calls to itself as calls to classes in other hierarchies. Also, $\mathcal{APP}_2$ makes little use of inheritance (compared to the internal and external calls).
- Of the three systems, $\mathcal{APP}_3$ makes the heaviest use of inheritance.

Figure 5: Mean call directions in five applications

$\mathcal{APP}$	N	i'	i_μ	i_σ	e'	e_μ	e_σ	p'	p_μ	p_σ
f:FinApp	313	0.44	0.29	0.02	0.47	0.26	0.01	0.09	0.10	0.01
e:Envy	123	0.28	0.18	0.02	0.57	0.22	0.02	0.16	0.16	0.01
v: VisualAge	527	0.31	0.18	0.01	0.49	0.19	0.01	0.20	0.13	0.01
i:IBM Smalltalk	666	0.56	0.36	0.01	0.37	0.33	0.01	0.07	0.12	0.00
m:Metric	14	0.31	0.26	0.07	0.60	0.24	0.07	0.08	0.11	0.03
a:All	1643 sum	0.38 mean	-	-	0.50 mean	-	-	0.12 mean	-	-

4 Experimental Results

The above technique was applied to five applications from the Smalltalk family:

- $\mathcal{APP}_f$: *FinApp* is an anonymous commercial applications for the financial market.
- $\mathcal{APP}_e$: *Envy* is a source code control system from Object Technology International.
- $\mathcal{APP}_v$: *VisualAge* is a general visual development tool with built-in database hooks from IBM. *VisualAge* is built on top of *IBM Smalltalk*.
- $\mathcal{APP}_i$: *IBM Smalltalk* is a interactive Smalltalk development environment with an incremental compiler.
- $\mathcal{APP}_m$: *Metric* is the metrics package that implements our butterfly heuristic.

Note that:

- Not all the classes of these $\mathcal{APP}$s were used in the analysis. Class with a majority of methods with source code missing were excluded.
- The $\mathcal{APP}$s can be categorised into *young* and *mature* applications. *FinApp* and *Metric* have only just been completed. *IBM Smalltalk*, *VisualAge* and *Envy* are much more mature products that have been extensively maintained.

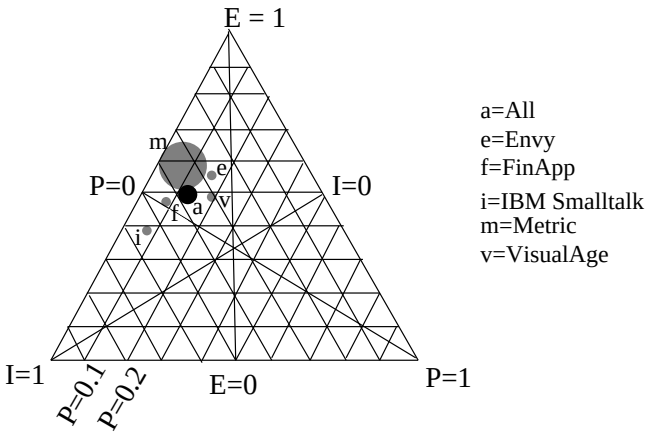
The centroids for each $\mathcal{APP}$ are shown numerically in Figure 5. Note that the μ figures in Figure 5 are very large. If we want to claim that some newly

analysed class is like one of the above $\mathcal{APP}$s, we would have to check this using $x' \pm 2\mu$ (to be 95% certain) and $x' \pm 3\mu$ (to be 99% certain). For the above data, such a test would be very relaxed (e.g. for *FinApp*, $i' = 0.44$, $i_\mu = 0.29$ and $0 \leq (i' \pm 2\mu) \leq 1$). This is a disappointing result since we had hoped to be able to use this technique interactively during the software development process to find classes that were unusually designed.

Nevertheless, the results of Figure 5 do allow us to conclude statistically that the call patterns of the studied applications were different. Applying a two-tailed t-test, we explored the null hypothesis that $\mathcal{APP}_\alpha.x' = \mathcal{APP}_\beta.x'$ for $\alpha, \beta \in \{f, e, v, i, m\}$ and $x' \in \{i, e, p\}$. If we could reject the null hypothesis that *any* of i', e', p' was the same for two applications, then we rejected the hypothesis that the centroid of the two applications was the same. In all cases, we could reject the null hypothesis at the 0.05 level of significance.

The ternary diagram for the data in Figure 5 is shown in Figure 6. The centroids, plus or minus $\max_\sigma$ are all distinct; i.e. our visual reading of Figure 6 is that the studied applications are different. Happily, this visual reading concurs with the above statistical analysis.

Figure 6: Centroids from Figure 5



We also studied the effects of including/ignoring the instance variable references I_v and P_v. The data from Figure 5 was collected ignoring the instance variable references. Figure 7 shows the effect on the centroids when we included I_v into i' and P_v into p'. In Figure 7, the columns marked *?* show the results of a two-tailed t-test of the null hypothesis that the mean values are the same

Figure 7: Measuring centroids plus instance references

$\mathcal{APP}$	i'	i_μ	?	e'	e_μ	?	p'	p_μ	?
FinApp	0.48	0.28		0.42	0.25		0.093	0.11	$\surd$
Envy	0.33	0.19		0.52	0.22	$\surd$	0.16	0.16	$\surd$
VisualAge	0.36	0.18		0.44	0.18		0.21	0.13	$\surd$
IBMSmalltalk	0.58	0.32	$\surd$	0.33	0.29		0.094	0.11	
Metric	0.38	0.24	$\surd$	0.54	0.24	$\surd$	0.08	0.092	$\surd$

with and without instance variable references[c] Values where the null hypothesis were accepted at the 0.05 significance level are marked with $\surd$. Note that most values are not marked; i.e. the usual case is that the means are statistically different.

While the statistics say that we cannot ignore the instance variable references, pragmatically we note that the i', e', p' values for Figures 5 & 7 are nearly the same (differing only in the second decimal place). Hence, we elect to ignore instance variable references. The alternative is to develop a parser that recognises "get" methods in external classes. Our belief is that such a parser could only ever be approximately correct.

5 Discussion

5.1 Software Engineering Implications

We choose to explore call graph visualisations in order to assist us in our software engineering research. This section discusses what software engineering results we can extract from Figure 6.

One striking feature of Figure 6 is the relatively low use of inheritance in the studied systems. The mean p' was 0.12 (see the *All* row of Figure 5) while the maximum p' was only 0.20. Note also that, with the exception of *IBM Smalltalk*[d], the mature applications *Envy* and *VisualAge* had a higher p' than the younger applications (i.e. *Metric* and *FinApp*. We speculate that as developers get more experienced with OO, they use more inheritance. If this is so, then we could evolve a "maturity measure" for OO applications based on p'. That is, the larger the p' figure, perhaps the more mature the classes are.

We have argued elsewhere[16] that the primary advantage of OO languages is their support for the many-to-one join across the $\mathcal{ISA}$ link. While OO

[c] i.e. by comparing the means of Figure 7 with the means of Figure 5

[d] IBM Smalltalk makes many calls to operating system functions. Hence, we suspect that the low p' value is a result of its functional nature.

languages provide extensive automatic support for the $\mathcal{ISA}$ link [e], they are usually record-at-a-time languages. Record-at-a-time languages are clumsy for applications were the usual join is many-to-many. Unlike the $\mathcal{ISA}$ link, external calls (measured by e') can be joins across links of arbitrary multiplicity. Codd has argued [5] that set-at-a-time languages (e.g. SQL, Prolog) are required to support joins across links of arbitrary multiplicity. Since the average call in the systems studied here is not inheritance, we suspect that that set-at-a-time processing needs to be added to the OO paradigm.

5.2 Are Our Measurements Valid?

One explanation for the relatively low values of p' is that Smalltalk is a single parent inheritance language. In a single parent OO language, if a class requires the services of more than one other class, it must reference an instance of the other class via an E call. We hypothesis that p' will be larger in multiple inheritance languages (e.g. Eiffel and C++) and are working on collecting the relevant statistics from applications built in those languages.

Another drawback with our analysis is that it ignores the runtime behaviour of an OO system. Call graphs are static. They describe the possible paths that could be taken by a running OO system. If a running system only ever uses some subset of these paths, then we should base our information hiding/ inheritance usage conclusions on those portions of the program actually exercised at runtime. We are currently exploring metering the runtime of Smalltalk systems (i.e. in the manner of [1,10]) order to confirm/refute the above static analysis.

6 Conclusion

One success criteria for any visualisation technique is this: does the technique allow us to find out something that was not known previously before looking at the diagram? Our ternary diagrams permit a better understanding of the relative importance of different call directions. For example, the low p' values for the studied applications was counter to our pre-experimental intuitions. Further, using the ternary diagrams, we are able to make novel conclusions about OO systems. Suppose we repeated the above analysis for systems developed using other languages. Three exciting results from such a study could be:

- The call graph pattern is the same in different languages. This would suggest that language choice in OO was not a major issue since, once the

[e] e.g. automatic copying of attributes from parent to child classes, automatic join up the generalisation link

applications are built, they all look the same.

- In all OO applications, p' remains very low. This would support the above argument that OO languages need to be extended beyond just $\mathcal{ISA}$ support.

- If we found that p' was the same in single and multiple inheritance systems, then this would assist in ending the single vs multiple inheritance debate.

Ternary visualisations of anonymous call graphs are a practical tool for large-scale data collection. Our tools take 5 hours to run on a Pentium 90 with 32MB of RAM. We can reasonably ask commercial organisations to run our tools overnight, and return to us the collected anonymous call graphs. Once the call graphs are pooled, then the contributing organisations can benefit from the experience contained in that pool. Our current goal is the generation of a database of ternary diagrams for different applications categorised by:

- The OO language used to develop it.

- The years of experience of the developers.

- The *domain* of the project; i.e. is the development intended for a specific application, or part of some re-use library?

- The change rate. Certain source code management systems allow an historical overview of how parts of an application changed over time. If we can link certain patterns of method calls to error rates, then we can use the ternary diagrams as a quality management tool.

The major negative result of this study was the large μ seen in i', e' and p'. Due to this large μ, we seem unable assess the design a single class by comparing it with applications mapped onto a ternary diagram. This area deserves further work. Perhaps after computing the centroids for many applications, we will be able to make value judgments about classes. In the meantime, we can use ternary diagrams to make general statements about the macro nature of OO applications.

One final note: the standard response to our plea for more empirical studies in software engineering is that "it's too hard". We hope that we have demonstrated here that this is not a defensible position. After an initial period of tool development, significant portions of the software process can be accurately quantified. We feel that such quantitative studies are important since very little of current software practice is being experimentally verified [9,17].

References

1. O. Agesen and U. Holzle. Type Feedback vs Concrete Type Inference: A Comparison of Optimisation Techniques for OO Languages. In *OOPSLA '95*, pages 91–107, 1995.
2. O. Agesen, J. Palsberg, and M. Schwartzbach. Analysis of Objects with Dynamic and Multiple Inheritance. In *ECOOP'93, Seventh European Conference on Object-Oriented Programming*, pages 329–349. Springer-Verlag, 1993.
3. G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. Algorithms for Drawing Graphs: An Annotated Bibliography. Technical report, Department of Computer Science, Brown University, 1993. To appear in **Computational Geometry and Applications**, currently available from wilma.cs.brown.edu by ftp.
4. Claude Berge. *Hypergraphs*. North-Holland, 1989.
5. E.F. Codd. A Relational Model of Data for Large Shared Data Banks. **Communications of the ACM**, 13:377–387, 1970.
6. M. Consens, A. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *14th International Conference on Software Engineering (Melbourne)*, pages 11 – 15, 1992.
7. Q.W. Feng, R.F. Cohen, and P. Eades. How to Draw a Planar Clustered Graph. In *Proc. of the First International Conference on Computing and Combinatorics (COCOON95)*, Lecture Notes in Computer Science. Springer-Verlag, 1995.
8. Q.W. Feng, R.F. Cohen, and P. Eades. Planarity for Clustered Graphs. In *Proc. of the Third European Symposium on Algorithms*, Lecture Notes in Computer Science. Springer-Verlag, 1995.
9. N. Fenton, S.L. Pfleeger, and R.L. Glass. Science and Substance: A Challenge to Software Engineers. **IEEE Software**, pages 86–95, July 1994.
10. D. Grove, J. Dean, C. Garnett, and C. Chambers. Profile Guided Reciever Class Prediction. In *OOPSLA '95*, pages 108–123, 1995.
11. D. Harel. On visual formalisms. **Communications of the ACM**, 31(5):514–530, 1988.
12. P. Haynes and T.J. Menzies. The Effects of Class Coupling on Class Size in Smalltalk Systems. In *Tools '94*, pages 121–129. Prentice Hall, 1994.
13. Silicon Graphics Inc. **CASEVision/workshop user's guide**. Silicon Graphics Inc, 1992. Volumes I and II.
14. W. Lai. *Building Interactive Diagram Applications*. PhD thesis, Department of Computer Science, University of Newcatle, Callaghan, New South Wales, Australia, 2308, June 1993.
15. T. Menzies and P. Haynes. Empirical Observations of Class-level Encapsulation and Inheritance. Technical report, Department of Software Development, Monash University, 1996.
16. T.J. Menzies. IS-A Object PART-OF Data Modeling? Technical Report TR95-3, Department of Software Development, Monash University, 1995.

17. T.J. Menzies and P. Haynes. The Methodologies of Methodologies; or, Evaluating Current Methodologies: Why and How. In *Tools Pacific '94*, pages 83–92. Prentice-Hall, 1994.
18. K. Misue and K. Sugiyama. An overview of diagram based idea organizer: D-ABDUCTOR. Technical Report IIAS-RR-93-3E, ISIS, Fujitsu Laboratories, 1993.
19. H.A. Muller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications.* PhD thesis, Rice University, 1986.
20. S. C. North. Drawing ranked digraphs with recursive clusters. preprint, 1993. Software Systems and Research Center, AT & T Laboratories.
21. J. Palsberg and M. Schwartzbach. Object-Oriented Type Inference. In *Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, 1991.
22. K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. **IEEE Transactions on Systems, Man and Cybernetics**, 21(4):876–892, 1991.
23. K.J. Supowit and E.M. Reingold. The Complexity of Drawing Trees Nicely. **Acta Informatica**, 18:377–392, 1983.

Note that some of the Haynes & Menzies papers can be obtained from *http://www.sd.monash.edu.au/ ~timm/pub/ docs/papersonly.html.*