Composing Subsystem Structures using (k,2)-partite Graphs[†]

Hausi A. Müller
James S. Uhl

Department of Computer Science
University of Victoria
P.O. Box 1700
Victoria, B.C., Canada, V8W 2Y2

DCS-128-IR March 1990

[†] This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant A9202.

Composing Subsystem Structures using (k,2)-partite Graphs[†]

Hausi A. Müller
James S. Uhl

Department of Computer Science
University of Victoria
Victoria, BC V8W 2Y2

Abstract

Subsystem composition is the process of constructing composite software components out of building blocks such as variables, procedures, modules, and subsystems. Hierarchical subsystem structures are formed by imposing equivalence relations on the resource-flow graphs of the source code. Composition algorithms often use a single equivalence relation (e.g., connection strength or data binding measure) to automatically form tree-shaped composite structures.

This paper describes a clustering algorithm that uses four equivalence relations for identifying subsystem structures. The resulting compositions are $(k, 2)$-partite graphs (a class of layered graphs) rather than strict tree hierarchies. The algorithm is an integral part of our interactive graph editor.

Keywords: Reverse engineering, design recovery, software maintenance, composition alternatives, exact interfaces, $(k, 2)$-partite graphs, composition models.

1. Introduction

You don't invent the answer, you reveal it.
—Jonas Salk

For the past three decades research in program transformation has mainly concentrated on improving the execution time of a program (i.e., optimizing compilers). Comparatively little research has been devoted to program transformations that benefit the people who change and maintain software. One avenue of research in this latter realm of program transformation is to optimize the structure of a given software system for ease of future changes.

Restructuring a system to make it more understandable is a promising but difficult undertaking. One common approach is to identify subsystem structures in the source code in order to recover system abstractions and refinements. Over the years, numerous batch algorithms have been proposed to generate subsystem hierarchies from module graphs, but no ideal composition measure has emerged from these investigations. The algorithms are usually based on software engineering principles that concern module interactions such as *low coupling*, *high strength*, *small interfaces*, and *few interfaces*.

Subsystem identification is repeated many times over the life span of a software project. During the design phase subsystem structures are often used to split the project into work assignments to manage the design and implementation of the project. At integration time, subsystem decompositions may serve as testing and integration plans. During the maintenance phase subsystem structures are often uncovered from the source code to verify existing documented structures and to be able to understand and limit the effects of local changes on the entire system. Thus, computing hierarchical composition structures is not only beneficial for reverse engineering and design recovery, but also for exploratory design and rapid prototyping.

Discovering and identifying subsystem structures is an art. Our work is based on the premise that an experienced software engineer will always be able to produce a ''better'' system

[†] This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant A9202.

decomposition than an automatic procedure — given sufficient time. However, the human designer needs assistance from the programming environment for the tedious and arduous tasks involved in the composition process. A designer may call upon the environment to produce alternative clusterings of a given set of modules and then decide on strategies to form compositions. As the hierarchy of subsystems is being built, the software engineer can interactively modify the layers and possibly undo some of the clusters if they are deemed inappropriate.

This paper outlines an algorithm for building multiple alternative subsystem hierarchies using four equivalence relations. Section 2 introduces the class of $(k, 2)$-partite graphs for modeling multiple hierarchies and outlines software structure models. Section 3 defines four clustering measures for building subsystem structures. The composition algorithm and its analysis are presented in Sections 4 and 5; The algorithm is illustrated with an example in Section 6. Related work is summarized in Section 7.

2. Software structure models

Software structures such as control flow, data flow, and resource flow are often modeled by directed weighted graphs. We use a special class of directed graphs called $(k, 2)$-partite graphs for presenting and representing software systems. We first give a definition of these graphs and then show how resource-flow graphs and multiple subsystem hierarchies can be expressed using $(k, 2)$-partite graphs.

2.1. The class of (k,2)-partite graphs

Definition. A directed graph $G = (V, E)$ is $(k, 2)$-partite if V can be partitioned into subsets $V_1, ..., V_n$, where for all $i \in \{1, ..., n\}$, $|V_i| \leq k$; and for all $(u, v) \in E$, there exists $i \in \{1, ..., n\}$ such that $u, v \in V_i$, or for $i < n$, $u \in V_i$ and $v \in V_{i+1}$. □

More informally, a $(k, 2)$-partite graph consists of a series of graph *levels* or *layers* as depicted in Figure 1 below. Layers are connected by means of *level edges*; however, level edges may only connect adjacent layers (i.e., adjacent sets in the sequence $V_1, ..., V_n$). The number of vertices per layer is bounded by k. By bounding the size of a layer, Mata-Montero was able to show that many intractable graph problems can be solved efficiently (i.e., in polynomial time) for $(k, 2)$-partite graphs [ElMM 90]. We have also proposed $(k, 2)$-partite graphs as the backbone for hypertext systems [Müll 89].

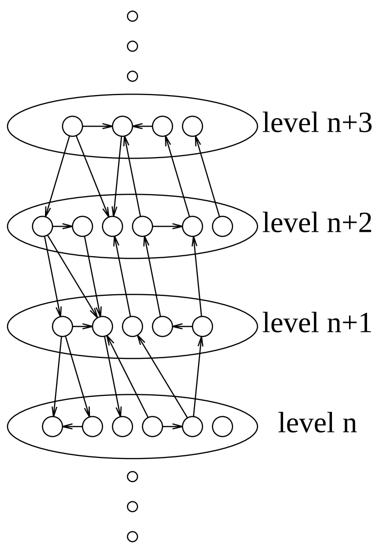

Figure 1. Directed $(k, 2)$-partite graph

2.2. Resource relations

The primary models used to describe, represent, and manage software structure are the *unit interconnection model* and the *syntactic interconnection model* [Perr 87].

It is convenient for us to think of a resource interconnection model as a directed weighted graph, where the vertices of the graph are the components of the system, and the edges are dependencies induced by the resource supplier-client relation. A directed edge from node a to b indicates that module a provides a set of syntactic objects to module b. Depending on the application, the edge weights are a list of resource names (e.g., $S = \{\alpha, \beta, \gamma, \delta\}$), the cardinality of the resource set (e.g., $|S| = 4$), or even absent. The main distinction between the unit and the syntactic models is the granularity of interconnection ranging from *files*, *subsystems*, and *modules* in the unit model to nameable entities defined in a programming language — *procedure*, *constant*, *type*, and *variable* — in the syntactic model.

2.3. Composition relations

The graphs induced by the resource supplier-client relation are ‘‘flat’’ and typically unwieldy. To add organizational axes composition relations are imposed on these graphs.

A composition relation collapses resource-dependency subgraphs to form subsystems. If the relation is constrained so that a given node can only appear in one subsystem, then the relation induces a strict tree hierarchy. In our approach this restriction is not enforced and hence the result is a layered graph or, more formally, a $(k, 2)$-partite graph. The edges between layers in such a graph represent composition dependencies whereas the edges within a layer represent resource dependencies.

Thus, a series of subsystem layers (resource-dependency graphs) are modeled by a sequence of layers $G_1 \cdots G_n$ of a $(k, 2)$-partite graph. The grain size of the nodes increases with the sequence number — from the objects of the syntactic model to the objects of the unit model.

It is useful to introduce a graph transformation which is usually called *collapse* for defining and composing subsystem structures. Collapse essentially replaces a subgraph — a set of modules — by a single node — a subsystem. Its inverse operation restores the original graph. To make this operation completely reversible, we not only have to restore the subgraph, but also the edges between the subgraph and the remaining graph.

Let $G = (V, E)$ be a resource dependency graph. For the purpose of describing the collapse operation we simply replicate this graph to form a series of graph layers $G_1, ..., G_n$. Connecting corresponding nodes of adjacent layers by means of level edges makes it a $(k, 2)$-partite graph.

Let $G_s = (V_s, E_s)$ be a subgraph of G_{i+1} (i.e., $V_s \subseteq V_{i+1}$ and $E_s \subseteq E_{i+1}$) and let E_c denote the edges between V_s and the remaining graph (i.e., the set of all edges such that one end point is in V_s and the other is in $V_{i+1} - V_s$). Let E_l be the set of level edges connecting the two corresponding node subsets of G_i and G_{i+1}. By collapsing G_s we mean the removal of G_s from G_{i+1} and replacing it with a single node $x \in V_{i+1}$. The edges E_c and the level edges E_l are detached from the subgraph nodes V_s and re-attached to the contracted node x as shown in Figure 2. The re-attached edge sets are called E_c' and E_l'. The following set equations summarize the effects of the collapse operation.

$$E_l = \{ (v, w) \in E_l \mid v \in V_s,\ w \in V_i \}$$

$$E_l' = \{ (v, w) \in E_l' \mid v = x,\ w \in V_i \}$$

$$E_c = \{ (v, w) \in E_c \mid v \in V_s,\ w \in V_{i+1} - V_s \}$$

$$E_c' = \{ (v, w) \in E_c' \mid v = x,\ w \in V_{i+1} - V_s \}$$

$$V_{i+1} = V_{i+1} - V_s \cup \{ x \}.$$

$$E_{i+1} = \{ (v, w) \in E_{i+1} \mid v, w \in V_{i+1} - V_s \} \cup \{ (v, w) \in E_{i+1} \mid v \in V_{i+1},\ w \in V_s \}$$

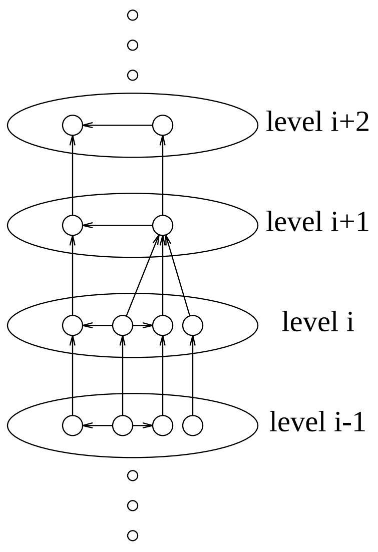

Figure 2. Collapse

2.4. Exact interfaces

Most programming languages are imprecise with respect to requisition and provision of resources [WoCW 88]. Modules import and export entire interfaces rather than specific objects.

Composition algorithms typically rely on the availability of the exact syntactic interfaces or the exact cross-references among software components.

Definition. The *exact interface* $EI(m) = (ER(m), EP(m))$ of a software component m consists of a set of *exact requisitions*, $ER(m)$, and a set of *exact provisions*, $EP(m)$ as defined by the following set equations.

$$ER(m) = \bigcup_{x \in Env(m)} (Prov(x) \cap Req(m))$$

$$EP(m) = \bigcup_{x \in Env(m)} (Prov(m) \cap Req(x))$$

$Env(m)$ — the environment of m — denotes the nodes in a layer of a $(k, 2)$-partite graph whereas m and x are single nodes in such a layer.

$ER(m)$ is defined as the intersection of $Req(m)$ — the set of objects referenced in m — and $Prov(x)$ — the object sets provided by the modules of the environment of m.

$EP(m)$ is defined as the intersection of $Prov(m)$ — the set of objects provided by m — and the object sets required by the clients of m. □

The exact interfaces of any level in a $(k, 2)$-partite graph can be computed by tediously inspecting the cross-reference listings produced by a compiler. However, Uhl has described and implemented algorithms for efficiently propagating the exact interfaces from layer to layer [Uhl 89].

Note that the exact provisions of a component are often a subset of the objects provided by the component. Consequently, subsystems can encapsulate large interfaces, providing a considerably smaller set of objects to the remainder of the system (i.e., a *small interface*).

3. Composition measures

Taxonomic hierarchies are formed automatically by computing cluster similarity measures [DuEv 82]. This section defines two pairs of measures for resource-flow graphs.

The purpose of the first pair is to capture the two software engineering principles *high strength within a component* and *low coupling among components*. The intention of the second pair is to identify loosely coupled components having *common clients* or *common suppliers*. This measure satisfies the software engineering principle *few interfaces*, because merging components with common neighbors reduces the number of interfaces among the components involved. The composition algorithm presented in the next section uses these measures for building composite structures out of routines, modules, and subsystems.

3.1. Interconnection strength measure

We define the *interconnection strength* $IS(v, w)$ of two nodes, v and w, in a resource-flow graph as the *exact* number of syntactic objects exchanged between the two nodes. Two components are said to be *strongly coupled* iff their interconnection strength is greater than a certain threshold T_{sc} and *loosely coupled* iff their interconnection strength is less than a certain threshold T_{lc}. T_{lc} and T_{sc} can be increased and decreased in a stepwise fashion to obtain alternative compositions and partitions, respectively.

Subsystems with high internal strength can be identified by clustering strongly coupled components. Subsystems with low coupling among them can be found by *separating* loosely coupled components or by using a graph partitioning algorithm for computing *articulation points*. If the removal of a vertex v disconnects a connected graph G, then v is said to be an articulation point. If G contains no articulation points then G is biconnected.

The designers of Infuse invoke the interconnection strength measure to build hierarchical experimental databases for managing source changes [MaKa 88, PeKa 87]. Their rationale is based on the premise that the probability of an interface error between two modules is proportional to the modules' interconnection strength. Choi and Scacchi compute articulation points with the objective of minimizing alteration distances [ChSc 90]

3.2. Common neighbor subset measure

We distinguish between the direct clients (immediate successors), $SUCC(x)$, and the direct suppliers (immediate predecessors), $PRED(x)$, of a node x in a resource-flow graph. Two components are similar with respect to their clients iff they provide objects to similar sets of clients. Analogously, two components are similar with respect to their suppliers iff they require objects from similar sets of suppliers. Thus, the common client and supplier subsets of a set M of components, $CS(M)$ and $SS(M)$, are defined by the following set equations.

$$CS(M) = \bigcap_{x \in M} SUCC(x)$$

$$SS(M) = \bigcap_{x \in M} PRED(x)$$

Two nodes, v and w, are said to be *common neighbors* with respect to their clients (suppliers) iff the cardinality of their client (supplier) subset $|CS(v, w)|$ ($|SS(v, w)|$) is greater than a certain threshold T_{cs} (T_{ss}).

For example, consider the graph component on the left in Figure 3; nodes a, b, and d are clients of both x and y; but c is only a client of x. Thus, the common client subset of x and y is $\{a, b, d\}$. Due to their similar clients, x and y are merged to form a subsystem — the graph component on the right in Figure 3.

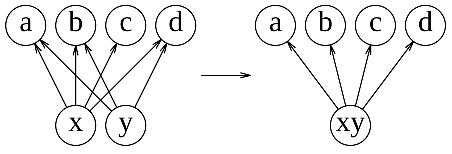

Figure 3. Merging neighbors

Often library routines which implement a related set of primitives do not depend on each other. Examples of such libraries under Unix include the the standard C library `stdlib` and the mathematics library `math`. Thus, if a routine out of one of these libraries is used by a given client c, it is likely that semantically related routines are also required by c. Schwanke and Platoff use a clustering algorithm based on this measure in their ARCH environment [ScPl 89].

4. Subsystem composition algorithm

We now have sufficient machinery to formulate the subsystem composition algorithm. The algorithm essentially consists of five major steps operating on resource-flow graphs. Each step identifies subgraphs according to some composition rule and then forms subsystems by collapsing these subgraphs. The steps were designed to be self-contained and efficient so that they can be invoked individually from our interactive graph editor which is part of the Rigi system for programming-in-the-large [MüKl 88, MHHL 89].

Subsystem composition algorithm:

Input: A directed weighted resource-flow graph, $G = (V, E)$. The weights on E are the cardinalities of the exact sets of objects exchanged among the components in V.

Output: A $(k, 2)$-partite subsystem composition graph consisting of a sequence of resource-flow graphs $G_1 \ldots G_n$.

Method: The first two steps preprocess the initial resource-flow graph and are thus only applied once. The last step cleans up the generated composition structure in a postprocessing step. Steps 3 and 4 are applied repeatedly to form layers of subsystems. The example in Section 6 shows how the different steps of the algorithm can be intermixed. Note that each step affects only two adjacent layers of a $(k, 2)$-partite graph.

1. Remove omnipresent nodes

For each node $v \in V$ in G compute the number of direct clients of v (i.e., immediate successors). If $|SUCC(v)|$ is greater than a certain threshold T_{op}, then v is said to be *omnipresent*. Omnipresent components obscure system structure and are therefore removed from G together with all their incident edges. An example of an omnipresent node is a debugging module containing debug variables or routines which are referenced by most other system components.

2. Compose by standard library

For each standard library L, identify the components of G that are members of L and collapse the identified subgraph to a subsystem.

3. Compose by interconnection strength

For each edge $(v, w) \in E$ in G compute its interconnection strength $IS(v, w)$. One of the following three conditions is then executed depending on the value of $IS(v, w)$. Note that only one layer is built with each invocation of this step and the interconnection strength relation is transitive; hence, the order in which node pairs are merged is inconsequential.

$IS(v, w) \geq T_{sc}$

v and w are collapsed into a single subsystem. If a node, say v, has already been assigned to a subsystem in a previous iteration of this step, then the collapse operation merges the other node, w, into that subsystem. If both v and w have already been assigned to subsystems in previous iterations, then the two subsystems are merged.

$IS(v, w) \leq T_{lc}$
: v and w are ''collapsed'' into two separate subsystems (i.e., each subsystem contains a single node). If a node, say v, has already been assigned to a subsystem in a previous iteration of this step, then the collapse operation has no effect.

$T_{lc} < IS(v, w) < T_{sc}$
: Node pairs in this category are neither strongly nor loosely coupled and therefore assigned to the same subsystem.

4. Compose by common neighbor
For each node pair v, w compute the common client subset $CS(v, w)$ and the common supplier subset $SS(v, w)$. If the cardinalities of $CS(v, w)$ and $SS(v, w)$ are greater than or equal to their respective thresholds T_{cs} and T_{ss} (i.e., v and w have either similar clients or similar suppliers and are therefore common neighbors), then v and w are collapsed into a subsystem. If a node, say v, has already been assigned to a subsystem in a previous iteration of this step, then the collapse operation merges the other node, w, into that subsystem. The algorithm is optimized based on the premise that resource-flow graphs usually have low density. Hence, the algorithm below finds the common neighbors of a node x by inspecting the neighborhood of x.

```
for each vertex x ∈ V do
  for each y ∈ SUCC(x) do
    for each z ∈ PRED(y) do
      CS(x, z) = SUCC(x) ∩ SUCC(z);
      if |CS(x, z)| ≥ $T_{cs}$ then
        collapse(x, z)
      end
    end
  end;
  for each y ∈ PRED(x) do
    for each z ∈ SUCC(y) do
      SS(x, z) = PRED(x) ∩ PRED(z);
      if |SS(x, z)| ≥ $T_{ss}$ then
        collapse(x, z)
      end
    end
  end
end
```

5. Clean up layers
Identify and remove the subsystems that contain only one component by merging them with their parent nodes.

5. Time complexity

Let n and e be the cardinalities of the node and edge sets of the initial resource-flow graph. The pre- and postprocessing (Steps 1, 2, and 5) can all be implemented in $O(n + e)$ time. Step 3, computing the interconnection strength for each edge in the graph, takes $O(e)$ time. Step 4 takes time $O(n^2)$ in the worst case. However, the neighborhood-search algorithm inspects only $O(n)$ pairs of nodes on average due to the sparsity of the resource-flow graphs. Thus, Step 4 requires $O(n^2)$ time in the worst case and $O(n)$ time in the expected case.

Steps 3 and 4 are invoked at most once for each layer built. In the worst case, even though Steps 3 and 4 may generate subsystem alternatives, $\lg n$ layers are typically generated. Thus, it takes at most $O((n^2 + e)\lg n)$ time in the worst case and $O((n + e)\lg n)$ time in the expected case to complete a system composition. Moreover, all steps of the subsystem composition algorithm are sufficiently efficient so that they can be routinely invoked in an interactive environment.

6. An example

This section illustrates Steps 3 and 4 of our composition algorithm.

A sample resource-flow graph is depicted in Figure 4. The edge weights indicate the number of objects exchanged among the components. Edges with no labels have a weight of one. By invoking shared neighbor composition step with a common client threshold $T_{cs} = 2$, we generate the graph in Figure 5 — a new layer in the composition graph. Another layer (Figure 6) is obtained by applying Step 3, interconnection strength reduction with a strong coupling threshold $T_{sc} = 3$. A further reduction with $T_{sc} = 1$ leads to a single node. Figure 7 shows the hierarchical subsystem structure with singleton nodes removed. The leaf nodes represent the initial components; the interior white and black nodes represent high-strength subsystems and shared-neighbor subsystems, respectively.

Figure 8 depicts a non-hierarchical subsystem structure obtained by applying Steps 3 and 4 to the same composition layer.

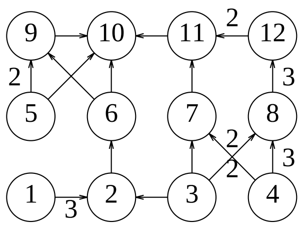

Figure 4. Initial resource-flow graph

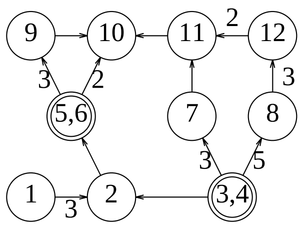

Figure 5. Composition by shared neighbors

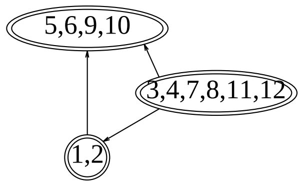

Figure 6. Composition by interconnection strength

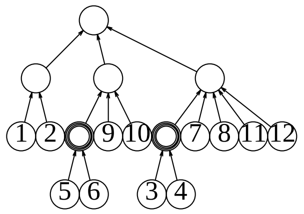

Figure 7. Hierarchical subsystem structure

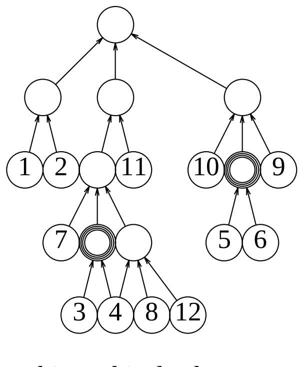

Figure 8. Non-hierarchical subsystem structure

7. Related work

Belady and Evangelisti use data bindings to form a flat module graph out of procedures [BeEv 82]. A data binding is an ordered triple (p, x, q) where p and q are procedures and x is a variable within the static scope of both p and q. Hutchens and Basili extended this approach to produce dendrograms (hierarchies of modules) [HuBa 85]. Selby and Basili also use cluster analysis based on data bindings to localize errors and to identify error-prone system structures [SeBa 88].

Kaiser, Maarek, and Perry use partitioning and clustering algorithms for change analysis in the Infuse project [KaPe 87, PeKa 88, MaKa 88]. Infuse clusters the set of modules involved in a change into a hierarchy of experimental databases where the hierarchy controls the integration of changes. Their algorithms are based on interconnection strength.

Another partitioning algorithm based on interconnection strength was recently proposed by Choi and Scacchi [ChSc 90]. Their objective is to obtain a subsystem decomposition with minimal coupling and minimal alteration-distance among modules. They compute the articulation points and the biconnected components of the module graph and then build a hierarchy by assigning to each detected articulation point and each biconnected component a subsystem.

Schwanke and Platoff recently outlined a clustering measure based on shared neighbors for their ARCH environment. They intend to use this measure for summarizing call graphs, splitting large include files, and improving system modularity.

Newbery proposed a graph-theoretic approach to the problem of reducing the complexity of a directed graph [Newb 89]. She uses edge clustering as opposed to node clustering in her extendible directed graph editor (EDGE).

8. Conclusions

This work grew out of an attempt to build subsystem hierarchies for the Sun graphics libraries *sunwindow* and *suntool* [Uhl 89]. As a result of this investigation, we realized that a more general and flexible representation is needed to account for the different views and alternative compositions. Moreover, because of the size of the graphs we had great difficulties in interactively building the first level of subsystems on top of the library modules using our graph editor Rigi.

As a consequence of this investigation, we developed $(k, 2)$-partite graphs which have most of the properties of strict hierarchies, but are amply flexible for modeling. We then augmented the interactive graph editor with the clustering algorithms presented in Section 4 to be able to compose subsystem structures more efficiently. The algorithms are intended to capture the software engineering principles *high strength within a subsystem*, *low coupling among subsystems*, and *small and few interfaces among subsystems*.

On the one hand, when composing subsystem structures software engineers make intuitive or subjective decisions based on experience, skill, and insight which cannot and should not be automated. On the other hand, composition algorithms are objective with respect to a given similarity measure, but usually take only one measure into account. Providing an expert designer with a selected set of clustering algorithms through an interactive graph editor is therefore an ideal solution.

References

[BeEv 82] Belady, L.A. and C.J. Evangelisti. ‘‘System Partitioning and its Measure,’’ *Journal of Systems and Software*, 2(1), pp. 23-29, February 1982.

[ChSc 90] Choi, A.C. and W. Scacchi. ‘‘Extracting and Restructuring the Design of Large Software Systems,’’ *IEEE Software*, 7(1), pp. 66-71, January 1990.

[DuEv 82] Dunn, D. and B.S. Everitt. *An Introduction to Mathematical Taxonomy*, Cambridge University Press, 1982.

[ElMM 90] Ellis, J.A.; M. Mata-Montero; and H.A. Müller. ‘‘Serial and Parallel Algorithms for $(k, 2)$-partite Graphs,’’ Technical Report, University of Victoria, February 1990.

[FlMu 88] Fletton, N.T. and M. Munro. ‘‘Redocumenting Software Systems using Hypertext Technology,’’ In *Proceedings of Conference on Software Maintenance — 1988*, (Phoenix, AZ, October 24-27), pp. 54-59, November 1988.

[HuBa 85] Hutchins, D.H. and V.R. Basili. ‘‘System Structure Analysis: Clustering with Data Bindings,’’ *IEEE Transactions on Software Engineering*, SE-11(8), pp. 749-757, August 1985.

[KaPe 87] Kaiser, G.E. and D.E. Perry. ‘‘Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution,’’ In *Proceedings of Conference on Software Maintenance — 1987*, (Austin, TX, September 21-24), pp. 108-114, November 1987.

[MaKa 88] Maarek, Y.S. and G.E. Kaiser. ‘‘Change Management for Very Large Software Systems,’’ In *Proceedings of Phoenix Conference on Computers and Communications*, (March 16-18, Scottsdale, AZ), pp. 280-285, March 1988.

[MHHL 89] Müller, H.A; D. Hoffman; N. Horspool; and M. Levy. ‘‘Presentation of Software Development Information in K2,’’ *INFOR — Special Issue on Intelligence Integration: Part 2*, 27(2), pp. 206-220, May 1989.

[MüKl 88] Müller, H.A; and K. Klashinsky. ‘‘Rigi — A System for Programming-in-the-large,’’ In *Proceedings of the 10th International Conference on Software Engineering (ICSE)*, (Raffles City, Singapore, April 11-15), pp. 80-86, April 1988.

[Müll 89] Müller, H.A. ‘‘(K,2)-Partite Graphs as a Structural Basis for the Construction of Hypermedia Systems,’’ Technical Report Department of Computer Science, University of Victoria, DCS-119-IR, June 1989.

[Newb 89] Newbery, F.J. ‘‘Edge Concentration: A Method for Clustering Directed Graphs,’’ In *Proceedings of the 2nd International Workshop on Software Configuration Management*, (Princeton, NJ, October 24). In *ACM SIGSOFT Software Engineering Notes*, 17(7), pp. 76-85, November 1989.

[PeKa 87] Perry, D.E. and G.E. Kaiser. ‘‘Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems,’’ In *ACM Fifteenth Annual Computer Science Conference*, (St. Louis, MO), pp. 292-299, February 1987.

[Perr 87] Perry, D.E. ‘‘Software Interconnection Models,’’ In *Proceedings of the 9th International Conference on Software Engineering*, (Monterey, CA), pp. 61-69, March 30 - April 2 1987.

[ScPl 89] Schwanke, R.W. and M.A. Platoff. ‘‘Cross References are Features,’’ In *Proceedings* *of the 2nd International Workshop on Software Configuration Management*, (Princeton, NJ, October 24). In *ACM SIGSOFT Software Engineering Notes*, 17(7), pp. 86-95, November 1989.

[SeBa 88] Selby, R.W. and V.R. Basili. ‘‘Error Localization During Software Maintenance: Generating Hierarchical Descriptions from Source Code Alone,’’ In *Proceedings of Conference on Software Maintenance — 1988*, (Phoenix, AZ, October 24-27), pp. 192-197, November 1988. 1989-90 J. Uhl M.Sc., Computer Science,

[Uhl 89] Uhl, J.S. ‘‘Discovering Structure in Large Software Systems,’’ M.Sc. Thesis, University of Victoria, June 1989.

[WoCW 88] Wolf, A.L.; L.A. Clarke; and J.C. Wileden. ‘‘A Model of Visibility Control,’’ *IEEE Transactions on Software Engineering*, SE-14(4), pp. 512-520, April 1988.