Developing a Measure for Process Cluster Evaluation

THOMAS KUNZ

12. Februar 1993

TI–2/93
Institut für Theoretische Informatik
Fachbereich Informatik
Technische Hochschule Darmstadt

Developing a Measure for Process Cluster Evaluation

Thomas Kunz
Institut für Theoretische Informatik
Technische Hochschule Darmstadt

February 12, 1993

Contents

1 Introduction 2

2 Process Cluster Evaluation: Criteria and Validation 3

3 Quantitative Measures 6

4 The Example Applications 9

5 Adopting and Using the Communications–based Measures 11

6 Adopting and Using the Semantic Closeness Measure 16

7 Improving the Semantic Closeness Measure 21

8 Application of the Filtered Semantic Closeness Measure 28

9 Summary and Conclusions 31

Abstract

Our preliminary research identified multiple approaches to automatic process clustering. Prototypical tools implementing these approaches were developed. The resulting cluster hierarchies have been evaluated by comparing them to the author's understanding of the application design. This paper discusses two quantitative measures for process cluster evaluation, taken from the literature and adopted to our specific context. The first set of measures, employing information about the interprocess communication during program execution, does not appear to be useful. The evaluation of clusters with these measures differs from a human evaluation. The second measure, evaluating the degree of similarity among processes using information derived by a static source analysis, shows greater conformance with a human evaluation. This measure is improved by adding information about actual interprocess communication during runtime and an example is given that demonstrates the use of this measure.

1 Introduction

We are interested in the debugging of distributed applications written in Hermes. Hermes is a high–level, process–oriented language for distributed computing, see for example Strom et al.[26] or Bacon and Strom[4]. Processes are both the unit of parallelism and modularization, see El–Kadi and Rotenstreich[7]. Processes have their own address space which cannot be accessed from outside. They communicate and synchronize by synchronous and asynchronous message passing. A Hermes application consists of a number of process and definition modules in separate files. The definition modules define data types used within the process modules. The process modules implement one or more Hermes processes.

Kunz[11] discusses the use of process and event abstraction to manage the complexity involved in distributed debugging. Ideally, these abstractions should be derived automatically and prototypical tools have been developed. These tools and some of the results achieved are described in Kunz and Taylor[12]. So far, the result evaluation was based on the author's understanding of the application examined. We tested whether the process clusters derived form meaningful units, for example a complex server such as `pathload` in the *hello.sh* application, or provide a well-defined function such as the `getdep` cluster in the Hermes *make.none* application.

Such evaluations, however, suffer from a number of drawbacks, see also Lakhotia[13]. It is difficult for outsiders to verify the results and conclusions. Typically, the evaluation is done for smaller applications only and therefore with questionable upward scalability. Furthermore, comparisons with other methods is made very difficult.

This paper addresses the problem of deriving a *quantitative* measure for cluster evaluation. Such a measure will be instrumental for the refinement of our process clustering tool. Ultimately, an automatic quantitative evaluation could be integrated into the clustering tool. This would allow the tool to examine multiple clustering alternatives in parallel and to choose one cluster based on the quantitative evaluation. Since the work reported here

is based on ideas developed for the modularization of sequential applications, the terms *module* and *process cluster* are used as synonyms from now on.

The paper is organized as follows. Section 2 discusses global issues in the development of an evaluation measure: evaluation criteria and measure validation. Section 3 presents two quantitative measures taken from the literature in more depth. These measures will be validated using clusters judged to be good or bad by human inspection. These clusters are discussed in Section 4. Sections 5 and 6 detail the modifications to the original measures necessary to apply them in our context and report on the results. None of the two measures accurately reflects the human evaluation of the same clusters, but the second measure, utilizing information from a static source analysis, shows the greater potential. Section 7 therefore discusses a modification of the second measure. Section 8 gives an example that demonstrates the utility of such a quantitative measure. A summary of the results and an outlook on future work is given in Section 9.

2 Process Cluster Evaluation: Criteria and Validation

2.1 Evaluation Criteria

As a first step, we have to define the criteria against which we want to evaluate the process clusters. At least the following three criteria are possible:

1. usefulness for debugging,
2. recovery rate of the actual design, and
3. adherence to modularization principles.

To develop a quantitative measure, the evaluation criteria has to be operationalized as well. This is nearly impossible for the first criteria, *usefulness for debugging*. Furthermore, it is extremely difficult to single out and evaluate the contribution of a specific process cluster to the overall success of the debugging effort.

Lakhotia et al.[14] describe an approach that evaluates the goodness of an architecture recovery technique (such as a process clustering tool) by comparing the recovered design with the actual design of an application. This approach, however, suffers from the following two drawbacks. First, it becomes necessary to know the actual application design, which is often not the case. On the contrary, design recovery is typically undertaken because the application design is not known at all or the existing description is outdated and therefore inaccurate. To evaluate a quantitative measure, though, it is possible to use applications with known design. Second, and more important, this approach restrains the evaluation of recovery techniques to the degree with which they recover the original design. In general, there exist multiple designs for a given application, and the one actually used is not necessarily the best one. The evaluation method proposed by Lakhotia et al.[14], however, penalizes recovery techniques even if they derive *better* designs.

For these reasons, we focus on the development of a quantitative measure evaluating the adherence of a cluster with certain modularization principles. Furthermore, we hyphothesize that process clusters adhering to these modularization principles facilitate the debugging effort as well. This follows from the fact that modularity enhances design clarity, see also Fairley[8]. Structures of higher quality follow from the use of well–defined modularization criteria and facilitate both implementation and program maintenance.

Fairley[8] lists a number of modularization criteria, such as *information hiding*, *data abstraction*, *levels of abstraction*, *coupling–cohesion*, or *problem modeling*. All of these, and other, criteria might be used as design goals, but the concept of coupling and cohesion has most frequently emphasized. A system, according to this concept, is structured to maximize the cohesion of elements in each module and to minimize the coupling between modules. A qualitative description of different levels of coupling and cohesion can be found in Fairley[8] or Rising and Calliss[24]. The measures discussed in this paper are attempts to quantify the degree of coupling and cohesion for a given software unit.

As pointed out by Page–Jones[21], coupling and cohesion are not two completely distinct concepts but two sides of the same coin. He generalizes coupling and cohesion into one single measure, named *connascence*. Two software elements A and B are said to be connascent if there is at least one change that could be made to A that would necessitate a change to B to preserve overall correctness, see Page–Jones[21, p. 148]. Unfortunately, this measure is not quantified and the rather vague requirement is imposed that a good design eliminates any unnecessary connascence and then minimizes connascence across module boundaries by maximizing connascence within module boundaries.

In the following discussion, we will use the term *good cluster* to denote a group of application processes with a high degree of cohesion and a low degree of coupling with their environment. Process groups that do not exhibit this characteristics are called *bad clusters*.

2.2 Measure Validation

The measures discussed here are validated as follows. Using cluster hierarchies that have been judged good or bad cluster hierarchies by human inspection, they are re-evaluated using the proposed quantitative measure., The results are displayed in a diagram similar to the one shown in Figure 1.

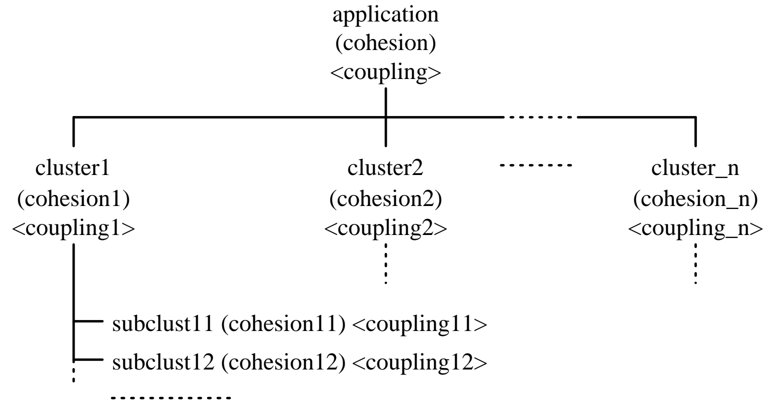

Figure 1: Evaluation output scheme

All process clusters are displayed, together with up to two values, the degrees of cluster cohesion and coupling as determined by a specific quantitative measure. To simplify the diagrams, process clusters containing only one process are omitted, since the degree of coupling and/or cohesion for such clusters is typically a default value.

Both the coupling and the cohesion value can be calculated in one of two ways. A first approach would take the cluster context into account, calculating the values for a given cluster with respect to all other clusters at the same abstraction level. Besides the problem of determining which clusters exist at the same abstraction level, the resulting cluster evaluation depends on the overall cluster hierarchy derived. To achieve a context–independent cluster evaluation, the cohesion value is determined based upon the individual processes that constitute the cluster, ignoring the internal structure of higher–level clusters. Similarly, the coupling of a cluster with its environment is calculated using the processes that form the cluster and all other processes in the application. The internal structure of higher–level clusters as well as other clusters in the environment are ignored.

A good cluster *hierarchy* is expected to show the following properties. At the level of individual processes, we have maximal cohesion but also maximal coupling. The higher we get in the cluster hierarchy, the less and less cohesive will the process clusters be. However, the clusters become less and less coupled as well. The top–level clusters have minimal coupling and cohesion. All clusters are expected to show a higher degree of cohesion than coupling, the difference between these two values might decrease for higher abstraction levels.

The preceding paragraph qualitatively describes a good process cluster *hierarchy*. A logical extension of the work reported here could be to develop a quantitative *hierarchy* evaluation measure. Such a measure would have to take many different factors into account, such as height of the hierarchy, balancedness, maximal number of subclusters in a cluster, and the individual cluster evaluations. The derivation of such a quantitative measure, however,

is lacking a firm foundation. Contrary to the evaluation of individual clusters, not much is known about what constitutes a good cluster hierarchy. Shaw[25], for example, states the need for the development for *higher–level* abstractions for larger scale systems, such as distributed applications. The paper discusses a number of potential abstractions and states that such abstractions, while used informally, are not yet systematically understood. This inability to evaluate the overall cluster hierarchy is also another reason why we neglect the hierarchy structure in the evaluation of individual clusters.

3 Quantitative Measures

This section discusses two measures taken from the literature. Later sections, which report the results obtained, will also describe how these measures were adopted to the Hermes context.

3.1 Overview

The basic ideas behind the cluster evaluation measures can be illustrated with the following analogy taken from Patel et al.[22]: If you want to find out who is a friend of X, you can go about it in one of two ways. You could identify all the individuals that communicate with X and count the frequency of these communications. The friends of X are the ones with the highest communication frequency. Alternatively, one could observe what X talks about and how often he talks about it. The friends of X will talk about the same things with approximately the same frequency as X with high probability. The two measures described in the following paragraphs are each based on one of these approaches.

3.2 Communications–based Measures

The first set of related measures was derived from work reported in Müller and Corrie[16] and Müller[15]. Similar measures have been reported in Yaung and Raz[28], but their work focuses on a slightly different problem. These measures evaluate subsystem structures in Rigi, a graph–based reverse engineering tool. The basic graph structure represents resource flow among the different modules of a software system. To adopt these measures to the Hermes context, the resource flow graph is replaced by an interprocess communication graph, assuming that interprocess communication is the primary means to achieve resource flow in Hermes. The measures described here are therefore named *communications–based measures.*

As part of Rigi, abstraction operations can be performed on the basic graph to collapse subgraphs into a single node. Three measures are provided to aid the reverse engineer in the semi–automatic abstraction. The idea is that a reverse engineer uses the abstraction operations to generate alternative abstractions and selects one based on the values of the three measures.

The measures are an attempt to quantify the software engineering principles *high cohesion within a subsystem* and *small and few interfaces* or *low coupling* among subsystems. The normalized data encapsulation quality DEQ of a weighted subgraph $G = (V, E, w)$ is calculated as follows:

$$DEQ(G) = \begin{cases} 1 & \text{if } G \text{ is a singleton node} \\ 1 - \frac{E_d(G) - C_d(G) - S_d(G)}{1 + E_d(G) + I_d(G)} & \text{otherwise} \end{cases}$$

where $I_d(G)$ is the number of data dependencies among the components of G,[1] $E_d(G)$ is the number of data dependencies between the components of G and all other components in the whole system, and $C_d(G)$ and $S_d(G)$ are the number of data dependencies absorbed due to common predecessors and successors (or clients and suppliers in the original terminology) if G is reduced to a singleton node.

Similarly, a normalized control encapsulation quality $CEQ(G)$ is defined. The normalized accumulated encapsulation quality EQ of G is the arithmetic mean of $DEQ(G)$ and $CEQ(G)$:

$$EQ(G) = \frac{DEQ(G) + CEQ(G)}{2}$$

It is argued that if the components of a subsystem are cohesive or tightly coupled, its data and control encapsulation qualities are *high*; if the components are largely independent or loosely coupled, the encapsulation qualities are *low*.

A second measure gives information about the partition quality. Let $G = (V, E, w)$ be a weighted resource flow graph again. The normalized partition quality PG of G is defined as:

$$PQ(G) = \frac{1}{1 + \frac{L(G) + n * M(G) + 2n * H(G)}{1 + n(n-1)}}$$

where n is the number of nodes in G and $L(G), M(G)$, and $H(G)$ are the number of low–, medium–, and high–strength interfaces in G, respectively. A high–strength interface in G, for example, is an edge with a weight above the high–strength threshold T_h. It is argued that the partition quality of a subsystem decomposition increases with the number of its small interfaces and the sparsity of its graph. PQ decreases with the number of medium and large interfaces and the density of G.

Finally, both measures are combined into the normalized composition quality CQ of G:

$$CQ(G) = \frac{\sum_{m \in V} EQ(m) + n * PQ(G)}{2n}$$

where n is the number of nodes in G, $EQ(m)$ the normalized accumulated encapsulation quality of a node m of G, and $PQ(G)$ the normalized partition quality of G. Information hiding (e.g. encapsulation qualities) and separation of concerns (i.e. partition quality) are considered equally important and therefore the two qualities are weighted equally, see Müller and Corrie[16, p. 12].

[1] the number of dependencies between two nodes A and B is the weight assigned to the arc connecting A and B

3.3 The Semantic Closeness Measure

Recently, a number of researchers have addressed the problem of calculating the similarity of software components, for example to find potential candidates for software reuse (see Ostertag et al.[20]), to evaluate the cohesion of composite software modules (see Patel et al.[22]), or to guide automatic system decomposition (see Paulson and Wand[23]). In terms of the reverse engineering taxonomy proposed by Chikofsky and Cross[6], most of these measures are part of a design recovery approach, utilizing outside, domain–dependent knowledge as well as knowledge obtained by an analysis of the application examined. Paulson and Wand[23], for example, base their complexity measure on an a-priori given partition of the domains of state variables. And Ostertag et al.[20] evaluate the similarity of software components based on manually assigned features from a classification library. The development of this classification library, as well as the manual assignment of descriptions/features, is a knowledge intensive, expensive operation.

The paper by Patel et al.[22] is an exception. Employing ideas from information retrieval, they develop a similarity measure that uses only information obtained by a static source code analysis. In information retrieval, a document is characterized by a set of index terms or keywords. If each document is represented as an n-dimensional binary vector, where n is the total number of keywords, the similarity of two documents X and Y can be expressed as:

$$Sim(X,Y) = \frac{X \cap Y}{|X|^{1/2} \times |Y|^{1/2}}$$

$X \cap Y$ is the set of keywords shared between the two documents and $|X|$ ($|Y|$) is the cardinality of the description vector. This measure can be generalized to n–dimensional vectors, resulting in:

$$Sim(X,Y) = \frac{X \times Y}{||X|| \times ||Y||}$$

where $X \times Y$ is the inner product of two n–dimensional vectors X and Y and $||X||$ and $||Y||$ are the magnitudes or lengths of these vectors.

One issue that remains to be solved is the process for assigning values to the vector coefficients. In the domain of information retrieval, the best known parameters for judging a term importance is its frequency count. Patel et al.[22] modified this idea and count the frequency of data type occurrence in software modules. They proceed as follows: if a reference is made to a variable V and V has type T, then this reference increments the counter associated with type T. Moreover, T might be a sub–part of some other type T'. In this case, all components in the path from T to the root of the structured type have their counters incremented. Furthermore, loop variables are filtered out, similar to the concept of stop lists in information retrieval.

They report that their similarity measure was able to identify related procedures by calculating a high similarity coefficient ($Sim \geq 0.8$) while unrelated procedures ended up with low similarity coefficients ($Sim \leq 0.4$).

To evaluate the cohesion of a composite module, a cohesion measure is computed as the average similarity measure over distinct pairs of procedures in the module:

$$Cohesion(P) = \frac{\sum_{i,j \in [1,m]: i \geq j} Sim(p_i, p_j)}{\sum_{i=1}^{m-1} i}$$

where P is a set of procedures $\{p_1, \ldots, p_m\}$. Module cohesion for well–designed Ada modules was consistently high, indicating the validity of the measure developed.

This measure uses information obtained by a static source analysis to determine the semantic similarity of two pieces of software. The pairwise similarities are then used to calculate the closeness of all pieces within a composite unit. We will therefore refer to this measure as the *semantic closeness measure.*

4 The Example Applications

To validate the quantitative measures, we evaluated clusters derived for two Hermes applications: *hello.sh* and *make.none*. This section summarizes the two applications, presents the cluster hierarchies derived, and discusses the goodness of the process clusters. Furthermore, a random cluster hierarchy for the *hello.sh* application is also given. This cluster hierarchy contains clusters judged to be bad clusters and was used to examine the behaviour of a quantitative measure when confronted with bad process clusters.

Makehermes is the Hermes version of the Unix make tool. A Hermes application consists of separately compiled process and definition modules which are imported by "linking" and "usage" lists respectively. Parseing these lists in the source reveals all dependencies, a separate *makefile* is not necessary. *Makehermes* builds a graph structure representing the discovered dependencies and checks, starting from the leaves, whether a source module has to be recompiled. Even with all the targets up-to-date, executing *makehermes* generates 175 processes (51 application processes plus 124 system processes). This specific execution of *makehermes* will be referred to as the *make.none* application from now on.

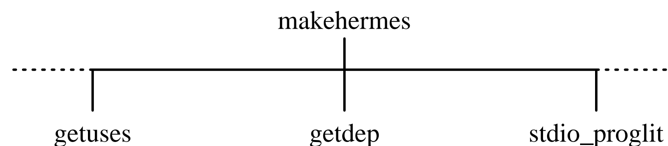

Figure 2: The cluster hierarchy for the *make.none* application

Figure 2 shows the cluster hierarchy derived for *make.none*. The application consists of one top–level cluster, `makehermes`. This top-level cluster has three non–trivial subclusters: `getdep`, `getuses`, and `stdio_proglit`. All three subclusters provide well-defined functions to the overall application. `Stdio_proglit` clusters all file I/O activities (read, write, open,

close, etc.). `Getuses` determines and returns all dependencies for a single module (e.g. the contents of the "linking" and "usage" lists) by parseing the source, and `getdep` iterates over all entries in the dependency graph, updating this global data structure. It determines the complete pathname for a module, invokes `getuses` to detect all dependencies, generates and adds capabilities for appropriate action routine (e.g. which compiler to invoke to update a module) as well as current timestamps for each dependency to the dependency graph. The `make` process uses this information later to build the application, starting from the leaves in the dependency graph. All clusters provide well-defined functions (they show *functional* cohesion) and therefore are judged to be a good.

hello.sh is the example program from the Hermes tutorial[27]. On its own, this application is trivial, but the version used here is started from the Hermes shell, which in turn is started from the Hermes cache. All these processes form part of the application too, increasing the number and variety of generated processes.

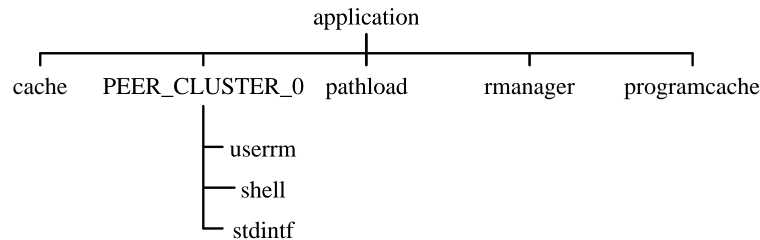

Figure 3: The cluster hierarchy for the *hello.sh* application

Figure 3 shows the cluster hierarchy derived for *hello.sh*. Similarly to Figure 2, clusters containing only one process are not shown. The clusters can be mapped back to the application in a meaningful way. The `pathload` and `rmanager` clusters consist of all the processes necessary to implement non–trivial services while `cache` contains the top-level processes of the user application. And `PEER_CLUSTER_0` summarizes all processes that make up the application started by `cache`: the `shell` application, which in turn invokes the `helloworld` application (via `stdintf`). Using the same argumentation as above, the clusters shown in Figure 3 are judged to be a good clusters as well too.

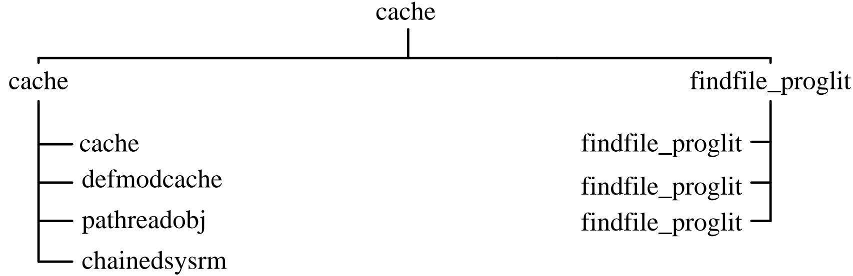

Figure 4: An arbitrary cluster hierarchy for the *hello.sh* application

Figure 4 shows an arbitrary cluster hierarchy for the *hello.sh* application. This cluster hierarchy is formed by grouping four consecutive processes (as defined by the internal process numbers) together and repeating this process until all processes are clustered into one big top–level cluster. The cluster name is the name of the first process or subcluster. This cluster hierarchy contains many bad clusters, such as the third `findfile_proglit` cluster at the lowest abstraction level or `pathreadobj`. Examining these clusters in more depth reveals that they consist of processes that have nothing in common and that it is impossible to describe their functionality.

5 Adopting and Using the Communications–based Measures

5.1 Adoption to Hermes

The encapsulation, partition, and overall composition quality measures discussed in Müller and Corrie[16] are based on a resource flow graph. Nodes represent software modules, arcs the resource flow among modules. These arcs are weighted with the number of resources shared between two connected nodes/modules. We apply these measures to a graph where nodes represent processes and arcs interprocess communication. The weights assigned to these arcs are the number of messages exchanged between two communicating processes.

The original measures depend on the existence of user–provided threshold values. These threshold values allow to classify edges into heavy, moderate, and light edges. Furthermore, the number of reduced client/server interfaces is influenced by thresholds as well. Only if the number of common successors/predecessors exceeds the respective threshold value, the resulting reduction in the number of interfaces is counted and increases the encapsulation quality. Consequently, the measures might vary widely for different threshold values. While this is acceptable in the semi–automatic setting described in Müller et al.[17], it is inadequate for a completely automatic evaluation. The original measures have therefore been reformulated to allow for a threshold–independent evaluation.

The normalized encapsulation quality $EQ(G)$ is calculated as:

$$EQ(G) = \begin{cases} 1 & \text{if } G \text{ is a singleton node} \\ 1 - \frac{E(G)-C(G)-S(G)}{1+E(G)+I(G)} & \text{otherwise} \end{cases}$$

where $I(G)$ is the number of messages exchanged among the processes of cluster G, $E(G)$ is the number of messages exchanged between processes within G and all other processes, and $C(G)$ and $S(G)$ are the numbers of edges absorbed due to common predecessors and successors if G is reduced to a singleton node.

The normalized partition quality $PQ(G)$ of a cluster G is calculated as:

$$PQ(G) = \frac{1}{1 + \frac{n*N(G)}{1+n(n-1)}}$$

where n is the number of processes in G and $N(G)$ is the number of existing interprocess communication relationships. This measure becomes more imprecise because we loose the ability to distinguish between low–, medium–, and high–traffic edges in G. All edges are treated identically, as medium–traffic edges.

Finally, both measures are combined into the normalized composition quality CQ of G:

$$CQ(G) = \frac{\sum_{m \in V} EQ(m) + n * PQ(G)}{2n}$$

where n is the number of nodes in G, $EQ(m)$ the normalized encapsulation quality of a node m of G, and $PQ(G)$ is the normalized partition quality of G. Nodes can be individual application processes as well as process clusters. This measure takes the internal structure of a cluster into account. Otherwise, the normalized composition quality would turn into:

$$CQ'(G) = \frac{PQ(G)}{2} + 0.5$$

because application processes have a default normalized encapsulation quality of 1. $CQ'(G)$, however, is only another measure of $PG(G)$ and therefore not interesting at all.

A second modification deals with the way the numbers of interfaces absorbed is counted. The original measure counts each absorbed edge with a value of 1, independent of its weight. However, this number is subtracted from the number of messages exchanged between processes within a cluster and all other processes, a value in which the weight of an edge is considered. To avoid comparing oranges with apples, we weigh the edges absorbed as follows. The weights of all edges to a common predecessor or successor except for the heaviest edge are added up. If all edges are assigned a weight of 1, EQ will be identical to the original measure, otherwise this modification increases the encapsulation quality (and therefore also the composition quality).

Figure 5 shows the collection of interprocess communication frequencies. Hermes process and definition modules are compiled by the respective compilers (`dcom` and `pcom`). A Hermes application is executed by a modified `hermes` interpreter, which generates trace files,

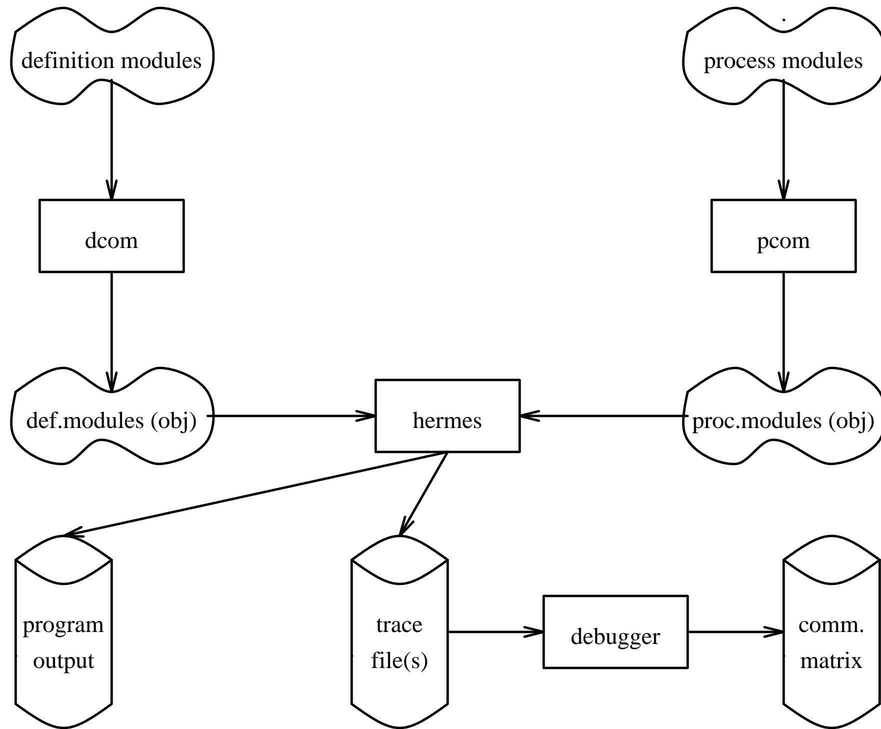

Figure 5: Layout of the communication collection

containing a raw event stream. The raw events are read in by the Hermes `debugger` and preprocessed. One user function dumps the preprocessed event set into the file `comm.matrix`, which will be read in by the cluster evaluation tool.

Alternatively, the cluster evaluation tool could read in the raw trace files directly. In this case, however, the evaluation tool would have to perform some of the error and consistency checks that are done within the `debugger` anyway (for example: exists a matching receive event for every send event). So the approach shown in Figure 5 was easier to implement.

5.2 Results

Figure 6 contains the encapsulation qualities for the *make.none* application. Obviously, the value for the top–level cluster, `makehermes`, has to be 1, since this cluster contains all application processes. For the three subclusters of `makehermes`, the values are comparatively high.

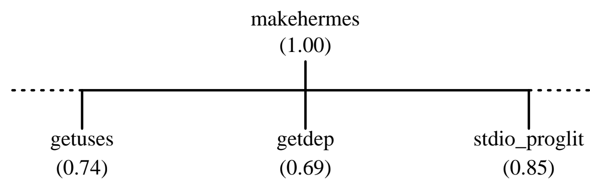

Figure 6: Encapsulation qualities of the *make.none* clusters

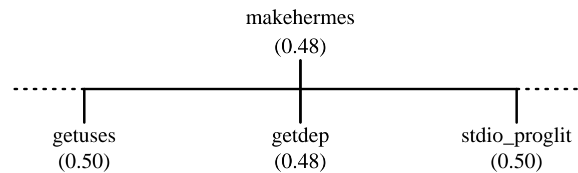

Figure 7: Partition qualities of the *make.none* clusters

Figure 7 shows the partition qualities for the *make.none* application. All values are in the neighbourhood of 0.5. Any cluster containing n user processes has approximately n internal interprocess communication edges. Very rarely does this number exceed n and in all cases communicates a process with at least one other process in his cluster, so the number of internal interprocess communication edges never falls below $n-1$. Therefore, the partition qualities of all clusters are nearly identical. This observation also holds for other applications, such as *hello.sh*.

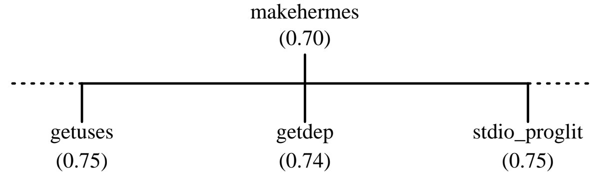

Figure 8: Composition qualities of the *make.none* clusters

Encapsulation and partition quality are combined into the overall composition quality and the resulting values are given in Figure 8. These composition quality values are determined by weighing the partition and the encapsulation quality equally. Given the small variations in the partition quality values, the resulting overall composition quality values show only small variances too.

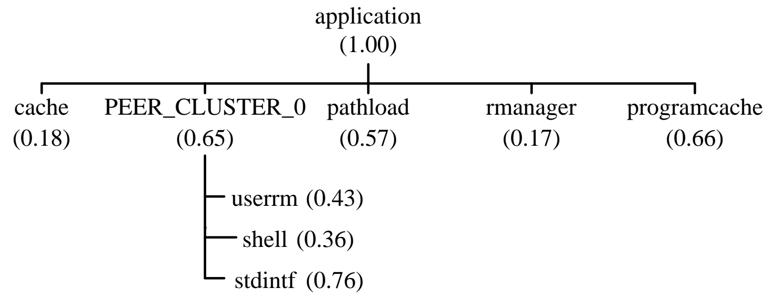

Figure 9: Encapsulation qualities of the *hello.sh* clusters

Of the three measures, only the encapsulation quality delivers values that vary enough to enable a distinction between good and bad clusters. Figure 9 presents the encapsulation quality values for the clusters of the *hello.sh* cluster hierarchy. The values vary widely, from as low as 0.17 for the `rmanager` cluster to as high as 0.76 for `stdintf` (ignoring the trivial case represented by the top–level `application` cluster).

Contrary to the similarity measure developed by Patel et al.[22], no values are reported in the literature against which to compare these results. Some insight, however, can be gained by applying the encapsulation quality measure to the arbitrary cluster hierarchy for the *hello.sh* application. Figure 10 contains the resulting encapsulation qualities.

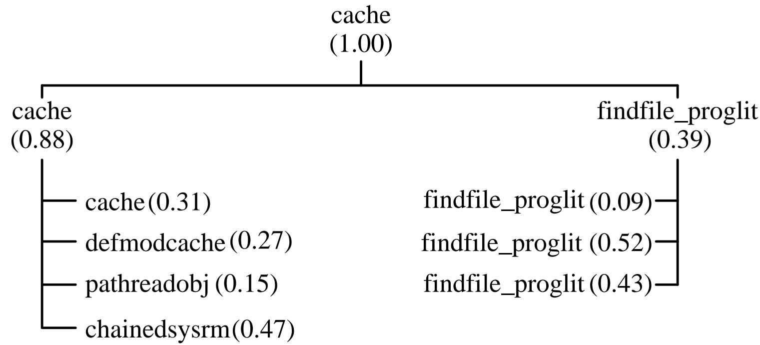

Figure 10: Encapsulation qualities of arbitrary *hello.sh* clusters

In the case of arbitrary process clusters, many encapsulation qualities are below 0.5, see for example the clusters `findfile_proglit`, `pathreadobj`, or `defmodcache`. Using 0.5 as the threshold value and demanding that a good cluster hierarchy should consist of clusters with an encapsulation quality value higher than this threshold, however, immediately contradicts the postulated goodness of the *hello.sh* cluster hierarchy. As shown in Figure 9, this cluster hierarchy contains 4 out of 8 clusters (not counting `application`) with an encapsulation quality lower than 0.5. Lowering the threshold to 0.17 or lower, so that all clusters in the *hello.sh* hierarchy are evaluated to be good, would also result in positive evaluations of the bad clusters in the arbitrary *hello.sh* cluster hierarchy.

5.3 Discussion

Of the three measures discussed, only the encapsulation quality measure varies enough to distinguish among clusters. Partition quality and overall composition quality are nearly identical for all clusters and therefore not useful.

The biggest problem preventing the encapsulation quality measure to be used as a cluster evaluation measure is demonstrated by the values reported in Figure 9. Many of the clusters we defined a priori to be good (by human inspection) are assigned low encapsulation qualities. This measure therefore seems unable to capture our notion of a good cluster.

6 Adopting and Using the Semantic Closeness Measure

A second quantitative measure evaluating process clusters is the *semantic closeness* measure. To use this measure, some modifications to the pairwise similarity measure proposed by Patel et al.[22] are necessary. This section describes the modifications and the results obtained.

6.1 Adoption to Hermes

The similarity measure defined by Patel et al.[22] assigns a characteristic vector to each module that counts the references to particular data types in its component. It has been shown that this measure works well for software units that work on the same user–defined global data types (such as a group of functions implementing a stack). The similarity of functions referencing only the built–in data types was more difficult to evaluate correctly.

In Hermes, sharing among processes is achieved by interprocess communication only. Therefore, particular emphasis is placed on communication related types. Three types strongly belong together: the type of the sending outport, the type of the receiving inport, and the type of the message exchanged. The similarity measure is based on 3–tuples of these communication types. The data types used local to a process module are ignored, similar to the suppression of loop variables mentioned above. Furthermore, all references to data types that have as component a communication related type (directly or indirectly) are counted when determining the characteristic vector for a process.

One problem that had to be solved are missing sources. The characteristic vector for each process is obtained by a static analysis of its source. In some cases, however, such a static source analysis is not possible. This can happen, for example, if the original source does not exist anymore. Another special feature of Hermes is that process modules can be written in C and be used just like regular Hermes processes, see Korfhage[10]. Any tool working on a Hermes source will not work on the C source. A default interpretation is assumed for processes for which a static source analysis cannot be performed. Staying within the friend analogy discussed above, we characterize an unknown person by assuming that he/she potentially talks about everything, though not in great detail. Therefore, the default interpretation is a characteristic vector with a 1 in each component.

Another problem is that processes can be textually defined within the source of another process. These processes (called program literals) or assigned, by the Hermes runtime system, the same name as the process within which they are textually defined, adding the suffix *_proglit*. A more sophisticated tool might be able to process multiple process definitions within one source file separately. However, this situation does not occur very often, so the tool implemented assigns the same characteristic vector to all processes implemented by one source file.

In summary, the modified similarity measure is calculated as follows. Each process is assigned a characteristic vector, either the default vector or a vector derived by a static source analysis. In the latter case, the vector entries count all references to variables of communication related types, such as inport types, outport types, message types, and compound types that contain at least one component with a communication related type. These vectors are used to calculate the pairwise similarity between any two processes as:

$$Sim(X,Y) = \frac{X \times Y}{||X|| \times ||Y||}$$

where $X \times Y$ is the inner product of the characteristic vectors X and Y and $||X||$ and $||Y||$ are their magnitudes or lengths. The *cohesion* for a process cluster P is the average pairwise

similarity among processes within this cluster:

$$Cohesion(P) = \frac{\sum_{i,j \in [1,m]: i \geq j} Sim(p_i, p_j)}{\sum_{i=1}^{m-1} i}$$

where P is a set of processes $\{p_1, \ldots, p_m\}$. Similarly, the *coupling* of a module with its environment is calculated as:

$$Coupling(P) = \frac{\sum_{i \in [1,m], j \in [1,n]} Sim(p_i, q_j)}{m \times n}$$

where P is a set of processes $\{p_1, \ldots, p_m\}$ and $\{q_j\}$ is the set of user processes not in P with $|\{q_j\}| = n$.

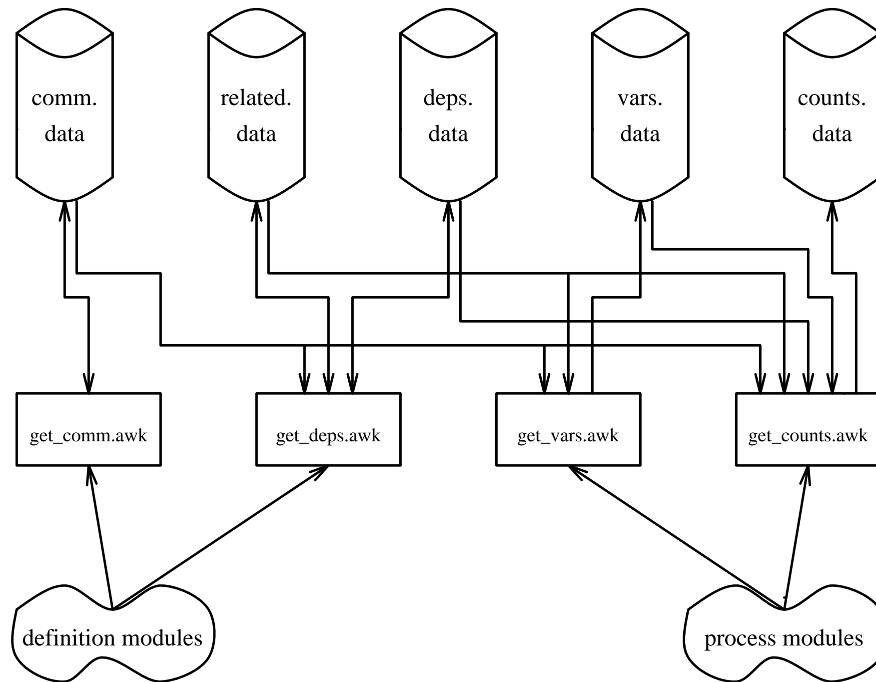

Figure 11: Design of the source analysis tool

Figure 11 gives the design of the static source analysis tool which is implemented as a collection of awk–scripts. The first script, `get_comm.awk`, collects information about inport types, outport types, and message types from all known definition modules and saves them in the file `comm.data`. This information is used by the second script, `get_deps.awk`, which repeatedly iterates over all known definition modules to collect two sets of information. File `related.data` contains all compound data types that directly or indirectly contain a component with a communication related data type. File `deps.data` contains specific information about these component relations (name of component, type).

The last two scripts are invoked for every known Hermes process module. The script `get_vars.awk` determines all identifiers for variables of a type stored in either `comm.data` or `related.data` and stores them in the file `vars.data`. The script `get_counts.awk` uses this information, together with the information about subcomponent names and types in `deps.data` to count all references to the appropriate types. The resulting characteristic vector is stored in `counts.data` and is used for the cluster evaluation.

6.2 Results

Figure 12 shows the degrees of cohesion and coupling calculated with the semantic closeness measure for the clusters in the *make.none* cluster hierarchy. According to our criteria stated earlier, the cluster hierarchy is a good hierarchy. The coupling is low for the three lower–level clusters. These three lower–level clusters also show a relatively high cohesion. With the exception of `stdio_proglit`, the cohesion value is lower than the average values reported in Patel et al.[22]. This, however, is to be expected, since information cohesion (common global data structures) as measured in Patel et al.[22] is a stronger form of cohesion than communicational cohesion, see also Fairley[8].

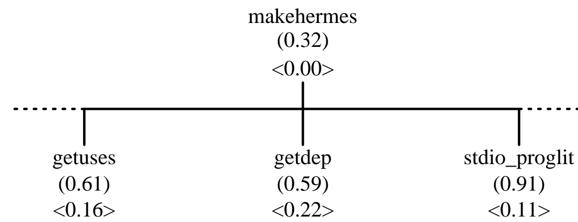

Figure 12: Semantic closeness of the *make.none* clusters

The *make.none* application, however, has one drawback when used to examine a semantic closeness measure. The three subclusters `getuses`, `getdep`, and `stdio_proglit` contain many instantiations of either identical processes or processes that get assigned the default characteristic vector. Consequently, many pairwise similarity values are 1. The *hello.sh* application is an application with a greater mix of different processes.

Figure 13 contains the degrees of cohesion and coupling for the clusters in the *hello.sh* cluster hierarchy. With the exception of `pathload`, the degrees of cohesion are smaller than in Figure 12. Furthermore, in some cases, the degree of coupling for low–level clusters such as `rmanager`, `userrm` or `programcache` is nearly as big as the cohesion value, indicating that processes within the cluster are not significantly more similar to each other than to processes outside the cluster. The most extreme example is `PEER_CLUSTER_0`, where the processes are slightly closer to processes outside the cluster than to other processes within the cluster.

In Figure 14, the cluster evaluation for the arbitrary *hello.sh* cluster hierarchy is given. The figure indicates that the degree of cohesion for arbitrary groups of processes is not

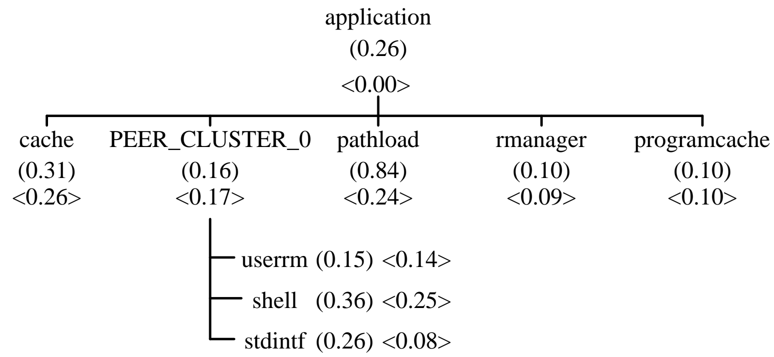

Figure 13: Semantic closeness of the *hello.sh* clusters

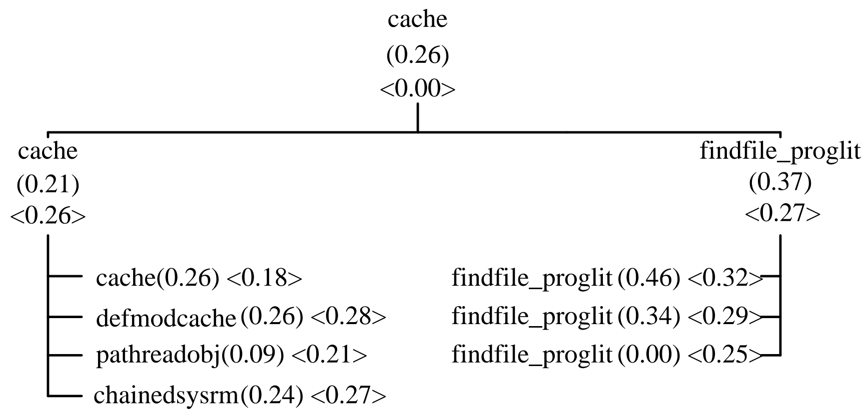

Figure 14: Semantic closeness of random *hello.sh* clusters

necessarily as high as the degree of coupling with the environment. An extreme example is the third `findfile_proglit` cluster with *zero* cohesion and a coupling of 0.25. Another clear examples is the low–level `pathreadobj` cluster. Both these clusters have been judged to be *bad* clusters by human inspection as well.

6.3 Discussion

The similarity measure defined by Patel et al.[22] has been adopted to the Hermes environment. Figures 12 and 14 indicate that the semantic closeness measure potentially distinguishes good and bad process clusters correctly. Figure 13 demonstrates that this measure delivers values that do not always allow unambiguous decisions about the cluster goodness. However, the semantic closeness measure appears to have a higher potential as cluster evaluation measure than any of the three communications–based measures discussed before. The next section discusses an improvement of this measure by including runtime information about actual interprocess communication.

7 Improving the Semantic Closeness Measure

The semantic closeness measure has shown to be superior to any of the three communications–based measures. However, even this measure does not assign coupling and cohesion values that would enable us to clearly distinguish good and bad process clusters in all cases. This is partly due to one problem described in the next paragraphs. To overcome this problem, we propose a modification to the semantic closeness measure that takes runtime information into account.

7.1 The Problem

Figure 15 depicts a hypothetical application simulating some aspects of a computer system. Three users, represented by three instantiations of the source module `User` access a `MailServer` and a `PrintServer`. These two servers use different instantiations of the same source module `FileBuffer` as interface to the file system. The indices are used only to distinguish the different instantiations in the following discussion. In the simulation "run" shown in Figure 15, most user requests to the two servers interrogate and set internal status variables, so the communication frequencies between the `User` processes and the two servers are higher than the communication frequencies between the servers and their respective copies of `FileBuffer`.

Table 1 contains pairwise similarities for all application processes. These values were assigned based on the assumption that the `User` processes communicate with the server processes over the same interface and that this interface differs from the one used between server processes and the `FileBuffer` processes. Each process is maximally similar to itself and other instantiations of the same source module. Both the `FileBuffer` and the

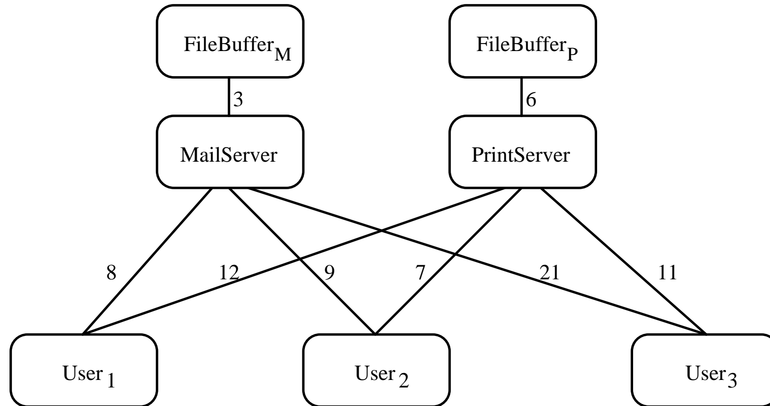

Figure 15: Example application

	FileBuffer$_M$	FileBuffer$_P$	MailServer	PrintServer	User$_1$	User$_2$	User$_3$
FileBuffer$_M$	1.0	1.0	0.5	0.5	0.0	0.0	0.0
FileBuffer$_P$	—	1.0	0.5	0.5	0.0	0.0	0.0
MailServer	—	—	1.0	0.8	0.5	0.5	0.5
PrintServer	—	—	—	1.0	0.5	0.5	0.5
User$_1$	—	—	—	—	1.0	1.0	1.0
User$_2$	—	—	—	—	—	1.0	1.0
User$_3$	—	—	—	—	—	—	1.0

Table 1: Pairwise similarity of application processes

`User` processes communicate with the server processes over one interface, so the pairwise similarity between a server process and one of the other processes is identical, arbitrarely set to 0.5. The two server processes are not necessarily completely identical, but more similar to each other than to the rest of the processes. They both know the interface to the `User` processes as well as the interface to the `FileBuffer` processes. And the `User` and `FileBuffer` processes are maximally different.

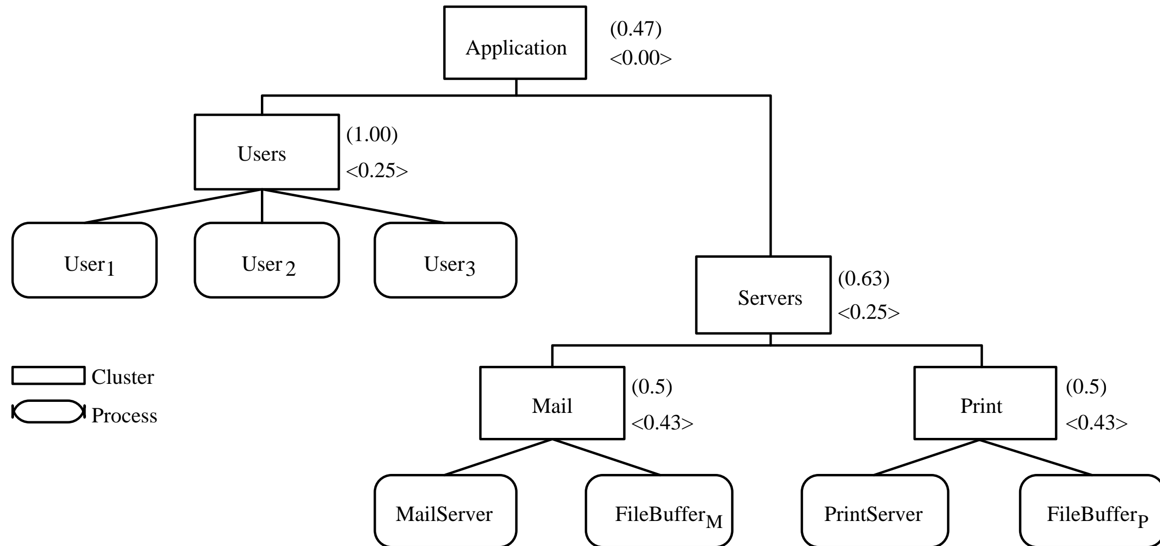

Figure 16: Cluster hierarchy for the example application

Figure 16 postulates a good cluster hierarchy for the example application plus cohesion and coupling values calculated using the semantic closeness measure. For all clusters, the cohesion is higher than the coupling. A cohesion value greater than the respective coupling value, however, does not always indicate that the cluster is good. This is examplified in Figure 17.

The characteristic vectors used to calculate the pairwise similarity are determined by a static source analysis. Consequently, `FileBuffer`$_M$ and `FileBuffer`$_P$ are indistinguishable (they are instantiations of the same source module). The clusters `Mail` and `NearlyMail` have identical degrees of cohesion and coupling, even though the second cluster is clearly worse. The values in the square brackets are the encapsulation quality values. They do not differ a lot, but hint at the basic difference between the two clusters in Figure 17. In the first cluster, the processes within the cluster communicate with each other, in the second cluster they do not. To reflect such difference in an evaluation measure, runtime information about interprocess communication has to be taken into account somehow. The encapsulation quality measure alone, while indicating this difference, is not sufficient to clearly differentiate between such cases in general.

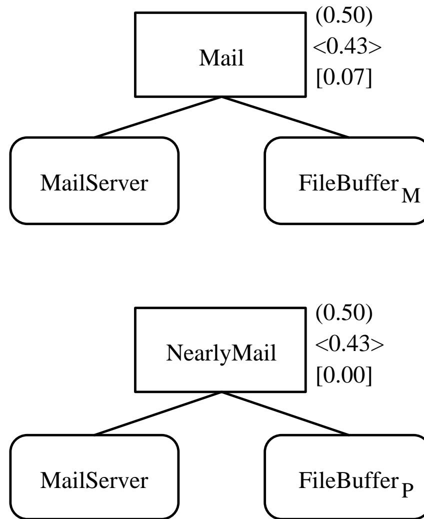

Figure 17: Problem with the similarity measure

7.2 Adding Runtime Information

Adding runtime information to improve the semantic closeness measure can be done in at least three different ways:

- NewValue = f(closeness, encapsulation), e.g. the semantic closeness measure and the encapsulation quality measure are calculated as before and combined to yield a new evaluation measure (adding them, multiplying them, weighing them equally or differently, ...)
- The rules for calculating the semantic closeness measure and/or the encapsulation quality can be changed to take the respective other value into consideration: add a component reflecting the semantic closeness to the calculation of the encapsulation quality, increase the pairwise similarity with the encapsulation quality of the process pair, ...
- A last approach modifies the input to the evaluation algorithms. Processes that do not communicate with each other could be assigned a similarity of zero. Another possibility is to multiply the communication frequencies with the pairwise similarity.

Within each of these three categories, a wide variety of possibilities exist. Exhaustively checking all possibilities is clearly impossible. Furthermore, some of these combinations would make the interprocess communication the more important criteria, though, as discussed above, the semantic closeness measure yields the better results. Reflecting on the different approaches, the third category, manipulation of the input values, reflects the observed difference directly. If, for example, the pairwise similarity of non–communicating

processes is reduced to 0, the two clusters in Figure 17 are assigned different values, with the degree of cohesion calculated for the good cluster `Mail` being higher than the degree of cohesion calculated for the bad cluster `NearlyMail`.

But filtering the pairwise similarity based upon the interprocess communication has to be done with care. As discussed above, an application consisting of n processes typically has $O(n)$ interprocess communication connections. If the filtering operation assigns a pairwise similarity measure of 0 to all processes that do not communicate directly with each other, the resulting $n \times n$ similarity matrix becomes very sparse. The semantic closeness measure is heavily influenced by such a modification, even to the point where the evaluation of process clusters is turned upside down.

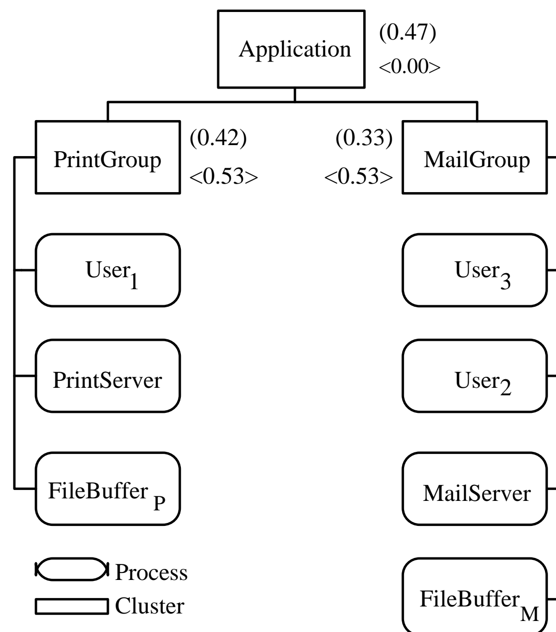

Figure 18: An alternative cluster hierarchy for the example application

Figure 18 shows an alternative cluster hierarchy for the example application of Figure 15. Each process is clustered with the server it communicates with most. This rule produces two clusters, one for each server. For both clusters, the cohesion value is smaller than the coupling value. After filtering the pairwise similarity as described above, however, the cohesion of both clusters exceeds their coupling.

So while filtering the pairwise similarity based upon the interprocess communication seems the right thing to do, the scope has to be narrowed. The section started with the observation that the similarity measure is unable to distinguish among multiple instantiations of the same source. Therefore, a filtering operation on the pairwise similarity is defined as follows:

$$Sim_f(X,Y) = \begin{cases} Sim(X,Y) & \text{if } X \text{ and } Y \text{ are instantiations of the same source} \\ Sim(X,Y) & \text{if both } X \text{ and } Y \text{ are unique instantiations of their source} \\ Sim(X,Y) & \text{if } X \text{ and } Y \text{ communicate with each other} \\ 0 & \text{otherwise} \end{cases}$$

In short, the pairwise similarity between two processes is reduced to zero if at least one of them has sibling instantiations, they are instantiations of different source modules, and they do not communicate with each other.

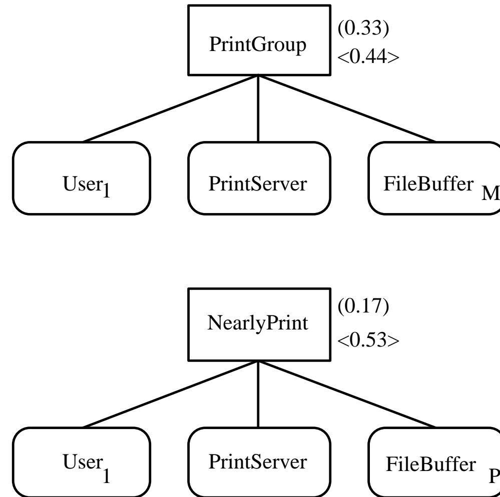

Figure 19: Filtering effect

Figure 19 shows the effect of this filtering operation on the `PrintGroup` cluster in Figure 18. Using the filtered semantic closeness measure, the cluster `PrintGroup` is assigned a cohesion of 0.33 and a coupling of 0.44. When replacing $\texttt{FileBuffer}_P$ with $\texttt{FileBuffer}_M$, the degree of cohesion is reduced to 0.17. The degree of coupling, on the other side, increases to 0.53. In both cases, the degree of coupling exceeds the degree of cohesion, indicating that `PrintGroup` as well as `NearlyPrint` are bad clusters. However, the difference between coupling and cohesion is even bigger for the worse cluster `NearlyPrint`.

7.3 Results

To verify the effects of the filtering operation, the clusters of our test hierarchies were re–evaluated with the modified semantic closeness measure.

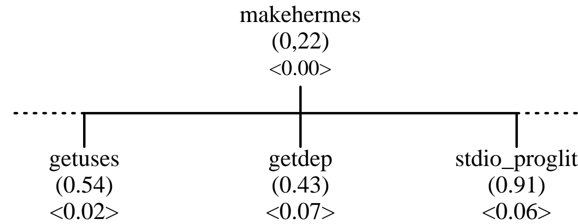

Figure 20: Filtered semantic closeness of the *make.none* clusters

Figure 20 shows the cluster evaluation for the *make.none* cluster hierarchy. Compared to the results reported in Figure 12, both the cohesion and the coupling values are smaller. This is expected, since the filtering operation reduces the pairwise similarity in some cases but never increases it. However, the coupling values are reduced more by the filtering operation than the cohesion values, indicating the goodness of the clusters even stronger than before.

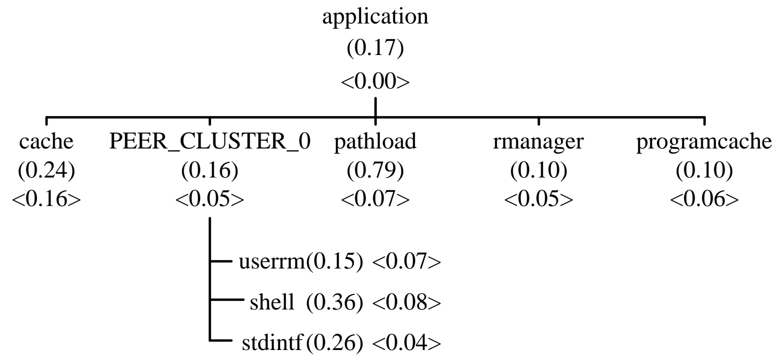

Figure 21: Filtered semantic closeness of the *hello.sh* clusters

Figure 21 presents the evaluation for the *hello.sh* application. Compared to Figure 13, the coupling and cohesion values are smaller in some case and stayed the same in other cases. Again, the degree of coupling decreased more than the respective degree of cohesion. In Figure 21, all clusters have a higher degree of cohesion than of coupling, indicating that all clusters in this hierarchy are indeed good clusters, in conformance with the human evaluation.

The evaluation of arbitrary process clusters for the *hello.sh* application is shown in Figure 22. The two bad clusters `pathreadobj` and `findfile_proglit` still show a higher degree of

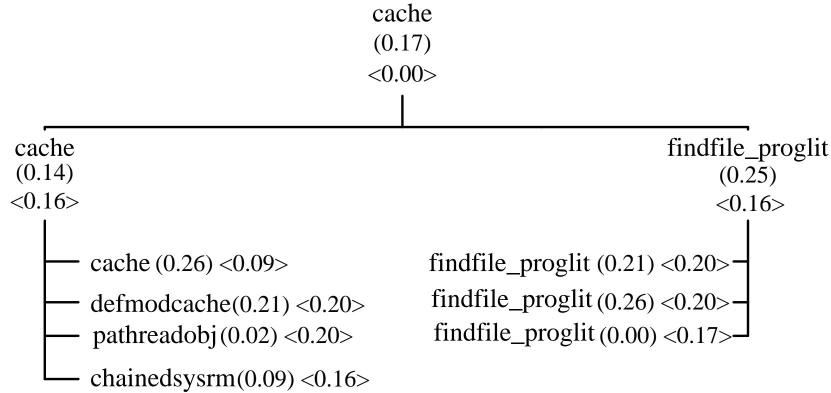

Figure 22: Filtered semantic closeness of arbitrary *hello.sh* clusters

coupling with the environment than internal cohesion. In fact, with the exception of the low–level cluster `defmodcache`, all relative evaluations (cohesion higher or smaller than coupling) remain unchanged by the filtering operation. In most cases, the absolute differences increase, both for good and bad clusters.

The `defmodcache` cluster deserves particular attention. It shows that the cluster evaluation can be turned upside down by the filtering operation. However, cohesion and coupling remain close to each other, both before and after the filtering operation. The cluster contains two (out of four) processes that cooperate closely. While this cluster is certainly not a good cluster, it is also not as bad as the two bad clusters identified previously. This ambivalence shows in the resulting evaluation as well.

7.4 Discussion

This section discussed the semantic closeness measure in more depth. A problem with this measure was pointed out and a modification to overcome this problem was proposed. This modification filters the pairwise similarity for a pair of processes, depending on whether they are unique instantiations of their respective source module and whether they communicate with each other. In essence, this modification became necessary because we use a measure developed to evaluate a static structure in a dynamic context, a process cluster hierarchy over all processes created during the execution of a distributed application. Figures 20 to 22 indicate that the semantic closeness measure accurately reflects the human evaluation of the same process clusters.

8 Application of the Filtered Semantic Closeness Measure

This section uses the filtered semantic closeness measure to evaluate different process cluster hierarchies derived for the *dcom* application. *Dcom* is the Hermes definitions module

compiler, implemented in Hermes itself. This application has an interesting property. Our prototypical clustering tool immediately clusters all application processes together. In the newer 0.8alpha version of *dcom*, the total number of processes created is 44, the older 0.7alpha version consisted of 216 processes. In the following, only the newer version is examined. The resulting cluster hierarchy, with 2 only levels of abstraction is clearly a bad hierarchy. No reduction of complexity can be achieved where either all processes have to be examined individually or are lumped together in one big cluster.

To enable control of the resulting cluster sizes, a statistical cluster analysis subroutine was added to the clustering tool. *Cluster analysis* is a generic name used to describe the numerous statistical approaches for identifying and creating classifications. Aldenderfer and Blashfield[2] identify seven major groups of cluster analysis methodologies. The majority of these approaches provide exclusive clustering, in which objects may not appear in more than one cluster. These approaches all begin with a set of data indicating the objects, or cases, to be clustered, here the application processes. Associated with the data set is a set of variables. Each case in the data set has a value for each of these variables and these values are used to differentiate between cases. In our context, the variables describe the interprocess communication behaviour of each process.

For a number of reasons, a hierarchical agglomerative clustering approach has been chosen and implemented as part of our process clustering tool. In hierarchical agglomerative approaches, each case in the data set is initially placed into its own cluster. Then, based on a *distance measure*, which defines the distance between two cases, and a *clustering method*, which defines the criteria for combining cases or clusters, two cases are joined to form a cluster. At each step, either an individual case is added to a cluster or two existing clusters are combined until all individual cases are grouped into one big cluster. For the the distance measure and clustering method chosen, the distance $D(i)$ at which the i–th clustering step is performed increases monotonically.

The implemented cluster analysis subroutine uses the constants `TOO_BIG`, `SIZE`, and `GLUE` to split up clusters that are too big as follows. Each cluster is checked for its size m. For clusters with more than `TOO_BIG` subclusters, the cluster analysis subroutine is used to split the cluster into $n < m$ intermediate subclusters, using the following algorithm. The cluster analysis tool checks the increase in the clustering distance for the formation of $k \in$ `[SIZE-GLUE, SIZE+GLUE]` clusters: $\delta(k) = D(m-k+1) - D(m-k)$. Then n is selected such that $\delta(n) = max_k\{\delta(k)\}$, that is, the cluster analysis stops joining clusters before merging two clusters that maximally increase the clustering distance for the range examined. The algorithm is based upon the assumption that bigger increases in the $D(i)$ are the result of greater dissimilarities among the clusters. The resulting n intermediate clusters are recursively checked for their size again.

To split up the big top–level cluster of the *dcom* application, the following set of constants was used: `TOO_BIG = 40`, `SIZE = 7`, and `GLUE = 2`. The top–level cluster, containing 44 subclusters (the application processes), was split into 7 intermediate subclusters, shown in Figure 23.

Figure 23 also gives the degrees of cohesion and clustering for each of the clusters. All

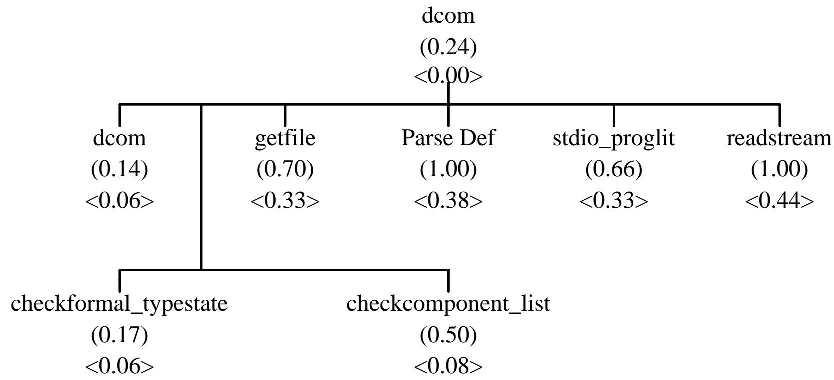

Figure 23: Splitting *dcom* into 7 intermediate subclusters

clusters have a higher degree of cohesion than coupling and none of the clusters has more than `TOO_BIG` subclusters.

One interesting question is whether the cluster hierarchy derived is the *best* hierarchy for the range of subcluster numbers examined ($k \in$ `[5,9]`). Given that we lack a quantitative evaluation of cluster *hierarchies*, this question is difficult to answer unambiguously.

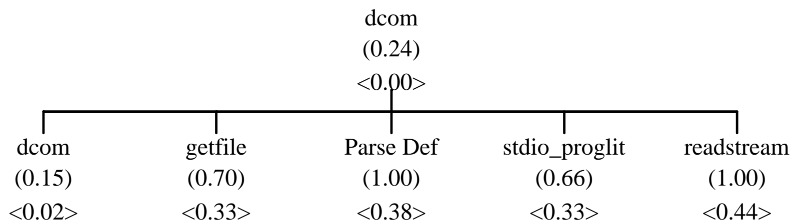

Figure 24: Splitting *dcom* into 5 intermediate subclusters

Figure 24 shows the cluster hierarchy that results from splitting `dcom` into 5 subclusters. As in Figure 23, all clusters show a higher degree of cohesion than clustering and none of the clusters is too big. Therefore, this cluster hierarchy is a good cluster hierarchy too. However, the hierarchy in Figure 23 is probably better because it splits `dcom` into more and smaller subclusters.

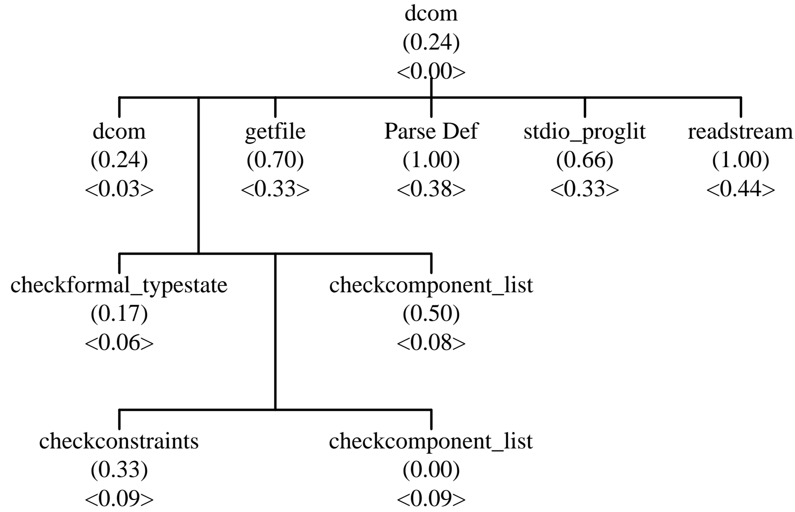

Figure 25: Splitting *dcom* into 9 intermediate subclusters

Splitting `dcom` in 9 subclusters results in the cluster hierarchy shown in Figure 25. This cluster hierarchy contains a cluster with a lower degree of cohesion than coupling, the intermediate cluster `checkcomponent_list` and is therefore judged inferior to the two preceding cluster hierarchies.

9 Summary and Conclusions

This paper discusses two quantitative evaluation measures proposed in the literature, adopts them to distributed applications written in Hermes, and checks their validity. Clusters derived for the *make.none* and *hello.sh* applications, judged as good or bad by human inspection, are used for this validation process. The semantic closeness measure, based on the pairwise similarity measure proposed by Patel et al.[22] conformed better with a human evaluation of the same clusters, particularly after adding a filtering operation, merging static and dynamic information.

An example demonstrates the application of such a quantitative cluster evaluation measure. We evaluated the clusters that result from splitting up huge clusters using a statistical cluster analysis approach. The resulting cluster hierarchy, shown in Figure 23, fulfills the requirements of a good hierarchy. Furthermore, it appears that the determination if the splitting point is done in a way that in fact results in one of the better cluster hierarchies within the range examined.

Where do we go from here? In a first step, this measure will be used to evaluate various new clustering rules, derived by an analysis of frequently used paradigms for distributed programming, see for example Nelson and Snyder[18], Ambler et al.[3], or Carriero and

Gelernter[5]. In a second step, the cluster evaluation measure could be integrated into the clustering tool. This would allow the tool to examine multiple clustering alternatives in parallel and to pick the best one.

References

[1] *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 1992.

[2] Mark S. Aldenderfer and Roger K. Blashfield. *Cluster Analysis*. Sage University Paper Series on Quantitative Applications in the Social Sciences, Series no. 07–044. Sage Publications, Inc., 1984.

[3] Allen L. Ambler, Margaret M. Burnett, and Betsy A. Zimmermann. Operational Versus Definitional: A Perspective on Programming Paradigms. *IEEE Computer*, 25(9):28–43, September 1992.

[4] David F. Bacon and Robert E. Strom. Implementing the Hermes Process Model. Technical Report RC 14518(#64891) 3/20/89, IBM T.J.Watson Research Center, Yorktown Heights, New York, USA, March 1989.

[5] Nicholas Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[6] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[7] Amr El-Kadi and Shmuel Rotenstreich. An Overview of Hermes. Technical Report GWU–IIST–91–24, Department of Electrical Engineering and Computer Science, The George Washington University, December 1991.

[8] Richard Fairley. *Software Engineering Concepts*. McGraw–Hill Series in Software Engineering and Technology. McGraw–Hill Book Company, New York et al., 1985.

[9] Leah H. Jamieson, Dennis Gannon, and Robert J. Douglas, editors. *The Characteristics of Parallel Algorithms*. The MIT Press, Cambridge, Massachusetts and London, England, 1987.

[10] Willard Korfhage. How to Write C–Hermes Code. Technical report, Polytechnic University, 6 Metro Tech Center, Brooklyn, NY, October 1991.

[11] Thomas Kunz. Distributed Debugging – A Case Study. Technical Report TI–3/92, Technical University Darmstadt, April 1992.

[12] Thomas Kunz and David Taylor. Distributed Debugging using a Reverse–Engineering Tool. In *Proceedings of the 3rd Reverse Engineering Forum*, Burlington, Massachusetts, September 1992.

[13] Arun Lakhotia. On evaluating the goodness of design recovery techniques. In *Proceedings of the 3rd Reverse Engineering Forum*, Burlington, Massachusetts, September 1992.

[14] Arun Lakhotia, Sanjay Mohan, and Pruek Poolkasem. On evaluating the goodness of architecture recovery techniques. Technical Report CACS TR–92–5–4, The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA, September 1992.

[15] Hausi Müller. Verifying Software Quality Criteria Using an Interactive Graph Editor. Technical Report DCS–139–IR, Department of Computer Science, University of Victoria, BC, Canada, August 1990.

[16] Hausi Müller and Brian D. Corrie. Measuring the Quality of Subsystem Structures. Technical Report DCS–193–IR, Department of Computer Science, University of Victoria, BC, Canada, November 1991.

[17] Hausi Müller, Brian D. Corrie, and Scott R. Tilley. Spatial and Visual Representations of Software Structures: A Model for Reverse Engineering. Technical Report DCS–169–IR, Department of Computer Science, University of Victoria, BC, Canada, September 1991.

[18] Philip A. Nelson and Lawrence Snyder. Programming Paradigms for Nonshared Memory Parallel Computers. In Leah H. Jamieson, Dennis Gannon, and Robert J. Douglas, editors, *The Characteristics of Parallel Algorithms*, pages 3–20. The MIT Press, Cambridge, Massachusetts and London, England, 1987.

[19] *Proceedings of the 3rd Reverse Engineering Forum*, Burlington, Massachusetts, September 1992.

[20] Eduardo Ostertag, James Hendler, Rubén Prieto Díaz, and Christine Braun. Computing Similarity in a Reuse Library System: An AI–Based Approach. *ACM Transactions on Software Engineering and Methodology*, 1(3):205–228, July 1992.

[21] Meilir Page-Jones. Comparing Techniques by Means of Encapsulation and Connascence. *Communications of the ACM*, 35(9):147–151, September 1992.

[22] Sukesh Patel, William Chu, and Rich Baxter. A Measure for Composite Module Cohesion. In *Proceedings of the 14th International Conference on Software Engineering*, pages 38–48, Melbourne, Australia, May 1992.

[23] Dan Paulson and Yair Wand. An Automated Approach to Information Systems Decomposition. *IEEE Transactions on Software Engineering*, 18(3):174–189, March 1992.

[24] Linda Rising and Frank W. Calliss. Problems with Determining Package Cohesion and Coupling. *Software – Practice and Experience*, 22(7):553–571, July 1992.

[25] Mary Shaw. Larger Scale Systems Require Higher–Level Abstractions. *5th International Workshop on Software Specification and Design*, 14(2):143–146, May 1989. Appeared as *ACM SIGSOFT Software Engineering Notes*.

[26] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Bill Silvermann, Daniel Yellin, Jim Russell, and Shaula Yemini. Hermes: Unix User's Guide, Version 0.8alpha. Technical report, IBM T.J.Watson Research Center, Yorktown Heights, New York, USA, March 1992.

[27] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. *HERMES: A Language for Distributed Computing.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[28] Alan T. Yaung and Tzvi Raz. Linkage Analysis of Processes. *Software – Practice and Experience*, 22(10):849–862, October 1992.