Evaluating Process Clusters to Support Automatic Program Understanding

Thomas Kunz
Department of Computer Science
University of Waterloo
Waterloo, Ont, Canada N2L 3G1

Abstract

Evaluating the design of a distributed application is difficult but provides useful information for program development and maintenance. In distributed debugging, for example, processes are often grouped together and treated as one entity to reduce the debugging complexity. We previously identified multiple approaches to automatic process clustering and prototypical tools implementing these approaches have been developed. The process clusters derived with these tools have been evaluated by comparing them to the author's understanding of the application design. This paper discusses a quantitative measure for process cluster evaluation. The measure uses information derived by a static source analysis as well as information about interprocess communication during the application execution. Experiments show that the resulting quantitative evaluation conforms with a human evaluation of the same clusters.

1 Introduction

Design evaluations are difficult but provide useful information, for example during the implementation of a new application. Our interest in design evaluation is motivated from a maintenance point of view. We are interested in the debugging of distributed applications written in Hermes. Hermes is a high–level, process–oriented language for distributed computing [1]. Processes are both the unit of parallelism and modularization. They have their own address space which cannot be accessed from outside and communicate and synchronize by synchronous and asynchronous message passing.

Understanding the behaviour of distributed applications is a very challenging task, due to their complexity. The top-down use of suitable abstraction hierarchies is frequently proposed to manage this complexity, Ideally, these abstractions should be derived automatically and appropriate tools for two complementary abstraction hierarchies are being developed in our research group: process clusters [2] and abstract events [3]. This paper addresses one issue with respect to the former abstraction technique, namely the evaluation of automatically derived clusters. Initially, this evaluation was based on the author's understanding of the application examined. We tested whether the process clusters derived form meaningful units, e.g. provide a well-defined function to the overall application. Such subjective evaluations, however, are difficult to verify for outsiders. 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 presents a *quantitative* measure for process cluster evaluation. This measure was instrumental for the refinement of our process clustering tool. The clustering tool can now examine multiple clustering alternatives in parallel and select the best one.

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 the quantitative measure developed. The measure is validated using clusters judged to be good or bad by human inspection, discussed in Section 4. A summary of the results, a brief discussion of the utility of process clustering for program understanding, and an outlook on future work conclude the paper.

2 Process Cluster Evaluation: Criteria and Validation

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 software maintenance,
2. recovery rate of the actual design, and
3. adherence to modularization principles.

To develop a quantitative measure, the evaluation criteria has to be quantifiable as well. This is nearly impossible for the first criteria, *usefulness for software maintenance*. It is extremely difficult to single out and evaluate the contribution of specific process clusters to the overall success of the software maintenance effort, such as debugging.

Lakhotia [4] describes 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. Therefore, this measure cannot be applied in general. Second, this approach limits the evaluation of recovery techniques to the degree with which they recover the original design. Usually, however, multiple designs for a given application are possible, and the one actually used is not necessarily a good one. The evaluation method proposed in [4] penalizes design recovery techniques even if they derive more appropriate designs.

For these reasons, we focus on the derivation of an internal measure, e.g., a measure that depends on attributes of the application under study only. Note that this approach precludes the use of outside information, such as application-domain specific knowledge or knowledge about the nature of the software maintenance effort. Internal measures are independent of such additional information and can therefore be calculated automatically, but might potentially be less accurate predictors of a good/appropriate design than other measures that incorporate such knowledge. The internal measure is then used to predict external properties of an application, such as reliability, understandability, and maintainability [5]. Our quantitative measure evaluates the adherence of a cluster with certain modularization principles. We hypothesize that process clusters adhering to these modularization principles facilitate software maintenance efforts. This follows from the fact that modularity enhances design clarity [6]. Structures of higher quality follow from the use of well–defined modularization criteria and facilitate implementation, program maintenance, and reuse [7].

Fairley [6] lists a number of modularization criteria, with the most important being the concept of *coupling and cohesion*. This criteria structures a system in a way that maximizes the cohesion of elements in each module and minimizes the coupling between modules. A qualitative description of different levels of coupling and cohesion can be found in [6, 8]. The measure presented here is an attempt to quantify the degree of coupling and cohesion for a given software unit. Throughout the rest of this paper, 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 the rest of the application. Process groups that do not exhibit this characteristic are called *bad clusters*.

To validate the measure developed, process clusters that have been judged good or bad by human inspection are used. These clusters are re-evaluated using the proposed quantitative measure. Both the coupling and the cohesion value can be calculated in one of two ways. A first approach takes the cluster context into account, calculating the values for a given cluster with respect to all other clusters. Consequently, the resulting cluster evaluation depends on the overall cluster hierarchy. To achieve a context–independent cluster evaluation, the cohesion value is calculated using only information about the individual application processes that constitute the cluster, ignoring the internal structure of higher–level clusters. Similarly, the coupling of a cluster with the rest of the application 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 the existence of other clusters are ignored.

3 The Quantitative Measure

Recently, a number of researchers addressed the problem of calculating the similarity of software components, for example to find potential candidates for software reuse [7, 9], to evaluate the cohesion of composite software modules [10], or to guide automatic system decomposition [11]. Most of these measures use outside, domain–dependent knowledge as well as knowledge obtained by an analysis of the application examined. A complexity measure based on an a-priori given partition of the state variable domains is employed in [11]. And [9] evaluates 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.

An approach utilizing no outside knowledge is described in [10]. Employing ideas from information retrieval, the paper presents 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 integer 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 integer 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. [10] adopted this idea and propose to count the frequency of data type occurrence in software modules. Furthermore, loop variables are filtered out, similar to the concept of stop lists in information retrieval. The results reported show that this similarity measure is 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. Module cohesion for well–designed modules was consistently high, indicating the validity of the measure developed.

The basic idea behind the measure can be illustrated with the following "cocktail-party" analogy taken from [10]: If you want to find out who is a friend of a given person 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 measure discussed here is based on the second approach. We also developed a set of measures based on the first approach, following ideas described in [12, 13, 14]. Preliminary research reported in [15] shows that this second set of measures is inferior to the measure presented here.

In Hermes, coordination 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 characteristic vector for each process is based on 3–tuples of these communication types. The local data types of 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 like regular Hermes processes, see [16]. 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 above "cocktail-party" analogy, 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 small values, such as 1, in each component.

In summary, the 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 as described above. The cluster *cohesion* is the average pairwise similarity of processes within the

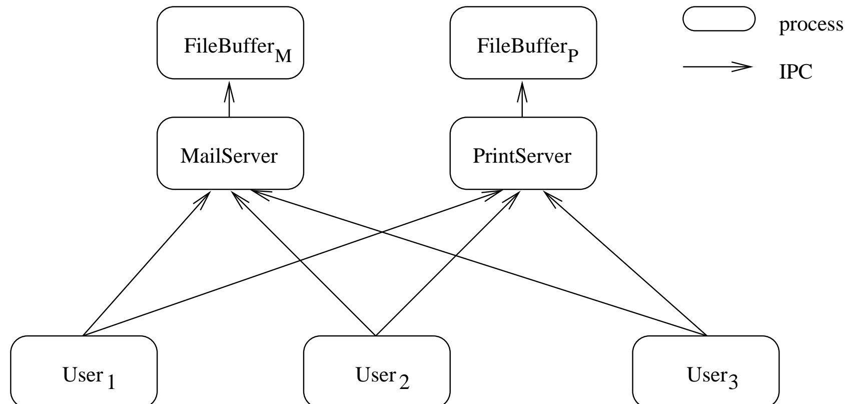

Figure 1: Hypothetical example application

	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 similarities

cluster, counting each process pair only once:

$$Cohesion(P) = \frac{\sum_{i>j} Sim(p_i, p_j)}{m(m-1)/2}$$

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

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

where P is a set of processes $\{p_1, \ldots, p_m\}$ and $\{q_1, \ldots, q_n\}$ is the set of application processes not in P.

The characteristic vectors used to calculate the pairwise similarity are determined by a static source analysis. Consequently, multiple instantiations of the same source module are indistinguishable. To demonstrate the resulting problem, the hypothetical application in Figure 1 is used. This application simulates 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.

Table 1 contains pairwise similarities for all application processes. These values were somewhat arbitrarily assigned based on the following assumptions: the `User` processes communicate with the server processes over the same interface and 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 `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, arbitrarily set to 0.5. The two server processes are 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.

In Figure 2, two potential process clusters are shown. The first cluster, `Mail`, consists of the processes `MailServer` and its associated interface to

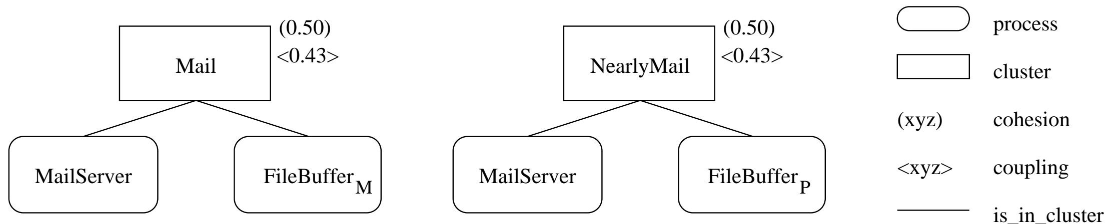

Figure 2: Two potential process clusters

the file system, $\texttt{FileBuffer}_M$. The second cluster, `NearlyMail`, has the process $\texttt{FileBuffer}_M$ replaced by the second instantiation of the same source, $\texttt{FileBuffer}_P$. The two clusters have identical degrees of cohesion and coupling, even though the cluster `NearlyMail` is worse from a design point of view. In the first cluster, the processes within the cluster communicate with each other to offer a common service to the rest of the application, in the second cluster they do not. To reflect this difference in our measure, runtime information about interprocess communication has to be taken into account. This can be achieved, for example, by reducing the pairwise similarity of non–communicating processes to 0. The resulting coupling and cohesion values for `Mail` cluster are: $Cohesion = 0.5$ and $Coupling = 0.15$. For the cluster `NearlyMail`, the quantitative values are: $Cohesion = 0$ and $Coupling = 0.25$. The difference between cohesion and coupling for the good cluster `Mail` is increased, indicating more strongly its goodness. For the bad cluster `NearlyMail`, the degree of coupling exceeds the degree of cohesion, which is the desired quantitative result.

Filtering the pairwise similarity based upon the interprocess communication has to be done with care. In our experience, distributed applications with n processes are typically not fully connected, e.g., it is not that case that every process talks to every other process. If the filtering operation assigns a pairwise similarity measure of 0 to all processes that do not communicate directly with each other, the $n \times n$ similarity matrix will become very sparse. The quantitative evaluation is heavily influenced by such a modification, even to the point where the qualitative evaluation (good or bad) of process clusters changes, see [15].

While filtering the pairwise similarity based upon the interprocess communication seems the right thing to do, the scope has to be narrowed. Given that the unmodified evaluation measure is unable to distinguish among multiple instantiations of the same source, the filtering operation on the pairwise similarity is defined as follows:

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

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.

4 Experimental Results

To demonstrate the quantitative measure, we manually evaluated clusters derived for two Hermes applications: *helloworld* and *makehermes*. This manual evaluation is then compared to the evaluation by the proposed quantitative measure. To examine the behaviour of the quantitative measure when confronted with bad process clusters, a random process cluster hierarchy for the *helloworld* application is derived and examined as well.

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. Because parsing 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.

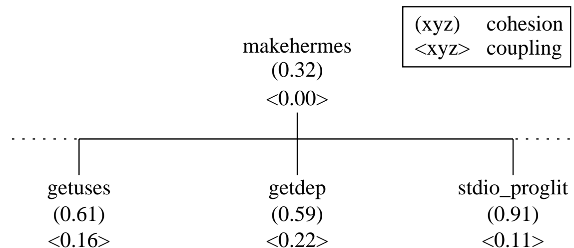

Figure 3: The cluster hierarchy for the *make.none* application

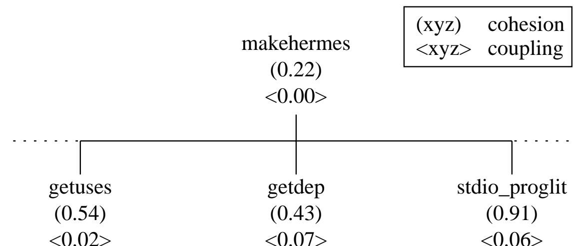

Figure 4: Filtered evaluation of the *make.none* clusters

Even with all the dependencies for a specific definition module up-to-date, executing *makehermes* generates 51 application processes. This number might seem high, but results from the fact that Hermes does not support functions or procedures. As mentioned previously, the only unit of modularization supported by Hermes is the concept of a process with own and protected address space. For example, each file I/O primitive is implemented as a separate process. This specific execution of *makehermes* will be referred to as the *make.none* execution.

Figure 3 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 parsing the source; and `getdep` iterates over all entries in the dependency graph, updating this global data structure. `getdep` determines the complete pathname for a module, invokes `getuses` to detect all dependencies, generates and adds capabilities for appropriate action routines (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 show *functional* cohesion and are therefore judged to be good.

Figure 3 also shows the cluster evaluations for the non–trivial clusters in the *make.none* cluster hierarchy, using the unfiltered pairwise similarity. The coupling is low for the three lower–level clusters. These three lower–level clusters also show a relatively high cohesion. To verify the effects of the filtering operation, the clusters were re–evaluated using the filtered pairwise similarity. Figure 4 shows the result. Compared to Figure 3, both the cohesion and the coupling values are smaller for some clusters. 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.

The *make.none* application has one potential drawback when used to validate the proposed 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. As a result, many pair-

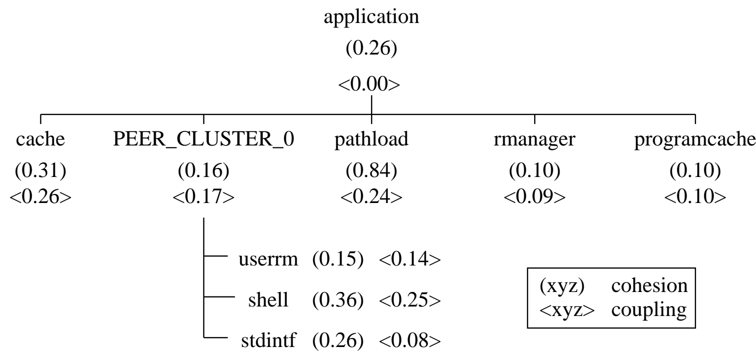

Figure 5: The cluster hierarchy for the *hello.sh* application

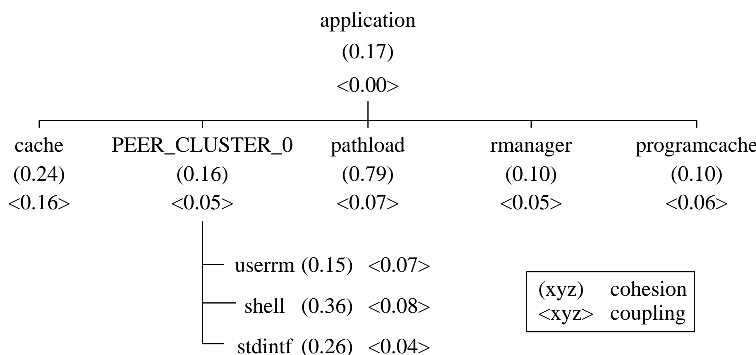

Figure 6: Filtered evaluation of the *hello.sh* clusters

wise similarity values are 1. *helloworld* is the example program from the Hermes tutorial [1] and consists of a greater mix of different processes. 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. This execution will be referred to as *hello.sh* in the following.

Figure 5 shows the cluster hierarchy derived for *hello.sh*. Similarly to Figure 3, 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 before, the clusters shown in Figure 5 are judged to be a good clusters.

Figure 5 also contains the degrees of cohesion and coupling for the non–trivial clusters in the *hello.sh* cluster hierarchy, using the unfiltered pairwise similarity measure. With the exception of `pathload`, the degrees of cohesion are smaller than in Figure 3. In some cases, the degree of coupling for low–level clusters such as `rmanager`, `userrm` or `programcache` is nearly as big as the degree of cluster cohesion. The most extreme example is `PEER_CLUSTER_0`, where the processes are evaluated to be slightly more similar to processes outside the cluster than to other processes within the cluster.

Figure 6 presents the evaluation for the *hello.sh* application using the filtered pairwise similarity measure. Compared to Figure 5, the coupling and cohesion values are smaller in some case and remain the same in other cases. As before, the degree of coupling decreases more than the respective degree of cohesion.

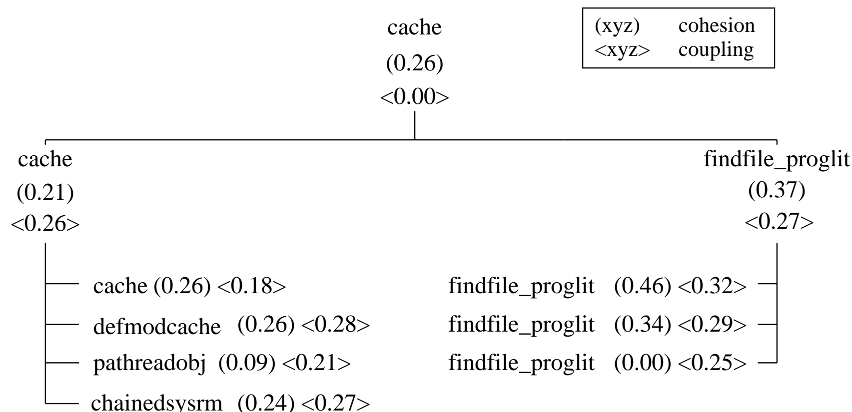

Figure 7: An arbitrary cluster hierarchy for *hello.sh*

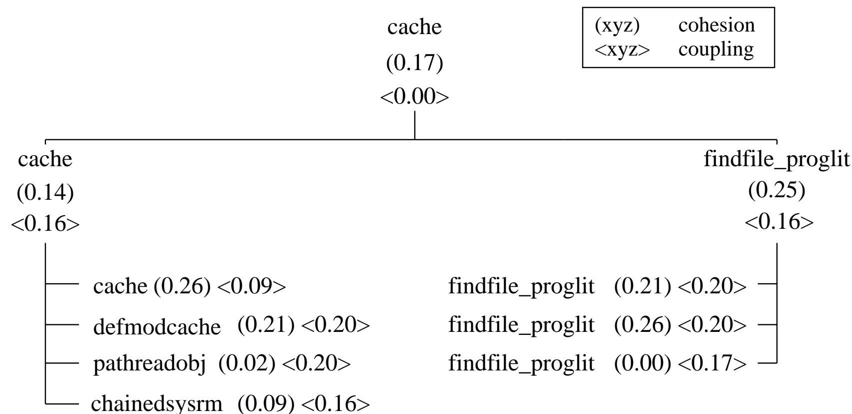

Figure 8: Filtered evaluation of the random *hello.sh* clusters

In Figure 6, 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.

Figure 7 shows an arbitrary cluster hierarchy for the *hello.sh* execution. This cluster hierarchy is formed by grouping four consecutive processes (as defined by the internal process numbers) together and repeating this process recursively 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 a number of bad clusters, such as the last `findfile_proglit` cluster or `pathreadobj`. Examining these clusters in more depth reveals that they consist of processes that belong to separate pieces of the overall application. It is impossible to describe the cluster functionality in a few words or single sentence, a typical test for high cohesion.

To examine the behaviour of our quantitative measure when evaluating *bad* clusters, this random cluster hierarchy was evaluated. The resulting values, using the unfiltered pairwise similarity measure, are shown in Figure 7. The figure contains a number of bad clusters (a lower degree of cohesion than coupling). Extreme examples are the last `findfile_proglit` cluster with *zero* cohesion and a coupling of 0.25 and the `pathreadobj` cluster. Both these clusters have been judged to be *bad* clusters by human inspection as well.

The evaluation with the filtered pairwise similarity measure is given in Figure 8. The two bad clusters `pathreadobj` and `findfile_proglit` still show a higher degree of coupling with the rest of the applica-

tion than internal cohesion. In fact, with the exception of the low–level cluster `defmodcache`, all qualitative evaluations (cohesion bigger or smaller than coupling) remain unchanged by the filtering operation. The `defmodcache` cluster deserves particular attention. It shows that the qualitative cluster evaluation can change due to the filtering operation. However, cohesion and coupling remain close to each other, both with and without filtering of the pairwise similarity. 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 clusters mentioned above. This shows in the more ambivalent results too.

In summary, Figures 3 and 7 indicate that the unfiltered measure potentially distinguishes good and bad process clusters correctly. Figure 5 demonstrates that this measure delivers values that do not always allow unambiguous decisions about the cluster goodness. Introducing the filtering operation improved the quantitative measure. Figures 4, 6, and 8 indicate that the filtered quantitative measure correlates well with a human evaluation of the same process clusters.

5 Summary and Conclusions

This paper presents a quantitative evaluation measure to evaluate process clusters for distributed applications. Clusters derived for the *make.none* and *hello.sh* executions, judged as good or bad by human inspection, are used to demonstrate the usefulness of the measure. The measure conforms with a human evaluation of the same clusters, particularly after adding a filtering operation that combines static and dynamic information.

The measure is now an integral part of our automatic process clustering tool, where it serves two purposes. First, it suppresses the derivation of clusters that exhibit higher coupling than cohesion. Second, the tool uses rules that describe process structures likely to result from the application of specific paradigms. Often, a number of rules can be applied in parallel, resulting in conflicting cluster alternatives. Rather than determining *a priori* a fixed order of rule application, which would be hard to justify, the evaluation measure is used to resolve clustering conflicts. Only the cluster with the best evaluation is built.

The process clustering tool is routinely used to derive process-cluster hierarchies for large Hermes applications. The users of this tool frequently find the hierarchies appropriate for their purpose, whether they want to debug the application or are trying to understand the application for the purpose of modification. The structure represented by the process cluster hierarchy provides valuable insight into the design of the application, supporting major changes or reorganizations. In one project, we set out to modify the `makehermes` application to support multiple concurrent compiles. The current implementation checks and if necessary recompiles source modules one at a time. This process can be speeded up by checking and recompiling siblings in the dependency graph concurrently. Given the number of processes, determining the right place for this change is non-trivial. The automatically derived process clusters provide an indication where the necessary modifications might fit into the overall structure of this application. The effort required to understand where and how to modify the original application would have been reduced if we had had our current tool at that time. In fact, our effort then more or less mirrored the extraction of the application structure which we can now do automatically.

A different example of the application of the quantitative measure is described in [15]. We evaluated the clusters that result from splitting up huge clusters using a statistical cluster analysis approach, as described in [2]. The resulting clusters are all evaluated to be good clusters. Furthermore, the implemented algorithm appears to select the number of intermediate clusters such that the resulting cluster hierarchy is, within the range examined, one of the better hierarchies.

A number of open problems still have to be addressed. The cluster evaluation might vary from execution to execution. For different sets of input data, differing numbers of processes might be created and the interprocess communication patterns might change. However, this is not a problem in a debugging context, where one is concerned with deterministically re-executing the exact same application [17]. To apply the measure in other contexts, however, a more thorough analysis of the stability of the measure with respect to various inputs is necessary.

Another area of future work is the improvement of the conflict resolution strategy in the case of conflicting process clusters. Currently, the cluster with the best quantitative evaluation is picked. In general, however, we are interested in selecting the best overall cluster *hierarchy*, not just a good process cluster. But selecting a good overall cluster hierarchy is similar to other optimization problems. Each cluster with a certain quantitative evaluation x occupies a certain range of processes or subclusters. Selecting the cluster

with highest x might lead to poor clustering in whatever is left, while picking a slightly worse cluster might increase the overall clustering quality.

The work reported here has been centered around Hermes as target environment. To explore the feasibility of our general approach to the visualization of complex distributed executions, we have ported the debugger to present behaviour in a number of different, widely–used distributed programming environments such as DCE. The static analysis necessary to derive the characteristic vectors has recently been re-implemented for C, the programming language of choice in the vast majority of these target environments. Once we finish porting other parts of the toolset to C, we plan to apply our abstraction tools to distributed applications in these environments and to use the results to improve the tools.

References

[1] 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.

[2] Thomas Kunz and James P. Black, "Using Automatic Process Clustering for Design Recovery and Distributed Debugging", *IEEE Trans. on Software Engineering*, vol. 21, no. 6, pp. 515–527, June 1995.

[3] Thomas Kunz, "Reverse Engineering Distributed Applications: An Event Abstraction Tool", *Int. Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 3, pp. 303–323, Sept. 1994.

[4] Arun Lakhotia, Sanjay Mohan, and Pruek Poolkasem, "On Evaluating the Goodness of Architecture Recovery Techniques", Tech. Rep. CACS TR–92–5–4, University of Southwestern Louisiana, Lafayette, LA, Sept. 1992.

[5] Jeff Tian and Marvin V. Zelkowitz, "Complexity Measure Evaluation and Selection", *IEEE Trans. on Software Engineering*, vol. 21, no. 8, pp. 641–650, Aug. 1995.

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

[7] Juan Carlos Esteva, "Automatic Identification of Reusable Components", in *Proc. of the Seventh International Workshop on Computer–Aided Software Engineering*, Toronto, Ontario, Canada, July 1995, pp. 80–87.

[8] Linda Rising and Frank W. Calliss, "Problems with Determining Package Cohesion and Coupling", *Software—Practice and Experience*, vol. 22, no. 7, pp. 553–571, July 1992.

[9] Eduardo Ostertag, James Hendler, Rubén Prieto Díaz, and Christine Braun, "Computing Similarity in a Reuse Library System: An AI–Based Approach", *ACM Trans. on Software Engineering and Methodology*, vol. 1, no. 3, pp. 205–228, July 1992.

[10] Sukesh Patel, William Chu, and Rich Baxter, "A Measure for Composite Module Cohesion", in *Proc. of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 1992, pp. 38–48.

[11] Dan Paulson and Yair Wand, "An Automated Approach to Information Systems Decomposition", *IEEE Trans. on Software Engineering*, vol. 18, no. 3, pp. 174–189, Mar. 1992.

[12] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl, "A Reverse-engineering Approach to Subsystem Structure Identification", *Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, Dec. 1993.

[13] Robert W. Schwanke, "An Intelligent Tool For Re–engineering Software Modularity", in *Proc. of the 13th International Conference on Software Engineering*, 1991, pp. 83–92.

[14] Alan T. Yaung and Tzvi Raz, "Linkage Metrics for Process Reengineering", in *Proc. of the 1993 CAS Conference*, Toronto, Ont., Canada, Oct. 1993, pp. 356–370.

[15] Thomas Kunz, "Developing a Measure for Process Cluster Evaluation", Tech. Rep. TI–2/93, Technical University Darmstadt, Feb. 1993.

[16] Willard Korfhage, "How to Write C–Hermes Code", Tech. Rep., Polytechnic University, 6 Metro Tech Center, Brooklyn, NY, Oct. 1991.

[17] Thomas J. LeBlanc and John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Trans. on Computers*, vol. 36, no. 4, pp. 471–482, Apr. 1987.