The Orphan Adoption Problem in Architecture Maintenance

Vassilios Tzerpos

University of Toronto
Toronto, Ontario, Canada

R.C. Holt

University of Waterloo
Waterloo, Ontario, Canada

Abstract

A lot of research time has been devoted to finding a solution to the problem of automatic clustering, especially in the field of Reverse Engineering, where decomposing a legacy system to subsystems could be the key to understanding it. Maintaining the obtained structure as a system evolves, however, is a problem that has attracted much less attention.

In this paper, we present the Orphan Adoption problem in architecture maintenance, and propose an algorithm to solve it. We also present case studies that validate the usefulness of our algorithm.

1 Introduction

The notion of "understanding" a software system takes on different meanings, depending on the level of granularity to which one refers. On a very refined level, it can refer to grasping the details of the implementation of a complicated algorithm, while on an overview level it is concerned with identifying the key components of a software system and the way they interact with each other. The latter interpretation has led to the creation of such research fields, as those of Software Architecture [GS93] and System Decomposition [Par72].

Early in the life of these research fields, it was recognized that in order to understand and model the architecture of software systems, it is necessary to introduce some new abstract entities, commonly called "subsystems". Subsystems are usually a collection of resources such as procedures, variables, source files or even other subsystems that either cooperate in implementing a well-defined part of the functionality of the system or provide related facilities to the rest of the system. Although subsystems are not necessarily explicit in the source code, research has shown that they commonly exist in the mental model developers have about their code.

While subsystem identification can be a straightforward task when utilizing expert advice from developers that understand the software system, it can be very difficult when it comes to reverse engineering a legacy system in the absence of such advice. This problem has led to extensive research on the topic of automatic subsystem identification, or as it is sometimes called *automatic clustering*. Apart from research on statistical and data clustering [Har75, JD88, Zup82], various approaches with different results have been proposed for the Software Engineering version of the problem [CS90, SAP89, MOTU93, HB85], but a general solution has yet to be found.

In this paper, we concern ourselves with a question that has attracted much less attention: what happens to the subsystem hierarchy of a software system as it evolves? In particular, we're interested in the effect of software evolution on the structure of a software system. We identify two interesting subproblems, namely that of assigning a newly introduced resource to a subsystem (incremental clustering), and that of accommodating structural changes (corrective clustering). We propose an algorithm to handle these problems and validate it by presenting three case studies.

The structure of the rest of the paper is as follows. Section 2 presents the problem and argues about the importance of a solution. Section 3 presents our algorithm, analyzes the criteria used and explains the reasoning behind them. The three case studies that validate the algorithm constitute section 4. Finally, section 5 concludes the paper and discusses future work.

2 The Problem

By applying various manual or semi-automatic methods, different CASE tools are able to come up either with a visualization of a software system's structure or with a subsystem hierarchy for it. However, if such a tool is to be used in conjunction with the software development process, it needs to always reflect the structure of the system as it evolves in each successive version of the source code. Therefore, changes to the structure of the system have to be accommodated.

We identify the following two problems as being important for a reverse engineering or redocumentation tool such as the Software Bookshelf [F^+97]:

1. *Orphan adoption.* We define an *orphan* as a newly introduced resource (e.g. procedure, variable or source file) to the system. An orphan adoption process will attempt to accommodate the orphan in the already existing subsystem hierarchy by assigning it to a subsystem. This process is called *incremental clustering.*

2. *Kidnappee adoption.* A new version of a software system might introduce structural changes that might demand that a resource be moved from its original subsystem to a new one. This is effectively equivalent to "kidnapping" a resource from its parent subsystem — thereby making it an orphan — and then readopting it to the appropriate subsystem. This process is called *corrective clustering.*

For the rest of the paper we will refer to both versions of this problem as "the Orphan Adoption problem".

An effective solution to this problem is important to a CASE tool for the following reasons:

- The visualizations provided by the tool have to be up-to-date, otherwise they are misleading to the developers.

- The layout of a visualization may convey important semantic information to the developers (e.g., the way procedures are laid out can imply their order of execution). Therefore, a minimal solution to the Orphan Adoption problem is required, in order to ensure that the appearance of the visualizations will be affected as little as possible.

- A corrective clustering algorithm can be used not only in order to "readopt kidnapped resources", but also to point out peculiar structural patterns. This in turn can lead to bug detection or to structural improvements that will increase the maintainability of the system.

As a result, there exists a need to devise criteria for adopting or readopting orphans. We distinguish the following categories of criteria:

1. *Naming criteria.* These criteria rely on the fact that a naming convention for resource names might be sufficient to identify the subsystem to which resources belong. It is very common for large software products to follow such a naming convention, (e.g., the first two letters of a file name might identify the subsystem to which this file should belong).

   Such criteria are, of course, specific to each software project, but a flexible regular expression analyzer should solve the problem fairly easily. The disadvantage of using naming criteria is that they rely on the developers to follow the naming convention, but this may not actually be the case.

2. *Structural criteria.* If a number of dependencies (such as procedure calling or source inclusion) between resources are defined, then structural criteria can be used as a basis for determining an appropriate parent subsystem for an orphan. This is done by examining the way the orphan interacts with the contents of the candidate subsystems.

   Typical examples of structural criteria include adopting the orphan to the subsystem it displays the largest connectivity with [Nei96], or to the subsystem whose interface size (number of externally visible resources) will decrease the most upon the incorporation of the orphan.

3. *Stylistic criteria.* These criteria attempt to take into account the style with which developers create or researchers decompose a system. For example, two kinds of subsystems are commonly identified in most systems [CTH95]: ones that implement a part of the functionality of the system (high cohesion subsystems), and ones that provide utilities for the rest of the system (low cohesion subsystems). A stylistic criterion will attempt to classify an orphan as a utility or not, and adopt it accordingly.

4. *Semantic criteria.* This kind of criteria attempts to adopt orphan resources by analyzing their source code and deciding to which part of the functionality of the system they refer. For example, a semantic criterion might figure out that a newly introduced file implements a user interface function. It will then place it in the corresponding subsystem. As a result, the automatic use of semantic criteria needs to employ program understanding techniques. Therefore, such criteria are much harder to implement.

In the next section we present an algorithm we used in solving the problem of Orphan Adoption. We also include an explanation of the criteria we used and with what priorities, since it is possible that different criteria might wield different results for the same orphan.

3 The Orphan Adoption Algorithm

After a number of experiments, we have devised an algorithm to deal with the problem of Orphan Adoption. The main flow of the algorithm is depicted in figure 1.

3.1 Overview

The algorithm accepts as input the name of a resource (i.e., the name of the orphan in question) and emits as output the name of the subsystem that has been chosen as an appropriate parent for the orphan. The predicate KIDNAPPED is true if the orphan was already part of a subsystem, and the predicate UTILITY is true if the parent of a "kidnapped" orphan was stylistically designated as a utility subsystem. Also, the predicates "#=1" and "#=2" are true if the cardinality of the parent subsystem was respectively one or two[1]. More than one predicate in the same condition are implicitly joined by an OR operator.

As can be seen in figure 1, naming criteria take precedence over structural ones in our algorithm. Also, stylistic criteria are only used for "kidnapped" orphans (if the parent was a utility subsystem or it had a cardinality of 1 or 2, then the orphan is reassigned to the original parent). Finally, our algorithm does not utilize any semantic criteria.

3.2 Structural Criteria

The use of the structural criteria requires further elaboration. In order to present them in a formal fashion, let us define the following:

- Let $S = \{s_i, i : 1..n_s\}$ be the set of all subsystems and $R = \{r_i, i : 1..n_r\}$ be the set of all resources. Note that since subsystems are considered to be resources, we have $S \subset R$. We define the relation $p : R \rightarrow S$, such that the pair (r_i, s_j) belongs in p, iff the subsystem s_j is the parent of resource r_i in the current subsystem hierarchy. We write $p(r)$ to denote the parent of resource r.

- Let $d = \{d_i, i : 1..n_d\}$ be the set of all defined dependencies between resources (such as procedure call or source inclusion). Each dependency d_i is a set of ordered pairs. The first element of each ordered pair is related to the second one with dependency d_i. For example, if d_3 denotes source inclusion and $d_3 = \{(r_1, r_2), (r_3, r_4)\}$, then resource r_1 includes resource r_2 and resource r_3 includes resource r_4.

  We define $D = \cup_{i=1}^{n_d} d_i$. We write $r_1 D r_2$ if resource r_1 depends on resource r_2.

- Let $W(s, r)$ be the weight of the induced dependency between a subsystem s and a resource r. More formally, $W(s, r) = |T|$, where $T = \{u | p(u) = s \wedge uDr\}$.

The structural criteria in our algorithm are used to assign a parent to an orphan o as follows:

$$p(o) = s_k, \text{ if } W(s_k, o) \geq W(s_i, o), \forall i \in 1..n_s$$

In other words, the algorithm places the orphan in the subsystem that seems to depend most on it (tie-breakers are discussed below). This criterion was deemed superior to the alternative one, of examining as well subsystems that the orphan depends on. This was based on the assumption that a newly introduced resource of a non-utility subsystem will primarily provide facilities for other members of its subsystem, while it can just as well depend on resources belonging in different subsystems. Common programming practice and a number of experiments with different systems suggest that this is a valid assumption.

3.3 Tie-breakers

In order to deal with cases where more than one subsystems exhibit maximum weight of dependency to the orphan, our algorithm employs two tie-breakers. In order of precedence, they are:

1. Let us define the interface of a subsystem s as the set $I(s)$, where $I(s) = \{r | p(r) = s \wedge \exists r' : p(r') \neq s \wedge r' D r\}$. If the addition of the orphan in one of the candidate subsystems decreases the cardinality of its interface, then this subsystem is preferred over the other candidates. Note that the orphan will always be part of the interface in this case, but its addition might "hide" some members of the previous interface.

2. The set of supplier subsystems for a subsystem s is defined as $Sup(s) = \{s' | s' \neq s \wedge \exists r : p(r) = s \wedge \exists r' : p(r') = s' \wedge r D r'\}$. Our algorithm will choose to adopt the orphan in such a subsystem, so that its supplier set is increased as little as possible.

If none of the tie-breakers manage to resolve the tie, then the orphan is adopted arbitrarily in one of the candidate subsystems.

[1] The reasoning behind this is the following. The contents of such small subsystems will always be adopted in other subsystems using structural criteria, since they cannot have any strong ties to their subsystem. However, if these subsystems exist in the decomposition, it probably means that the developers feel that they should be there because of their semantic importance.

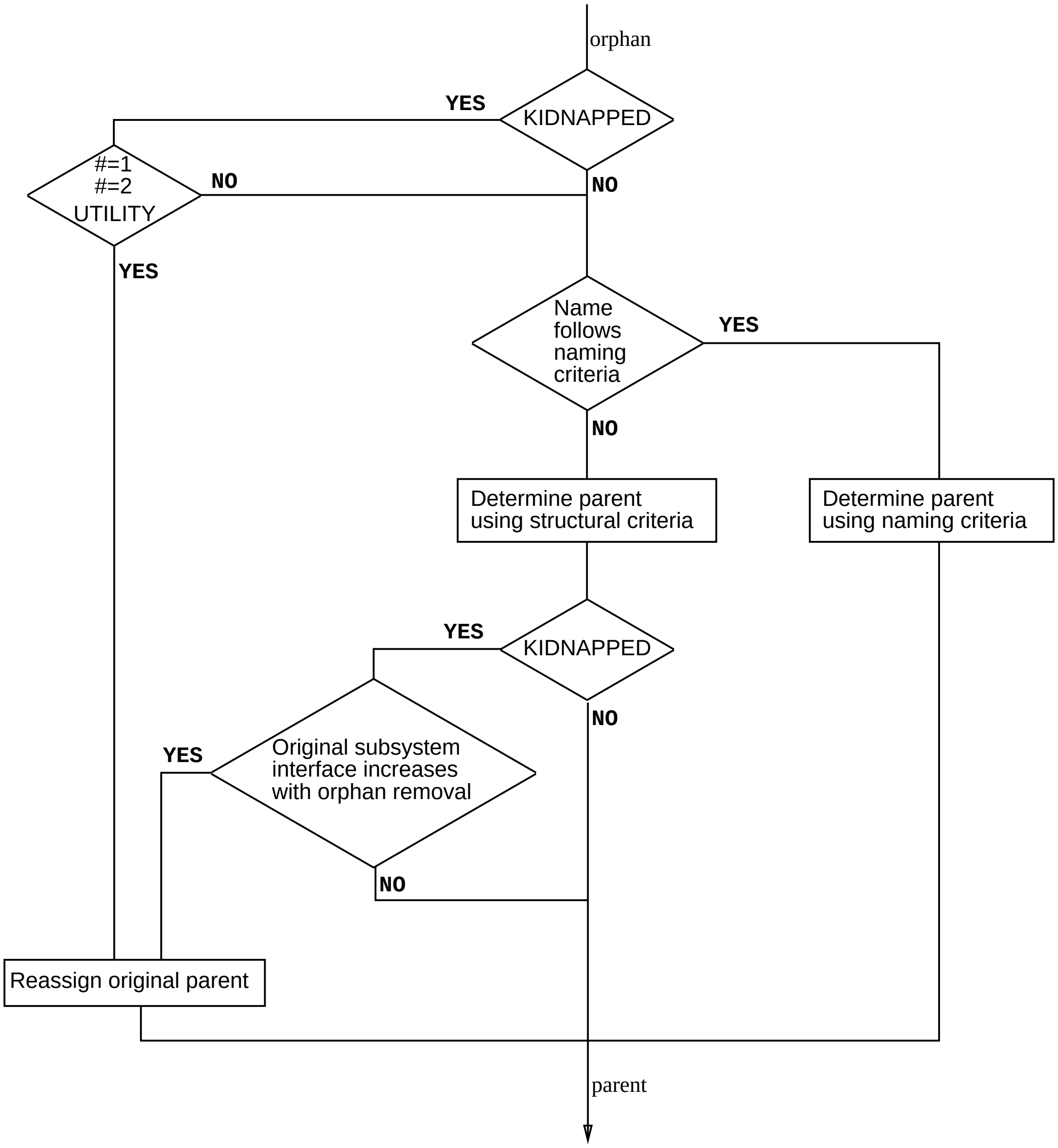

Figure 1: Flowchart of orphan adoption algorithm.

3.4 Interface Minimization

The aforementioned assumption, that an orphan o will primarily provide facilities for other members of their subsystem, is generally true only if o is a newly introduced resource to the system. As a result, if o is a kidnapped resource, this assumption does not necessarily hold. In this case, it is more likely that the orphan is a point of entry to its subsystem, and that only external resources depend on it. Because of this, its removal from the subsystem would be ill-advised. That is why our algorithm has the following extra check for kidnapped orphans.

If the cardinality of the interface of a subsystem s increases when o is removed from it, this implies that o was a point of entry to the subsystem. Using this information, our algorithm will not remove o in this case.

4 The Case Studies

In order to evaluate the validity of our algorithm, we conducted three case studies with systems of various sizes. In each of these studies, the resources that are being clustered into subsystems are source files.

As shown in the results below, the effectiveness of our algorithm was established, not only in dealing with the Orphan Adoption problem, but also in identifying parts of the system that need restructuring.

4.1 Case study 1 : MiniTunis

MiniTunis is a toy operating system that has been used to teach the Operating Systems course in the University of Toronto for a number of years. It consists of approximately 7,000 lines of code written in Object Oriented Turing. The structure of the system was well-known to the authors, and can be seen in figure 2 in a diagram produced by the Landscape tool [Pen92].

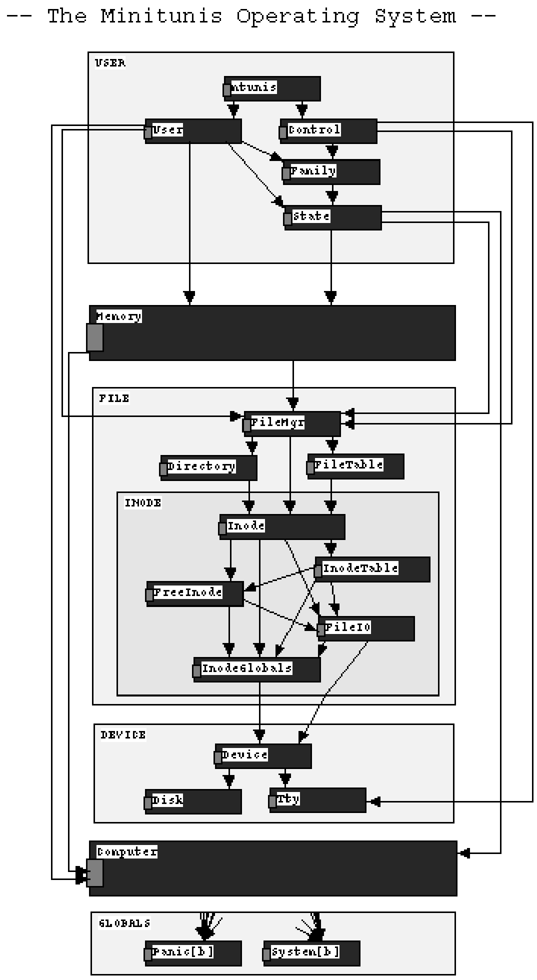

Figure 2: The system decomposition of the MiniTunis operating system.

Since the system is not evolving, there is no incremental clustering to be done. However, we used this case study in order to verify that the corrective clustering part of our algorithm will do a reasonable job with a system whose structure is well-known. The results were very encouraging, since all modules (the dark boxes in the diagram) were readopted back into their original subsystem (see Table 1).

4.2 Case study 2 : Neuma

In our next case study, we considered evolving systems rather than established ones. Neuma, a company specializing in configuration management, kindly supplied us with data from 10 different versions of one of their products (approx. 80,000 lines of code each). This data contained information about source filenames, source inclusion relations between files and system decomposition for each version.

In the experiment we conducted, we took the system decomposition for version $n-1$ and the source inclusion data for version n, and attempted to come up with a system decomposition for version n by adopting all the orphans that version n introduced. By comparing the output of our algorithm to the actual decomposition of version n provided by the developers, we were able to evaluate the effectiveness of our algorithm.

Out of 17 orphans introduced in the system at any point during its evolution, 14 of them were adopted correctly (a percentage of 82%). Also, the corrective clustering part of our algorithm, when applied to all 10 versions, readopted again all resources to the same subsystems from which they had been "kidnapped".

4.3 Case study 3 : TOBEY

Our last case study was based on an industrial system that is still under development. TOBEY is a

system name	lines of code	dependency used	number of resources	incremental clustering	corrective clustering
MiniTunis	7,000	import	20	—	100%
Neuma	80,000	include	40-57	82%	100%
TOBEY	250,000	proc call/data ref	939	88.8%	95.1%

Table 1: Summary of results.

	MiniTunis	Neuma	TOBEY
Naming Criteria	0%	0%	63.9%
Structural Criteria	65%	45.7%	15.2%
Stylistic Criteria	20%	0%	18.7%
Interface Minimization	15%	54.3%	2.2%

Table 2: Breakdown of correct results by criterion category.

large system (approx. 250,000 lines of code) written in a proprietary procedural language. The data we acquired included source filenames, as well as procedure call and data reference relations. By utilizing a naming convention (that is not followed consistently throughout the project), and by obtaining initial help from the developers, we were able to come up with a first approximation of a system decomposition for TOBEY.

We then applied the corrective clustering part of our algorithm to this decomposition. The result was a list of suggestions in order to improve the decomposition. Out of 939 source files, 46 were misplaced according to the algorithm. From them, 33 were valid suggestions according to the developers. The rest of them, while not necessarily wrong, were not optimal. In some cases, the developers felt that the orphan should stay in the original subsystem, but some restructuring should take place in order to justify this.

At a later stage, we acquired data from a more recent release of TOBEY and tried to adopt the orphans found in it. Out of 89 newly introduced resources, 79 were adopted correctly according to the developers (a percentage of 88.8%).

4.4 Observations

Table 1 summarizes the results of our case studies. Table 2 shows which criterion was responsible for what percentage of the correct results our algorithm produced for each of the systems examined. From the data in the two tables, we can make the following observations about the performance of our algorithm in dealing with the three example systems:

- Our algorithm provides a satisfactory solution to the problem of Orphan Adoption. It helps to reduce significantly the manual effort required to produce a system decomposition for a new version of a system.
- The corrective clustering part of the algorithm makes valid suggestions for resource migration, and it is able to identify parts of the system that might be in need of restructuring.
- Our algorithm seems to be effective for various data models, since it handled three different cases successfully (import relations for MiniTunis, source inclusion for Neuma, procedure call and data reference for TOBEY).
- All the criteria utilized by our algorithm seem to contribute in its effectiveness. Even when naming conventions are not available, as in the case of the first two systems, the rest of the criteria share the task of correctly (re)adopting orphans.

5 Conclusions – Future work

We have presented the problem of Orphan Adoption that occurs during the evolution of a software system. It is a problem that CASE tools need to solve effectively in order to support the software development process.

We have also proposed an algorithm to solve this problem, and described case studies that suggest that the algorithm can deal with it effectively. More case studies have to be conducted in order to further validate the effectiveness of the algorithm.

Future work also includes improvements for the algorithm, such as detecting orphans that might belong to a utility subsystem, dealing with more drastic structural changes such as the introduction of a new subsystem, or possibly an extension of the algorithm to the automatic clustering problem, utilizing the fact that the corrective clustering part of our algorithm can help rectify an "almost correct" decomposition.

Acknowledgments

The work presented here has been supported by the Consortium for Software Engineering Research (funded by IBM Canada and by the Natural Science and Engineering Research Council of Canada), as well as by the Information Technology Research Center of Ontario. The TOBEY case study was made possible by the access that IBM allowed to the system's source code and by Jim McDaniel's extracting parser, which he built as part of the CSER research project. Michael Bentley contributed to this paper by doing preliminary research in his Bachelor's thesis in the area of orphan adoption.

References

[CS90] Song C. Choi and Walt Scacchi. "Extracting and restructuring the design of large systems". *IEEE Software*, pages 66–71, January 1990.

[CTH95] Ian H. Carmichael, Vassilios Tzerpos, and R.C. Holt. "Design Maintenance : Unexpected Architectural Interactions". *International Conference on Software Maintenance*, pages 134–137, October 1995.

[F+97] P. Finnigan et al. "The Software Bookshelf". *IBM Systems Journal*, 1997.

[GS93] D. Garlan and M. Shaw. "An Introduction to Software Architecture". *Advances in Software Engineering and Knowledge Engineering*, 1, 1993.

[Har75] John A. Hartigan. *"Clustering Algorithms"*. John Wiley & Sons, 1975.

[HB85] David H. Hutchens and Victor R. Basili. "System Structure Analysis: Clustering with Data Bindings". *IEEE Transactions on Software Engineering*, 11(8):749–757, August 1985.

[JD88] A. Jain and R. Dubes. *"Algorithms for Clustering Data"*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[MOTU93] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. "A reverse engineering approach to subsystem structure identification". *Journal of Software Maintenance: Research and Practice*, 5:181–204, December 1993.

[Nei96] James M. Neighbors. "Finding Reusable Software Components in Large Systems". *Working Conference on Reverse Engineering*, pages 2–10, November 1996.

[Par72] D.L. Parnas. "On the Criteria to be used in decomposing Systems into Modules". *Communications of the ACM*, 15:1053–1058, December 1972.

[Pen92] David A. Penny. "The Software Landscape: A Visual Formalism for Programming-in-the-Large". *Ph.D. Thesis, Department of Computer Science, University of Toronto*, 1992.

[SAP89] R.W. Schwanke, R.Z. Altucher, and M.A. Platoff. "Discovering, Visualizing, and Controlling Software Structure". *Siemens Corp. Research, Inc., 755 College Rd East, Princeton, NJ 08540*, 1989.

[Zup82] J. Zupan. *"Clustering of Large Data Sets"*. Research Studies Press, England, 1982.