Electronic Notes in Theoretical Computer Science 82 No. 5 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 17 pages

Architecture Recovery by Semi-Automatic Component Identification

Jonas Lundberg [1]

Software Tech. Group, MSI
Växjö University
351-95 Växjö, Sweden

Welf Löwe [2]

Software Tech. Group, MSI
Växjö University
351-95 Växjö, Sweden

Abstract

Architecture recovery is the process of analyzing a system in order to identify the system's components, and their connectors. This paper gives a presentation of dominance analysis, and how it can be used to identify software components in object oriented legacy systems. The actual dominance analysis is applied on a high level representation of the system that we refer to as the class graph, a directed graph where the nodes are the system classes, and the edges correspond to classes interacting with each other. The result indicates that dominance analysis is a useful tool to identify certain types of components (the passive components) but insufficient to recover the complete system architecture. We also discuss how dominance analysis can be combined with other identification methods to recover a larger part of the architecture, and a novel approach to merge the results from different identification methods into a single data structure, a semi-lattice over the power set of all system classes, that permits a unified view of the different results.

1 Introduction

A thorough system understanding is necessary before any maintenance or component reuse can take place. To get this understanding is often a time consuming process since most legacy systems are usually sparsely (or inadequately) documented. Estimates of the proportion of resources and time

[1] Email: jonas.lundberg@msi.vxu.se
[2] Email: welf.lowe@msi.vxu.se

devoted to maintenance range from 50% to 75% [7,8]. The greatest part of the software maintenance process, in turn, is devoted to understanding the system being maintained. Fjeldstad and Hamlen report that 47% and 62% of time spent on actual enhancement and correction tasks, respectively, are devoted to comprehension activities [9].

Two ways to gain system comprehension reverse are engineering and architectural recovery. Reverse engineering is the process of analyzing a system in order to identify the system's components, their interrelationships, and to representations the system at a higher level of abstraction [1,2,3]. Architecture recovery [4,5,6], is an activity related to reverse engineering. Whereas reverse engineering leaves unspecified what is actually meant by a "higher level of abstraction" - architectural recovery defines it to be the software architecture of the system.

There are still debates about the definition of software architecture, but most agree that it should include at least components and connectors and their hierarchical decomposition. Components are the computational parts and connectors describe the interactions between these components [10,11]. The software architecture can be viewed as the skeleton of the system. Having this skeleton description enables the maintainer to identify load-bearing and potentially weak parts that need to be carefully addressed when a system is to be evolved. Furthermore, having a clear picture of a component's dependencies allows one to modify the component itself without affecting other parts of the system or to change the dependencies in order to handle evolving concerns about performance, interoperability, and reuse.

The dominating activity in architecture recovery is to identify *possible* architectural entities (components and connectors) in a given system. We emphasize the word "possible" since we, in general, can not definitely detect these entities. This is of course due the vague definitions of the entities to detect, the component and the connectors. What we can do is to detect a set of *candidate entities*, and to present them to a user which then have the final decision whether they should be considered as architectural entities or not. That is, we advocate a semi-automated process where software analysis tools are used to identify possible candidate entities, and where a system engineer decides whether the identified entities are to be considered as architectural entities of the system at hand.

Many different approaches to identify architectural entities in object oriented systems have been presented the last few years. We have different component identification methods where one tries to group related classes into components using different heuristics like software metrics [12,13] and common properties [14,15,16]. The dominating method when it comes to connector identification is pattern detection, where one tries to identify certain well-known, and often used, inter class communication patterns like call backs and the observer pattern [17]. In this paper, we present and evaluate a component identification method based on the graph theoretical concept *dominance.*

Dominance analysis applied to the system call graph is a well-known method within the imperative language community to identify modules [18,19]. To our knowledge, it has not been applied to object oriented systems before.

In Section 2, we present dominance analysis and related graph theoretical concepts. In Section 3, we give an overview of the identification process we have in mind and discuss necessary prerequisites. In Section 4, we show how dominance analysis can be used to identify components in object oriented systems. We also discuss the limitations of the method presented. In Section 5, we discuss how dominance analysis can be combined with other identification methods to recover a larger part of the architecture, and a novel approach to merge the results from different identification methods into a single data structure, a semi-lattice, that permits a unified view of the different results. Finally, in Section 6, we make a summary of our efforts and present future work that needs to be done.

2 Dominance Analysis

This section describes a number of graph theoretic concepts that are relevant to dominance analysis. Dominance analysis is a graph based technique to identify certain nodes in a directed graph. The use of dominance analysis as a method to identify related parts of a program was introduced by Cimitile and Visaggio [18]. They applied dominance analysis on call graphs derived from imperative systems to identify candidates for reusable modules. This idea has been further elaborated in [19,20]. Dominance analysis has also been used extensively in compiler optimization to identify loops in the basic block graph [21,22]. In this paper, we apply the same technique to identify components in object oriented systems. Most of the material in this section can be found in advanced compiler text books [21,23,22].

Dominance is a relation between nodes in directed graphs $G = (N, E)$, where N is a finite nonempty set of nodes and $E \subset N \times N$ is a set of edges. Furthermore, a *root node* of a directed graph is a node $r \in N$ with no incoming edges. A *rooted directed graph* $G_r = (N, E, r)$ is a directed graph (N, E) with a unique root node $r \in N$.

Dominance is a relation between nodes in rooted directed graph. This relation can formally be defined as:

Definition 2.1 Let $G_r = (N, E, r)$ be a rooted directed graph. We say that $a \in N$ **dominates** $b \in N$, written $a\ dom\ b$, iff every possible path from the root node r to b includes a. We say that a is a **direct** dominator of b, written $a\ ddom\ b$, iff $a \neq b \wedge a\ dom\ b$ and there does not exist a node $c\ (\neq a, b)$ such that $a\ dom\ c \wedge c\ dom\ b$. Finally, we say that a **strongly directly** dominates b, written $a\ sdom\ b$, iff $a\ ddom\ b$ and a is the only predecessor of b.

The dominance relation *dom* also induces a partial ordering of the nodes in the graph since *dom* is reflexive ($\forall a : a\ dom\ a$), transitive ($a\ dom\ b \wedge b\ dom\ c \Rightarrow$

a dom c), and antisymmetric ($a\ dom\ b \wedge b\ dom\ a \Rightarrow a = b$). Furthermore, the direct dominance relation identifies, for each single node, a single dominator from the collection of dominators of that node. This relation can be considered as a parent-child relation and hence, we have a tree, the dominance tree.

Definition 2.2 The **dominance tree** corresponding to a rooted directed graph $G_r = (N, E, r)$ is a graph $T_d = (N, E_d, r)$ where

$$E_d = \{(a, b) \in N \times N | a\ ddom\ b\}$$

In the right hand side of Figure 1 we show the dominance tree corresponding to the directed graph on the left. We use dotted edges to separate the strongly directly relation from the plain directly relation. We see here, for example, that 3 strongly directly dominates 4 since 3 is the only predecessor to 4 whereas 3 only directly dominates node 5 since node 5 has more than one predecessor.

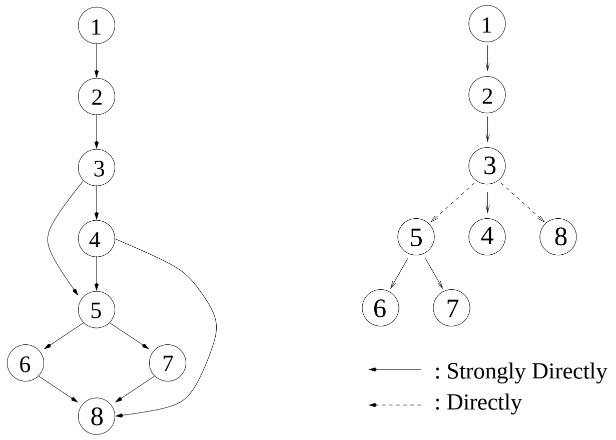

Fig. 1. A directed graph with corresponding dominance tree.

In Figure 1, we constructed a dominance tree starting from an acyclic graph. This is not a necessary condition. We can construct a dominance tree from every rooted directed graph.

Muchnick [21] presents two algorithms to compute the dominance tree corresponding to a rooted directed graph. The first one is quite simple and can be implemented with a running time that is $O(n^2e)$ where n is the number of nodes and e is the number of edges. The second algorithm is a slightly modified version of the one developed by Lengauer and Tarjan [24]. It can be implemented with a running time that is $O(n\alpha(n, e)$, where α is a very slowly growing function, but is much more complicated.

3 Prerequisites

3.1 Class Graphs

Previous works have applied dominance analysis on call graphs derived from systems written in some imperative language. We will apply dominance analysis on what we refer to as class interaction graphs derived from object-oriented systems.

Definition 3.1 A **class interaction graph** (or just **class graph**) is a rooted directed graph (N, E, r) where the set of nodes N consists of the classes used in the system. For any two classes $A, B \in N$, if A is a potential **user** of B then the edge (A, B) appears in the graph. The complete collection of edges is denoted E. The root class $r \in N$ is the class that contains the main method.

Notice that we have edges interconnecting classes if a class is a *potential user* of another class. The reason for this formulation is twofold:

(i) The term *use* is deliberately vague since we would like our definition of the class graph to be as flexible as possible. In many cases "A uses B" means that class A invokes one or more methods provided by class B. However, it can also mean that A accesses a public field of B, or that A and B interacts via more complicated communication patterns like call-backs or the observer pattern. In the latter case, the edge (A, B) indicates the main direction of the information flow. Hence, the term "A uses B" must be precisely defined for each instances of a class graph [3].

(ii) The word "potential" was included in the definition since we, in general, can not derive an exact class graph. There are basically three reasons for this: 1) We do not in general have exact control flow information. For example, the actual control flow path taken in if-statements is in general not known. 2) We can not in general resolve the polymorphic calls. 3) Reflection and dynamic class loading makes it is impossible for static analysis to determine which classes and methods may be accessed using these mechanisms. This problem is in some sense worse than the other two since we can not even make conservative approximations. Issues related to program analysis in the presence of these mechanism are discussed at great length in [25]. We handle the first two cases by using *conservative approximations*. That is, we include all the edges where there is a potential method call. We treat polymorphism in a similar way. A call $A.a_i$ from a class C will not only result in an edge (C, A) but also edges from C to all the subclasses of A that are possible targets of the call.

3.2 Class Graph Construction

A good starting point for class graph construction is a call graph. In an object oriented system, this means nodes of type $A.a_i$ where A is a class and a_i one of its methods. Every edge $(A.a_i, B.b_j)$ in the call graph corresponds to a potential call from within $A.a_i$ to a method $B.b_j$. In order to get a

[3] We intend to include inheritance relations by using delegations where a call is targeted to its defining class and delegated to it's subclasses. Experiments will show if this approach is feasible.

uniform presentation, we use the same notations for field accesses. That is, access to a field c_j of class C from within $A.a_i$ is represented by and edge $(A.a_i, C.c_j)$. In the examples that follows, we will also use capital letters to denote constructors. For example, the $i : th$ constructor of a class A is denoted as $A.A_i$.

On the left hand side of Figure 2, we see the call graph corresponding to a simple program implementing the observer pattern [26]. The class `Main` acts as a driver for the observed subject `IntBag` that stores a number of integers and the two observers `Adder` and `Printer` that get notified whenever new integers are added to `IntBag`. The class `ObjectArray` is a utility class that is used to store both integers and observers in `IntBag`.

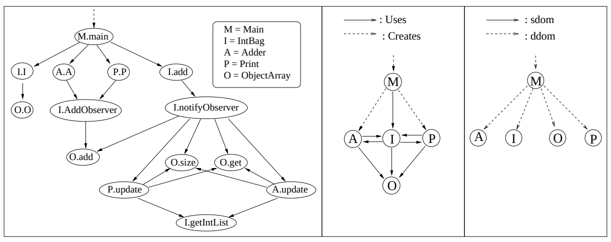

Fig. 2. (Left) The call graph corresponding to a simple implementation of the observer pattern. (Middle) The corresponding class graph. (Right) The corresponding dominance tree.

The actual class graph construction is straight forward once a call graph has been derived. It is done in a process we refer to as *folding*. Folding is a reduction of the call graph with respect to a partitioning $P = N_1, \ldots, N_k$, where the block N_i is all the members (fields and methods) defined in class C_i. Algorithmically, it can be expressed as:

(i) Collapse all nodes of the type $A.a_i$ into a single node A. Denote the set of new nodes N.

(ii) Add the edge (A, B) to E if there exist a_i and b_j such that $(A.a_i, B.b_j)$ was an edge in the call graph.

(iii) Let $r \in N$ be the node that contains the main function.

In the middle of Figure 2, we see the resulting class graph after we have folded the call graph on the left hand side. The dotted lines indicates that the only usage is creation. For example, the only interaction between the Main class (M) and the Adder class (A) is that M calls the constructor of A.

3.3 A Semi-Automatic Approach

Understanding the architecture of a legacy system is time consuming since such system is often only sparsely documented. The only trustworthy source of information is in such cases the system implementation. Hence, the architecture has to be retrieved from this source. As real world legacy system tend to be large, the source code cannot be read directly. Instead, we propose to use semi-automatic program analysis to extract the information. As these analysis are not unique by nature, system engineers have to be involved to accept or reject certain results proposed by the automatic analysis. Hence, the result of such analyses ought to be presented in a form that is intuitive to the system engineer. Therefore, program analyses must go hand in hand with interactive software visualization. In Figure 3, we show an overview of the process we have in mind. The starting point is the source code of the program about to be investigated.

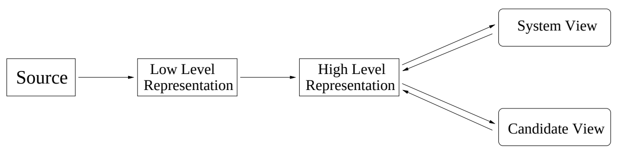

Fig. 3. Overview of the architecture recovery process.

What then follows is a series of abstractions where certain information is extracted from the source code. Implicitly, this means that we reduce the amount of information in order to achieve comprehension at the cost of accuracy. In our case, we extract the information needed to construct a call graph, which is our low level representation of the program. The low level representation contains information that we consider essential in order to recover the system architecture. However, for any non-trivial real-world program this structure is still much too complex to provide the system engineer with architectural insight. For example, a medium size program often results in a call graph with many thousand nodes, and about twice as many edges. In order to achieve an architectural understanding, we must therefore reduce the complexity of the low level representation by further abstractions. In our case, the previously described folding process is such an abstraction. The result, the class graph, is what we refer to as a high level representation of the program. It is in some sense the lowest comprehensible representation of a program, a representation that can be understood by humans within a reasonable amount of time. A visualization of the high level representation is a *system view* - a complete view of the system at hand. The program analysis up to this point is completely automated. The next phase, where classes and their interactions are grouped into components and connectors, requires human interactions.

In this paper, we present dominance analysis as method to identify components. We will see that dominance analysis is good at identifying certain types of components, but not sufficient to recover the complete architecture. A realistic architecture recovery tool would require the capability to use many different identification methods. (The use of multiple identification methods will be further discussed in section 5.) What these methods all have in common is that they provide the tool user (a system engineer) with a set of component and connector candidates. Each of these candidates is presented to the system engineer as a *candidate view*. The fewer, and more accurate, the better. The final decision whether these candidates should be accepted or not is left to the system engineer. The system is notified once a candidate is accepted, and both the high level representation and the system view gets updated. The class graph described earlier is a suitable high level representation for this scenario. If a few classes is accepted as a component, the class graph is updated by collapsing these classes into a single composite node. If a few use edges (and maybe some classes) are identified as a connector, they are replaced by an composite edge that represents the connector.

In summary, the semi-automated architecture recovery process we have in mind is iterative. A system engineer is fed with component and connector candidates, presented as candidate views, to be accepted or rejected. Each accepted entity results in an update of the system high level representation as well as the system view.

4 Dominance Analysis applied to Class Graphs

Dominance analysis applied to imperative systems start by collapsing strongly connected regions of the call graph since *"two or more procedures involved in recursive call relationships cannot be reused separately because they present an extremely high level of coupling . . . "* [18]. When dealing with object-oriented systems represented by class graphs, in general, the same argument does *not* hold. Take the observer pattern in Figure 2 as an example. Here the subject `IntBag` and the two observers (`Adder` and `Printer`) forms a strongly connected region. However, it should not be appropriate to collapse those classes into a component since most software designers that use the observer pattern want to establish a loose coupling between the subject and the observers. To start then by bundling those classes into a component would therefore, in many cases, be great abuse of the designers original intension. Furthermore, the observer pattern is only one example of class interaction patterns that result in strongly connected regions of the class graph. In reality, strongly connected regions in the class graph is very common. Every exchange of messages in between two classes results in a strongly connected region, and we can not convincingly argue that these two classes can not be reused separately. Hence, we do not start by collapsing all the strongly connected regions when applying dominance analysis to object-oriented systems represented by class graphs. On the left

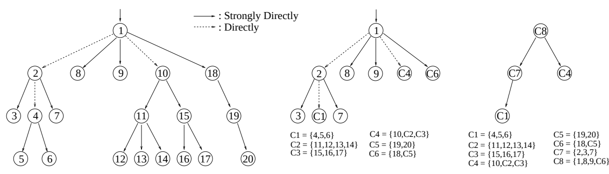

Fig. 4. (Left) A dominance tree derived from a hypothetical class graph. (Middle) The result after having collapsed some nodes using Rule 4.1. (Right) The result after having collapsed some nodes using Rule 4.2.

hand side of Figure 4 we see a dominance tree derived from a hypothetical class graph derived from some hypothetical object-oriented program. The nodes 1-

20 represents the classes and the edges represents their directly dominance relations. Node 1 is the class containing the main function. The strong direct dominates relation A *sdom* B is easy to understand in terms of classes uses each other. It simple means that class A is the only class that uses class B. The strong direct dominance relation thus captures a fundamental characteristic of a typical functional dependence between two classes in a software system: if a class B is used by the class A only, then B implements a sub-functionality of a more general functionality defined by A. Furthermore, B is an obvious candidate to be considered as a *part of* A and can without effecting the other classes be hidden within A. This idea can be extended to subtrees of the dominance tree that only consists of *sdom* edges. We therefore have our first candidate rule:

Rule 1 *Every subtree of a node N having only sdom edges is a component candidate. The subtree of N is a* ***part of*** *the component CN having node N as its interface.*

To group all classes that is part of a *sdom* subtree into a component (or at least a candidate component) makes sense since these classes constitutes a rather independent unit that only interacts with the rest of the program through the interface node N. This property makes them easy to understand and reuse. In the middle of Figure 4, we have collapsed all such components into a single node. In a more realistic scenario should these component candidates be presented visually to a system engineer that would have the final decision whether they should be considered components or not. They should be collapsed into a single node only if accepted. Notice also that collapsed components have the same type of dominance relation with their parent node as the root node of their *sdom* subtree.

An *sdom* subtree can be interpreted as a service provided to the rest of the program that is represented by its interface (root) node. In terms of services, the target of a *ddom* edge implements a service that is used by more than one class/component, and thus it may not be advisable to hide any target of a *ddom* edge in its dominator. On the other hand, the target of an *sdom* edge is only used by its dominating node, and can therefore be considered as a candidate to be hidden within its dominating node. If we look at the middle of Figure 4 we see that we still have *sdom* edges after all *sdom* subtrees have been collapsed. The remaining targets of *sdom* edges should not be considered as component candidates on their own, however, they can be considered as candidates to be hidden in their parent:

Rule 2 *The target of a sdom edge is a candidate to be a part of its dominating node.*

In the right hand side of Figure 4, we have hidden all the targets of *sdom* edges in their dominating parent. Notice also that we have denoted the remaining edges as *sdom* edges. For example, $C1$, previously the target of a *ddom* edge, has been changed to be the target of an *sdom*. The reason is that

$C1$, before we applied Rule 2, was used by two or more of the nodes 2, 3, and 7, and now, after nodes 3 and 7 been hidden in $C7$, is only used by the new composite node $C7$. Hence, contrary to Rule 1, we must recalculate the dominance relations after having applied Rule 2. (Alstrup and Lauridsen [27] describe a technique for incrementally updating a dominator tree.) We can now in principle reuse Rule 1 and collapse the whole dominance tree into a single node.

The result of applying these two rules to the hypothetical case presented in 4 looks very promising. We have reduced a system consisting of 20 classes to a set of only 4 components. If, however, we take a closer look at the two rules we see that both are based on the existence of what we refer to as *passive components*. A passive component is a set of classes having a single class as interface and which classes are not using any classes outside the component. In fact, every subtree of a dominance tree represents a passive component. The important questions that remains to be answered is: How often do passive components appear in real-life systems? In order to answer that question we must apply dominance analysis on a number of real systems. We can however immediately say that complex interaction patterns like for example the observer pattern, ruins this property. Taking a look at the right hand side of Figure 2, we can see the result of applying dominance analysis to a complex class graph where classes are exchanging messages in a non-trivial way. The result is a dominance tree with depth one where the root node is the only dominating node. This is a null result from which no conclusions can be drawn.

5 Using Multiple Identification Methods

In the previous section, we concluded that dominance analysis is a useful method to identify certain types of components (the passive components) but insufficient to recover the complete system architecture. This is a property that most identification methods (e.g.[14,15,16,19,18,17]) have in common. They are good at capturing certain specific types of components (or connectors) but unable to handle others. The obvious approach is therefore to combine different identification methods. In this section we will present an approach using a few different identification methods.

5.1 Complex Interaction Pattern Reduction

The major weakness of dominance analysis is that it can't handle complex class interaction where two classes are exchanging messages. For example, two classes interacting via callbacks or the observer pattern. This suggests that dominance analysis should be combined with communication pattern identification where frequently used communication patterns are identified and collapsed into one-directional composite connectors. In Figure 5, we illustrate this approach. The left hand side illustrates an exchange of messages between two classes A and B that forms an instance of the observer pattern. Heuzeroth *et al* show in [17] how a combination of static and dynamic analysis can be used to identify this kind of patterns. Once they have been identified, we can collapse them into a one-directional composite edge that reflects the direction of the major information flow. The advantage of identifying, and collapsing, a complex interaction pattern like the observer patterns is now twofold: 1) We can simplify the class graph and thus present the system engineer with a more comprehendible system view, 2) we remove the kind of complex class interaction that dominance analysis can't handle. Hence, we can expect the dominance analysis to be more efficient when a number of complex interaction patterns have been identified and reduced to a single composite edge.

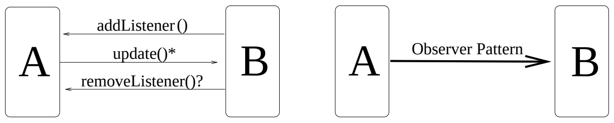

Fig. 5. (Left) An instance of the observer pattern identified in the class graph. (Right) The same situation after the observer pattern have been collapsed to a composite edge

5.2 Component Identification using Concept Analysis

Concept analysis is a mathematical technique that have been used frequently to identify modules in imperative systems (e.g. [28][14][15]). The basic idea, as outlined in [15], is to associate so-called features with different run-time scenarios generated by a profiler. Each part of the program used in the scenario is annotated with this feature. The outcome is a mapping (a table of binary relations) between scenarios and features upon which concept analysis can be applied to identify program parts jointly required by any subset of features. This approach will provide a functional decomposition of the program at hand given that features and functionality are related. We can see no reason why this approach shouldn't work for object oriented systems as well. (To our knowledge, component identification based on concept analysis has not been applied to object oriented systems before.) If each scenario attaches a feature to each class used in that scenario, we can use the same idea to generate component candidates - set of classes jointly required by subsets of features.

Once again, we have a twofold advantage in introducing a second component identification method: 1) Concept analysis can identify components that dominance analysis can't handle, 2) Concept analysis can *verify* components already identified by dominance analysis.

5.3 A Unified Approach

The advantage of having multiple entity identification methods is that they can be used in conjunction to reduce the size of the candidate sets, and as a complement, to identify entities other methods can't handle. Candidates identified by more than one method are more likely to be important, non-overlapping candidates generated by different methods is also interesting whereas overlapping (contradictive) candidates, at a first glance, can be ignored. Each of the three methods described above can be used to identify architectural entities in an object oriented system. Furthermore, they generate entity candidates that, if accepted by the system engineer, can be represented as reductions of the class graph. Thus, the class graph works as a unifying *system data structure* for the recovery process. The fact that we have a single structure to represent the system is essential if we want that results (accepted candidate entities) due to one identification method to be incorporated into the next analysis.

However, it is somewhat disturbing that each analysis creates its own candidate view. First, it makes the decision process for the systems engineer more complicated as more candidate views are to observe. It could easily lead to a situation where it is unclear how to handle the (potentially contradicting) information from the different analyses. Second, it make the integration of a new analysis difficult. It required an implementation of the analysis itself, a data structure to capture its results and a visualization of that data structure.

Hence, the results of the different analyses should be unified. In order to do that, it is essential to identify a common data structure for analysis results, a structure where we can merge the results from different sources, and that permits a uniform presentation for the user. In this section, we will present a novel approach to merge different entity candidates resulting from different analyses into a single data structure, a semi-lattice over the power set of system classes.

In Figure 6, we show an overview of the process we have in mind. The starting point is the source code of the program about to be investigated.

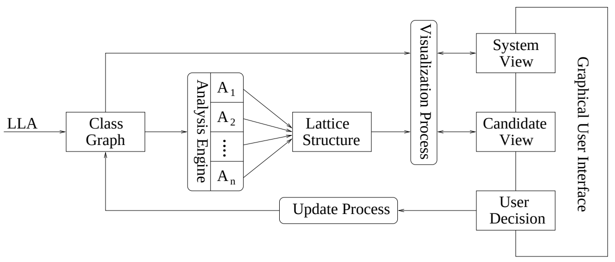

Fig. 6. Overview of the architecture recovery process using many different analysis methods.

The first phase of the analysis, the low level analysis (LLA) leading to the initial class graph, without any composite nodes and edges, is done without any human intervention. (The class graph construction was described in section 3.2. A call graph, constructed using some low level analysis, is folded into a class graph.) The next phase, the entity identification, requires input from a system engineer. As depicted in Figure 6, all the results derived using various analysis methods, $A_1, A_2, ..., A_n$, are merged into a single *lattice structure* from which a single candidate view is presented. The candidate view is in turn connected to the system view. The idea is that possible system decompositions presented in the candidate view should also be indicated in the system view. The effect of a certain system decomposition induced in the candidate view by the user should immediately be displayed in the system view. Once a decision been made by the user, e.g. to collapse some nodes, the information is feed back to the analysis engine via an updated class graph, and the whole process can restart. That is, we propose an iterative process where user input triggers new analyses, resulting in updated data to be displayed. In general, we distinguish component from connector analyses.

Component Analyses
Each component identification method propose candidate sets $\sigma_i \in \mathcal{P}(C)$, where $\mathcal{P}(C)$ is the power set of all system classes $\mathcal{C}$, or partitions $P = \{P_1, \ldots, P_n\}$ of the class set $\mathcal{C}$ where $P_i \in \mathcal{P}(C)$, or a hierarchy thereof (as the notion of components is recursive, a component may contain other components). The fact that each analysis produce results that can be seen as elements of $\mathcal{P}(C)$ makes it possible to represent all data they produce as elements of a semi-lattice $L = (\subseteq, \mathcal{P}(C))$ with $\top = \mathcal{C}$. Each individual analysis contributes to the construction of L:

- each candidate set σ_i becomes an element of L.
- for each partitioning P, each individual $P_i \in P$ becomes an element of L.
- for each cluster of candidate sets, the set of contained candidate sets becomes an element of L.

The constructed lattice L is now a single data structure containing all information from the different analyses. Note, that the order relation between the individual elements in the L is the subset relation over the contained classes. Hence, a hierarchy of components as potentially proposed by a component analysis, is captured by L, as well. In Figure 7 we show a lattice corresponding to the candidate sets produced by dominance analysis in Figure 4. We see

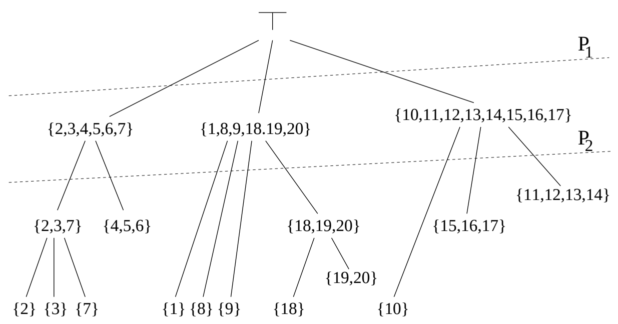

Fig. 7. A lattice representation of the analysis results earlier presented in Figure 4. The figure also includes two possible partitions of the system. Each one corresponding to a decomposition of the system classes.

that each proposed candidate set is represented as a lattice element, and that the partial ordering that comes with the lattice, corresponds to hierarchical components. It is now straight forward to add results from other identification methods as long as their results can be seen as elements of $\mathcal{P}(C)$. New proposed candidate sets will result in new lattice elements. Thus, by using lattices we can present the result of multiple component identification methods in a single view, based on a single data structure.

The lattice approach also displays synergy information, e.g. partitions of the system cross-cutting the results proposed by individual analyses. As an individual analysis could be strong in detecting one type of components, but weak for others, synergetically proposed partitions could be even more promising. The fact that each horizontal cut of a lattice represents a partition of the system is a property that can be used to visualize various decompositions of the system. In Figure 7, we show two such partitions. P_1 represents a high level decomposition of the system containing only three components. Partition P_2 represents a more fine grained decomposition involving components with fewer classes, some of which are singletons.

Furthermore, if an analysis cannot detect components in a certain situation, it would either create no element of L at all, or singleton sets containing individual classes of the system. Hence, weak results would not accumulate and deteriorate views based on L. Another advantage is that L is the only data structure to visualize and relate to the system view. Hence, the visualization (routine) does not change with new analyses added to the framework.

The use of different identification methods will in general result in candidate sets that overlap each other. How to interpret these sets is not obvious and will probably require some further analysis from the user. (Possible candidates are the individual sets, their union, and their intersection.) The overlapping sets will in the lattice be represented (and recognized) as elements found close to each other with no ordering relation.

Finally, we would like to capture if a candidate sets is supported by more than one analysis. Therefore, we use an extended lattice $L = (\sqsubseteq, \mathcal{P}(C) \times N)$ with $(C,i) \sqsubseteq (C',j) \Leftrightarrow C \subseteq C'$. The construction is almost identical: whenever a new candidate set C is proposed, $(C,1)$ is inserted into the lattice. If C already exists as $(C,i) \in L$, then (C,i) is replaced by $(C,i+1)$. This "weight" can then be used by the system engineer to make a better decision whether a component candidate should be accepted or not. This idea can be extended to other preferences, e.g. to give credits for partitions steaming from a single analysis.

Connector Analyses

To include connector analysis results into the lattice approach described above is somewhat tricky since methods interconnecting the classes are not part of the lattice structure. Our approach is therefore somewhat implicit, we only regard the clustering of classes participating in the communication. More formally, we propose three *clustering categories* $(\mathcal{S}, \mathcal{R}, I)$ where $\mathcal{S} \subseteq \mathcal{P}(C)$ is a set of candidate sets $\{S_1, S_2, .., S_n\}$ representing the various senders, $\mathcal{R} \subseteq \mathcal{P}(C)$

is a set of candidate sets $\{R_1, R_2, .., R_m\}$ representing the various receivers, and I is a single candidate set representing the connector component itself. I may be empty, indicating a component entirely consisting of method interactions. We use clusters (sets of candidate sets) to represent the sender, and receiver side, since many interaction patterns induces a many-to-many relation between senders and receivers. An example of a one-to-many relation is the observer pattern where we frequently have many observers listening to one observable. In our notation, each observer is represented by a candidate set R_i, the observable by a candidate set S, and the interaction I by the empty set. We believe that this approach is general enough to handle most communication patterns used today.

Once a triple like $(\mathcal{S}, \mathcal{R}, I)$ has been identified, it can be incorporated into the lattice structure. Each cluster, a set of classes, forms an element of $\mathcal{P}(C)$, and can therefore be positioned in the lattice.

6 Summary and Future Work

We have used dominance analysis to identify possible software components in an object oriented system. The actual dominance analysis is applied on a high level representation of the system that we refer to as the class graph – a directed graph where nodes are the system's classes and edges class interactions. It can easily be obtained from the system call graph.

The result is that dominance analysis can identify what we refer to as passive components. (A passive component is a set of classes having a single class as interface and which classes are not using any classes outside the component.) Components that interact with their surrounding using more complex interaction patterns are not identified. Hence, dominance analysis can not be used to recover the complete architecture of the system at hand, it must be combined with other identification methods.

We discuss how dominance analysis can be used together with other entity identification methods (connector identification based on pattern detection, and component identification based on concept analysis). We also present a novel approach to merge the results from different identification methods. Our approach is based on a common candidate data structure, a semi-lattice over the power set of all system classes. This approach not only makes it possible to uniformly handle different identification methods, it also brings forward synergy information that would be hard to detect using separate structures for each identification method. For example, partitions of the system classes cross-cutting different identification methods.

To fully evaluate dominance analysis as method to identify components in object oriented systems, we must investigate how often passive components appear in real-life systems. In order to answer that question, we must apply dominance analysis on a number of real systems. A work that we recently have started and that will be reported elsewhere.

References

[1] Elliot J. Chikofsky and James H. Cross II, "Design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, January 1990.

[2] Michael L. Nelson, "A survey of reverse engineering and program comprehension," .

[3] Hausi A. Muller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne D. Storey, Scott R. Tilley, and Kenny Wong, "Reverse engineering: a roadmap," in *ICSE - Future of SE Track*, 2000, pp. 47–60.

[4] Rainer Koschke, *Atomic Architectural Component Recovery for Program Understanding and Evolution*, Ph.D. thesis, Institute for Computer Science, University of Stuttgart, 2000.

[5] Rick Kazman and S. Jeromy Carrière, "Playing detective: Reconstructing software architecture from available evidence," *Automated Software Engineering: An International Journal*, vol. 6, no. 2, pp. 107–138, April 1999.

[6] R. Kazman, S. G. Woods, and S. J. Carriere, "Requirements for integrating software architecture and reengineering models: Corum II," in *Working Conference on Reverse Engineering (WCRE'98)*, October 1998.

[7] Barry W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[8] B. Lientz, E. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, June 1978.

[9] R. K. Fjeldstad and W. T. Hamlen, "Application program maintenance study: Report to our respondents," in *Proceedings GUIDE 48, Philadelphia*, April 1983.

[10] D. Garlan and M. Shaw, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[11] Dewayne E. Perry and Alexander L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[12] T. Systa, P. Yu, and H. Muller, "Analyzing java software by combining metrics and program visualization," in *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'2000)*, February 2000.

[13] S. R. Chidamber and C. F. Kemerer, "A Metric Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, june 1994.

[14] Michael Siff and Thomas Reps, "Identifying modules via concept analysis," in *Proc. of. the Internation Conference on Software Maintenance.* 1997, pp. 170–179, IEEE Computer Society Press.

[15] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon, "Aiding program comprehension by static and dynamic feature analysis," in *Proceedings of the International Conference on Software Maintenance.* November 2001, IEEE Computer Society Press.

[16] Gregor Snelting, "Software reengineering based on concept lattices," in *CSMR*, 2000, pp. 3–10.

[17] D. Heuzeroth, T Holl, and W Löwe, "Combining static and dynamic analyses to detect interaction patterns," in *Proceedings of the Sixth International Conference on Integrated Design and Process Technology*, 2002.

[18] A. Cimitile and G. Visaggio, "Software salvaging and call dominance tree," *The Journal of Systems and Software*, vol. 28, no. 2, pp. 117–127, february 1992.

[19] Jean-Francois Girard and Rainer Koschke, "Finding components in a hierarchy of modules - a step towards architectural understanding," in *Proceedings of the International Conference on Software Maintenance 1997.* 1997, IEEE Computer Society Press.

[20] Simon C Shaw and Michael Goldstein, "Moralising the call graph as a means of program comprehension," Tech. Rep., Department of Mathematical Sciences, University of Durham, 2002.

[21] S. S. Muchnick, *Advanced Compiler Design Implementation*, Morgan Kaufmann Publishers, San Francisco, California, 1997.

[22] M.S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.

[23] W. M. Waite and G. Goos, *Compiler Construction*, Texts and Monographs in Computer Science. Springer-Verlag, 1984.

[24] Thomas Lengauer and Robert E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM TOPLAS*, vol. 1, no. 1, pp. 121 –141, July 1979.

[25] Peter F. Sweeney and Frank Tip, "Extracting library-based object-oriented applications," in *Proceedings of the Eighth International Symposium on the Foundations of Software Engineering (FSE-8)*, November 2000, pp. 98 –107.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[27] Stephen Alstrup and Peter W. Lauridsen, "A simple dynamic algorithm for maintaining a dominator tree," Tech. Rep., No 96/3, Department of Computer Science, University of Copenhagen, Denmark, 1996.

[28] C. Lindig and G. Snelting, "Assessing modular structure of legacy code based on mathematical concept analysis," in *Proceedings of the International Conference on Software Engineering*, 1997, pp. 349 – 359.