Moral Dominance Relations for Program Comprehension

Simon C. Shaw, Michael Goldstein, Malcolm Munro, and Elizabeth Burd, *Member*, *IEEE*

Abstract—Dominance trees have been used as a means for reengineering legacy systems into potential reuse candidates. The dominance relation suggests the reuse candidates which are identified by strongly directly dominated subtrees. We review the approach and illustrate how the dominance tree may fail to show the relationship between the strongly directly dominated procedures and the directly dominated procedures. We introduce a relation of generalized conditional independence which strengthens the argument for the adoption of the potential reuse candidates suggested by the dominance tree and explains their relationship with the directly dominated vertices. This leads to an improved dominance tree, the moral dominance tree, which helps aid program comprehension available from the tree. The generalized conditional independence relation also identifies potential reuse candidates that are missed by the dominance relation.

Index Terms—Directed graphical model, generalized conditional independence, dominance tree, program comprehension, reengineering, reuse candidate, reverse engineering, testing.

1 Introduction

For many companies, software drives the business and provides the only true description of their operations. As businesses evolve, so should the software. Thus, it is necessary to perform software maintenance, "the modification of software products after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment," [18]. Program comprehension, in this setting, involves acquiring knowledge about programs, as well as any remaining documentation and operating procedures. We aim both to understand the software through visualization and to identify areas of the code which may be remodularized as a means of aiding maintenance by localizing the impacts of change. Further, these identified modules are potential reuse candidates.

In this paper, we are concerned with functional abstractions through aggregation based upon the calling structure of a piece of code. To restrict the discussion to a manageable length, we thus consider only persistent data such as files or tables and regard the program as consisting of a database, D, and a collection of procedures which may be called and which operate on the database. The database is viewed as encoding the state of the program. We shall expand upon this description in Section 3. The separate modules we discuss are coarse-grained persistent objects as opposed to fine-grained volatile objects. As Cimitile et al. [11] point out, this is more appropriate when considering potential reuse; they also provide a good review of work on fine-grained volatile objects.

The potential reuse candidates are obtained from subtrees on the dominance tree, an abstraction of the calling structure. Various authors [6], [7], [9], [10], [11], [12] have worked on the identification of potential reuse candidates from the dominance tree and, in Section 2, we review the techniques. Müller et al. ([23]) have also worked on the identification of reuse candidates. Canfora et al. ([8]) talk of the balance between "the ability to simply partition a legacy system into objects versus the ability to abstract an architecture (i.e., relations between objects)." In the examples in Section 2, we highlight some limitations of the dominance tree in its failure to both develop an architecture and, in certain simple cases, failure to highlight potential reuse candidates. In Section 3, we introduce an alternative relation on the call graph, that of generalized conditional independence (g.c.i.). In Section 4, we show how the adoption of the g.c.i. relation gives a formal underpinning for the selection of reuse candidates from a modified form of the dominance tree. The theory is illustrated by a series of simple examples.

2 Program Comprehension Using Dominance Trees

In this section, we review current approaches to program comprehension using the dominance tree derived from the call graph. Through a series of examples, we illustrate the methodology and suggest a number of potential problems with the current practice. In Sections 3 and 4, we introduce a formal approach, based around the g.c.i. relation, to address these problems.

- *S.C. Shaw and M. Goldstein are with the Department of Mathematical Sciences, University of Durham, Science Laboratories, South Road, Durham, DH1 3LE, UK.*
E-mail: {S.C.Shaw, Michael.Goldstein}@durham.ac.uk.
- *M. Munro and E. Burd are with the Research Institute in Software Evolution, Department of Computer Science, University of Durham, Science Laboratories, South Road, Durham, DH1 3LE, UK.*
E-mail: {malcolm.munro, Liz.Burd}@durham.ac.uk.

Manuscript received 19 Dec. 2001; revised 8 Oct. 2002; accepted 27 May 2003.
Recommended for acceptance by G. Canfora.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 115590.

0098-5589/03/$17.00 © 2003 IEEE Published by the IEEE Computer Society

2.1 The Call Graph

The call structure of a piece of code provides a high-level description of the flow of the program. This structure describes the procedural units and the relationships between them. Examples of procedural units are functions in the language C, paragraphs in COBOL, and methods in Java. In this paper, we use the generic term "procedure" and the relationships between procedures are termed "calls." No attempt is made to introduce other relationships that may, for example, enable data flow analysis. The calling structure may be visualized by presenting it as a graph. We follow the notation and terminology of Lauritzen [20] so that a graph is a pair $\mathcal{G} = (V, E)$, where V is a finite set of vertices and E is the set of edges, a subset of $V \times V$ the set of ordered pairs of distinct vertices. An edge $(f, g) \in E$ is directed, denoted $f \rightarrow g$, if $(f, g) \in E \wedge (g, f) \notin E$. We term f a parent of g and g a child of f. The full collection of parents of g are denoted by $pa_{\mathcal{G}}(g)$ while $ch_{\mathcal{G}}(f)$ denotes the full collection of children of f. An edge $(f, g) \in E$ is undirected, denoted $f \sim g$, if $(f, g) \in E \wedge (g, f) \in E$. If $(f, g) \notin E$, we write $f \not\rightarrow g$ and if $(f, g) \notin E \wedge (g, f) \notin E$, we write $f \not\sim g$. If all edges on the graph are directed then the graph is said to be directed, whereas it is undirected if all edges are undirected. A path of length n from f to g is a sequence $f = f_0,\ f_1, \ldots, f_n = g$ such that $(f_{i-1}, f_i) \in E\ \forall i = 1, \ldots, n$. We write $f \mapsto g$. If both $f \mapsto g$ and $g \mapsto f$, then f and g are said to connect and we write $f \rightleftharpoons g$. If either $f \mapsto g$ or $g \mapsto f$, we state that there is a direct path between f and g. There is an undirected path between f and g if there is a sequence $f = f_0, f_1, \ldots, f_n = g$ such that either $(f_{i-1}, f_i) \in E$ or $(f_i, f_{i-1}) \in E$ for all $i = 1, \ldots, n$. A directed graph is weakly connected if there is an undirected path between any pair of vertices and strongly connected if there is a directed path between every pair of vertices. If $f \mapsto g$ and $g \not\mapsto f$, then f is an ancestor of g and g a descendent of f. The complete collection of ancestors of g are denoted by $an_{\mathcal{G}}(g)$ while the complete collection of descendents of f are denoted by $de_{\mathcal{G}}(f)$. An n-cycle is a path of length n from f to itself. If the graph $\mathcal{G}$ contains no cycles, then it is said to be acyclic. An acyclic connected undirected graph is termed a tree; a rooted tree is a directed acyclic graph (DAG) obtained from a tree by choosing a vertex as a root and directing all edges away from this vertex. For $V_S \subseteq V$, we may obtain the subgraph $\mathcal{G}_S = (V_S, E_S)$, where E_S is obtained from $\mathcal{G}$ by keeping the edges with both endpoints in V_S. A cycle thus generates a strongly connected subgraph.

Definition 1. *A call graph is a directed graph, $\mathcal{G}_C = (V_C, E_C)$. The finite set of vertices, V_C, consists of the procedures which are either called or call other procedures in the program. For any two procedures $f, g \in V$ if there is a call to g by f, then the edge (f, g) appears on the graph. The complete collection of edges is denoted by E_C.*

Some languages permit direct or indirect recursion and so the call graph may be cyclical. As Cimitile and Visaggio [12] highlight, "the existence of recursions among procedures is in fact indicative of the implementation of a functionality through a recursive algorithm" and suggest that two or more procedures in such a recursive call relationship exhibit a high level of coupling and may be considered as a single module. By collapsing every strongly connected subgraph into a single vertex, we may convert the call graph into a DAG. This remodularization will also simplify the visualization by reducing the number of vertices and edges without damaging the architecture of the system, as calls from and to the procedures in the cycle are maintained on the modified call graph. We proceed by assuming that such a remodularization has been performed on the call graphs we consider, so that we deal only with DAGs. Notice that, by virtue of containing no cycles, DAGs must have at least one vertex that has no parents: If $\mathcal{G} = (V, E)$ is a DAG, then there exists $f \in V$ such that for all $g \in V, (g, f) \notin E$. Thus, in terms of the call graph, f is a procedure which is not called by any other procedure. Such procedures are often called entry points; in this paper, we term them root nodes.

Fig. 1a shows an example of a very simple call graph; it has a single root node $A000$. It is straightforward to understand the calling structure of the program. For example, once $D000$ has been called, the execution of the program exists purely in the collection $D^* = \{D000, D100, D200, D110\}$ until $D000$ is exited. Similarly, once $B000$ has been called, execution exists purely in the collection $B^* = \{B000, B100, B200\}$ until $B000$ is exited. Notice that, following a call to $C000$, execution does not exist purely in the collection $C^* = \{C000, C100, C200, C110\}$ since $C000$ calls $B000$ and, so, execution may switch to B^*. However, execution cannot switch to D^*. Observe that, by removing the procedure $A000$ and the three calls it makes from the call graph, we are left with a subgraph that consists of two disconnected subgraphs, $B^* \cup C^*$ and D^*. Intuitively, it seems that B^* and D^* can be considered as reuse candidates with D^* being accessed by $A000$ and B^* by $A000$ and $C000$. A more formal approach is required to both strengthen the intuitive argument for this example and to handle call graphs with many thousands of procedures and calls. The most familiar approach is to make a further abstraction of the call structure by converting the call graph into a rooted tree using the dominance relation; the rooted tree is termed the dominance tree.

2.2 The Dominance Tree

The dominance tree aims to assist program comprehension by reducing information overload during the early stages of comprehension and by identifying collections of procedures which may be remodularized into single modules. The dominance tree is a rooted tree whose root is a root node of the call graph $\mathcal{G}_C = (V_C, E_C)$ and is constructed using the relations of direct dominance and strong direct dominance, [17].

Definition 2. *If $f \in V_C$ is a root node of the call graph $\mathcal{G}_C$ and $de_C(f)$ the descendents of f on $\mathcal{G}_C$, then we construct the subgraph $\mathcal{G}_{C_f} = (f^*, E_{C_f})$, where $f^* = \{f\} \cup de_C(f)$. For procedures $g, h \in f^*$, g dominates h on $\mathcal{G}_{C_f}$ if and only if every path $f \mapsto h$ on $\mathcal{G}_{C_f}$ intersects g. We say that g directly*

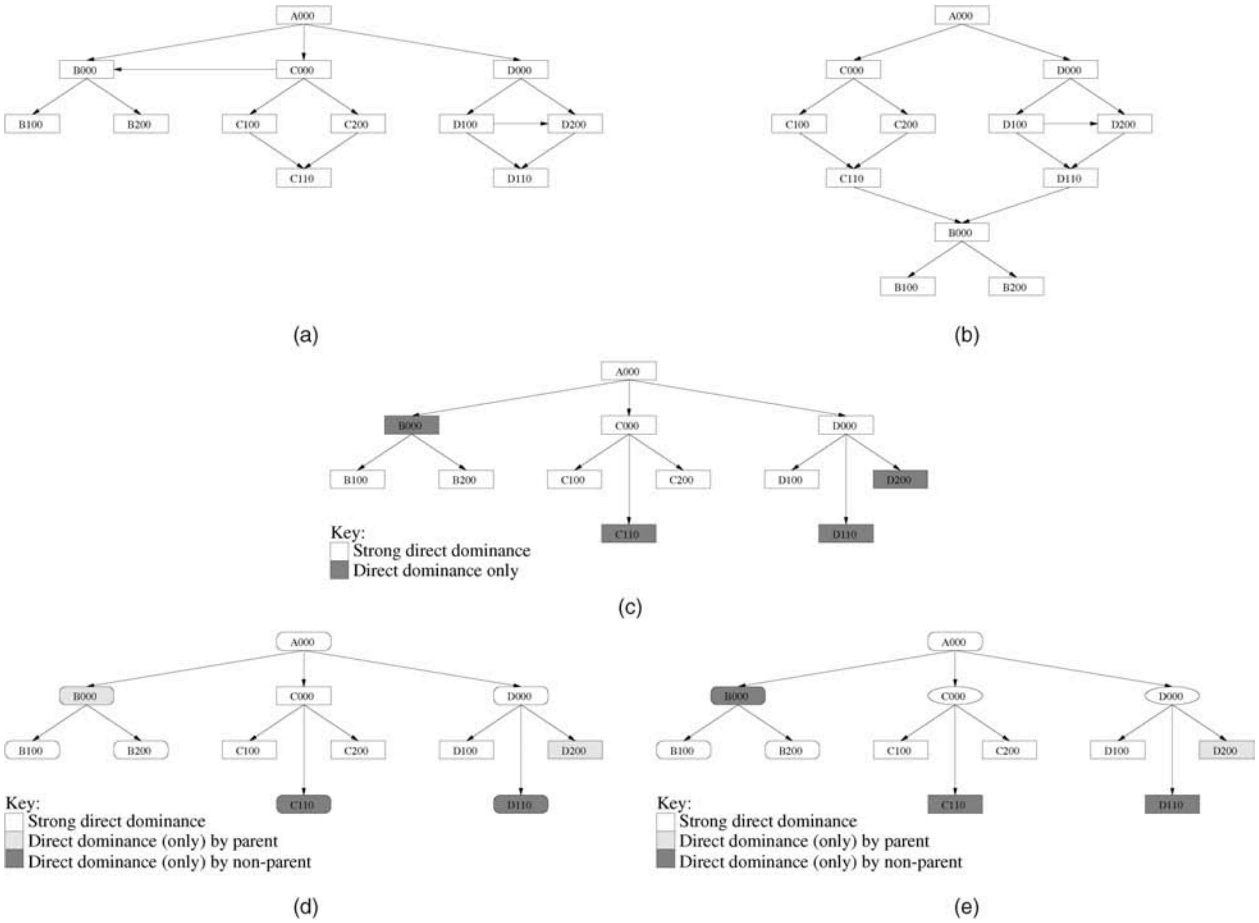

Fig. 1. (a) A simple call graph. Procedure $A000$ calls procedures $B000$, $C000$, and $D000$. Procedure $C000$ calls procedures $B000$, $C100$, and $C200$, and so on. (b) A second simple call graph. (c) The dominance tree corresponding to both the call graph in (a) and that in (b). (d) The moral dominance tree resulting from the call graph of (a). (e) The moral dominance tree resulting from the call graph of (b).

dominates h on $\mathcal{G}_{C_f}$ if and only if all procedures that dominate h on $\mathcal{G}_{C_f}$ dominate g on $\mathcal{G}_{C_f}$. g strongly directly dominates h on $\mathcal{G}_{C_f}$ if and only if g directly dominates h on $\mathcal{G}_{C_f}$ and is the only procedure in f^* that calls h.

The root node, f, trivially dominates all procedures $h \in de_C(f)$ and the direct dominance relation identifies, for each h, a single procedure from the collection of dominators of h.

Definition 3. *The dominance tree corresponding to a root node, f, is the graph $\mathcal{G}_{\mathcal{D}_f} = (f^*, E_{\mathcal{D}_f})$ formed from $\mathcal{G}_{C_f} = (f^*, E_{C_f})$, the subgraph of the call graph $\mathcal{G}_C = (V_C, E_C)$, where $f^* = \{f\} \cup de_C(f)$ and $E_{\mathcal{D}_f} = \{(g,h) \forall g, h \in f^* : g$ directly dominates h on $\mathcal{G}_{C_f}\}$. The vertex h is shaded if g only directly dominates h on $\mathcal{G}_{C_f}$.*

Each root node of $\mathcal{G}_C = (V_C, E_C)$ will generate its own dominance tree. If $\mathcal{G}_C$ has a single root node, f, then $f^* = V_C$ and $\mathcal{G}_{C_f} = \mathcal{G}_C$ and we denote the dominance tree by $\mathcal{G}_\mathcal{D} = (V_C, E_\mathcal{D})$. The call graph in Fig. 1a has a single root node and the sole corresponding dominance tree, $\mathcal{G}_\mathcal{D}$, is shown in Fig. 1c. Vertices which are not strongly directly dominated are shaded. For example, $D000$ strongly directly dominates $D100$, while $D110$ is only directly dominated by $D000$. Procedures that are only directly dominated have become disinherited from some, possibly all, of the procedures which called them: They had at least two calling vertices and may not be directly dominated by any of them. In Fig. 1c, $D110$ is directly dominated by $D000$, but $D000$ does not call $D110$ in Fig. 1a. Thus, $E_\mathcal{D} \not\subseteq E_C$: The dominance tree is not merely the call graph with some edges removed. Procedures which are only directly dominated indicate a more complex relationship in the call graph than that shown on the dominance tree and so information is lost in the abstraction from the call graph to the dominance tree at the shaded vertices. Intuitively, the greater the proportion of shaded vertices, the more problematic program comprehension may be from the dominance tree.

One benefit of the dominance tree to program comprehension is that it reduces the complexity of the visualization of the call graph and the layout is straightforward. In commercial applications, see, for example [6], procedures have been found to call or be called by over 100 procedures and so a reduction to a single edge on the dominance tree greatly increases the readability of the graphic. It is with such large complex problems in mind that, in Definition 3, we follow the convention of, for example, [2], [3], [4], [5] in distinguishing between strong direct dominance and direct dominance by vertex shading as opposed to the dashed and solid edged approach adopted in, for example, [9], [12], [11], as this improves the visual representation. As an illustration of this, the dominance tree in Fig. 2 of [2] contains 230 procedures and it is easy to identify whether subtrees within the dominance tree contain shaded vertices whereas the scale would make it hard to distinguish between dashed and solid edges.

In addition to visualization, dominance trees have been used to identify potential reuse candidates within the code which may then be reengineered into separate modules. This modularization helps make the code more flexible and maintainable. Burd and Munro [4] write that "the directly dominates and strongly directly dominates relations define where remodularization can occur. For instance, where directly dominates relations are identified, this means that calls are made to other vertices within the branch of the tree." For example, on the dominance tree in Fig. 1c, $C110$ is only directly dominated which indicates that it must be called by at least one of $C100$ or $C200$. We do not know whether $C000$ calls it. From the call graph in Fig. 1a, we can confirm that both $C100$ and $C200$ call $C110$ and $C000$ does not.

2.3 "Single Call In" and "Multiple Calls In" Subtrees

For a dominance tree $\mathcal{G}_{\mathcal{D}_f}$ and any procedure $h \in V_{\mathcal{D}_f}$, we are interested in the subtree consisting of the collection of procedures $h^\bullet = \{h\} \cup de_{\mathcal{D}_f}(h)$, the procedure h and all of its descendents on $\mathcal{G}_{\mathcal{D}_f}$. The dominance relation means that the only calls from $V_{\mathcal{D}_f} \setminus h^\bullet$ to $h^\bullet$ on the call graph $\mathcal{G}_C$ are to h itself. If h is strongly directly dominated by $g \in V_{\mathcal{D}_f} \setminus h^\bullet$, then this is the only call to h from the procedures in $V_{\mathcal{D}_f} \setminus h^\bullet$ on the call graph and this call may be deduced from $\mathcal{G}_{\mathcal{D}_f}$; $\mathcal{G}_{\mathcal{D}_f}$ illustrates how $h^\bullet$ is accessed by the other procedures, $V_{\mathcal{D}_f} \setminus h^\bullet$. Consequently, we introduce the term "single call in" subtree to describe $h^\bullet$.

If, however, h is only directly dominated by $g \in V_{\mathcal{D}_f} \setminus h^\bullet$, then the limit of the information given by the dominance tree is that we only know there is a call from at least one procedure, $\tilde{g} \in de_{\mathcal{D}_f}(g) \setminus h^\bullet$, to h on the call graph. We do not know whether g itself calls h or indeed how many such $\tilde{g}$ there are. In this case, we introduce the term "multiple calls in" subtree to describe $h^\bullet$. "Multiple calls in" subtrees lead to a lack of uniqueness of the dominance tree: Different call graphs lead to the same dominance tree. For example, the same tree is produced by setting $pa_{C_f}(h) = g^\bullet \setminus h^\bullet$ rather than the actuality of $pa_{C_f}(h) \subseteq g^\bullet \setminus h^\bullet$. The lack of uniqueness becomes more apparent the larger $g^\bullet \setminus h^\bullet$ is. Without recourse to the call graph, this could lead to difficulties in identifying reuse candidates and assessing the impact of change by mapping ripple effects, the changes that become necessary to make due to maintenance on another part of the code.

As an illustration, consider the call graph in Fig. 1b. The corresponding dominance tree is identical to the dominance tree for the call graph in Fig. 1a. $B000$ is only directly dominated by $A000$ and, so, any call graph corresponding to the dominance tree in Fig. 1c must contain at least one call from $C000^\bullet \cup D000^\bullet$ to $B000$. For the call graph in Fig. 1a, there is a solitary procedure, $C000 \in C000^\bullet \cup D000^\bullet$, while, in the call graph in Fig. 1b, there are two procedures, $C110$ and $D110$, in $C000^\bullet \cup D000^\bullet$ which call $B000$. Notice that $A000$ does not call $B000$ on the call graph in Fig. 1b. The two call graphs have very different structure; in particular, notice that for the call graph in Fig. 1b, $B000^\bullet$ is contained in the set of descendents for every other procedure and, so, a change to $B000$ could ripple through all of $C000^\bullet$ and $D000^\bullet$. Even for these very simple examples, a comprehension based purely on the shared dominance tree in Fig. 1c will fail to capture the differences between the two call graphs.

Current practice is to use the subtrees which we term "single call in" subtrees as the basis for identifying potential reuse candidates. Consider a dominance tree, $\mathcal{G}_{\mathcal{D}_f}$, where the procedure g strongly directly dominates procedures $h_1, \ldots, h_n$ and $\cup_{i=1}^{n} = ch_{\mathcal{D}_f}(g)$, the children of g on $\mathcal{G}_{\mathcal{D}_f}$. Burd and Munro [3] identify the individual subtrees $h_i^\bullet$ for each $i = 1, \ldots, n$ as potential reuse candidates. There are no calls between the $h_i^\bullet$ on the call graph $\mathcal{G}_C$. Once execution enters $h_i^\bullet$, it cannot switch to any other procedure $\tilde{g} \notin h_i^\bullet$ until h_i is exited. If, however, there exists a procedure h_{n+1} such that h_{n+1} is only directly dominated by g on $\mathcal{G}_{\mathcal{D}_f}$, then there is at least one $h_j^\bullet$ such that execution can switch from $h_j^\bullet$ to $h_{n+1}^\bullet$. This is the case with the dominance tree in Fig. 1c. $C000^\bullet$, $D000^\bullet$ are "single call in" subtrees strongly directly dominated by $A000$. Execution cannot switch between $C000^\bullet$ and $D000^\bullet$. However, $B000^\bullet$ is a "multiple calls in" subtree directly dominated by $A000$ and, so, execution could switch from either $C000^\bullet$ to $B000^\bullet$ or from $D000^\bullet$ to $B000^\bullet$. For the call graph in Fig. 1a, it is $C000^\bullet$ to $B000^\bullet$ while, for that in Fig. 1b, it is both $C000^\bullet$ to $B000^\bullet$ and $D000^\bullet$ to $B000^\bullet$. For the call graph in Fig. 1a, see Section 2.1, we might suggest the reuse candidates to be $B000^\bullet \cup C000^\bullet$ and $D000^\bullet$ with $B000^\bullet$ a separate module within the module $B000^\bullet \cup C000^\bullet$. This isolation is not apparent on the dominance tree. For the call graph in Fig. 1b, we might argue that $B000^\bullet$ was a separate module used by the modules $C000^\bullet$ and $D000^\bullet$. While the dominance tree, Fig. 1c, identifies that, for both call graphs (in Figs. 1a and 1b), execution cannot switch between $C000^\bullet$ and $D000^\bullet$, there is no automated way we can link these collections correctly with $B000^\bullet$: The relationship differs in the two call graphs. As Burd and Munro [4] point out, "this represents a failure to properly isolate candidates at an appropriate level of granularity." In addition to the relationship between "single call in" branches and "multiple calls in" branches, differing relations between the "single call in" branches are not visible on the call graph. For example, consider the "single call in" branches $C000^\bullet$ and $D000^\bullet$ on the dominance tree in Fig. 1c. $C000$ and $D000$ have no shared descendents on the call graph in Fig. 1a, while for that in Fig. 1b, the set $B000^\bullet$ is contained in the set of descendents of both $C000$ and $D000$. We would like a means of recognising whether, and how, the branches on the dominance tree are related.

2.4 Further Problems with Dominance Trees

The problem of multiple root nodes. The dominance relation is determined from a specific root node of the call graph. When a call graph has multiple root nodes, multiple dominance trees must be generated and the same procedures may appear on different dominance trees. Burd and Munro [4, Section 4] found this problem in case studies of

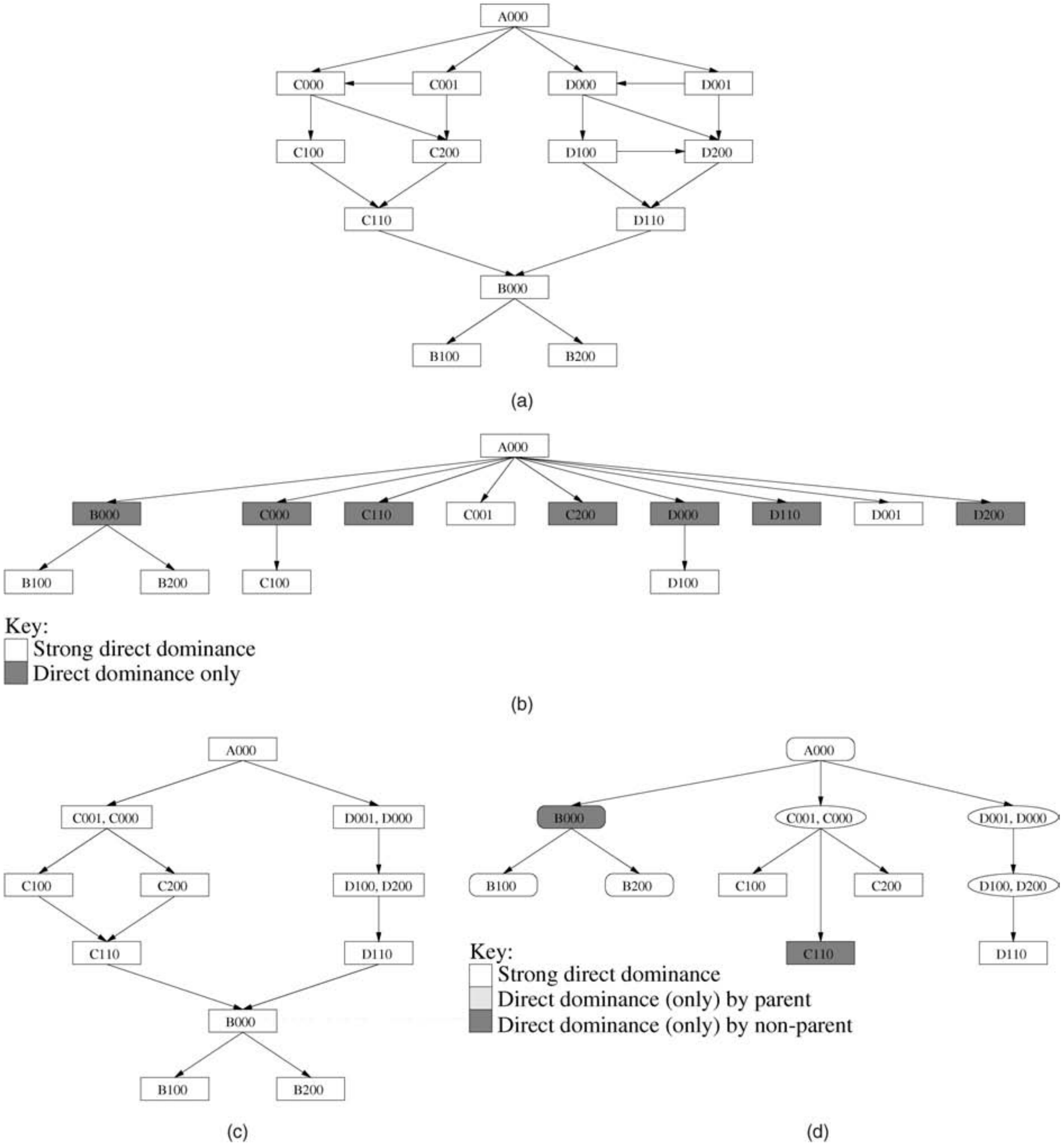

Fig. 2. (a) A third call graph. (b) The corresponding dominance tree. (c) The modified call graph obtained by merging vertices in the cliques on the call graph in (a). (d) The moral dominance tree corresponding to the modified call graph.

C code. They write that "within the case studies, the largest number of dominance trees identified from a single code file was 41 ... The fact that multiple dominance trees are generated can be problematic if procedures are shared between individual dominance trees. In all cases identified through the case study, this was found to be the case." It is not clear how to assess the relationship between multiple dominance trees.

Failure to capture potential reuse candidates. Consider the call graph in Fig. 2a. Its structure is similar to that of Fig. 1b and we could argue that the collections $C^* = \{C001, C000, C100, C200, C110\}$ and $D^*=\{D001, D000, D100, D200, D110\}$ form two separate modules which access the module $B^* = \{B000, B100, B200\}$. The corresponding dominance tree is shown in Fig. 2b. Only B^* appears as a subtree on the dominance tree while both C^* and D^* collapse: Of the 10 procedures in $C^* \cup D^*$, only four are strongly directly dominated. Excluding $A000^\bullet$, the "single call in" subtrees all contain a single procedure. For this example, the dominance tree provides a poor representation of the call graph and is not helpful for program comprehension.

The dominance relation, see Definition 2, may be viewed as a graph separation property. For a graph $\mathcal{G} = (V, E)$, a subset $G \subseteq V$ is said to be an $(f, h)\text{-separator}$ if all paths from f to g intersect G. Thus, for a directed graph, for each path $f \mapsto h$, there exists $g \in G$ such that $f \in an_{\mathcal{G}}(g)$ and $h \in de_{\mathcal{G}}(g)$. If F, G, and H are nonoverlapping subsets of V, then G is said to separate F from H if G is an

(f, h)-separator for every $f \in F$, $h \in H$. Thus, g dominates h on $\mathcal{G}_{C_f}$ if and only if g separates f from h on $\mathcal{G}_{C_f}$, that is, g is an (f, h)-separator. The separation property may be applied to any three collections of vertices whereas the dominance relation applies to only three vertices, one of which is a root node and, in particular, only a single vertex is considered for the separator set. For the call graph in Fig. 2a, it is this search for a single vertex that causes the collections C^*, D^* to collapse for $A000$ is separated from $\{C100, C200, C110\}$ by $\{C001, C000\}$, while $\{D001, D000\}$ separates $A000$ from $\{C100, C200, C110\}$. By considering separations of individual vertices rather than collections of vertices on the original call graph, we may fail to identify potential reuse candidates and comprehend the calling structure. Indeed, by lifting the restriction of graph separation away from a root node, we can identify the differences between the call graphs in Figs. 1a and 1b. In the former, $C000^\bullet$ and $D000^\bullet$ are separated by the empty set; in the latter, $C000^\bullet$ and $D000^\bullet$ are separated by $B000$. This discussion suggests that an approach based upon more general graph separations rather than the dominance relation will be a more fruitful approach to program comprehension and we now develop such an approach.

3 g.c.i. Representations for the Calling Structure

In this section, we show how we can formalize the relationship between any collection of procedures on the call graph. To achieve this, we use the calculus of g.c.i. properties, a relation on triples of uncertain quantities, W, X, Y, which identifies whether given information on W, information on X has a bearing on the uncertainty about Y.

3.1 Procedures and Uncertainty in the Calling Structure

For simplicity of exposition, we regard a piece of code as consisting of a database, D, which encodes the state of the program (for example, the variables), and a collection of procedures which may be called and which operate on the database. Having been called, each procedure is viewed as processing an input in order to perform an action. Upon completion of the action, control of the program returns to the procedure which made the call. For example, the action may be to read an item in the database or to write to the database. The result of the action is, thus, dependent upon the state of the database immediately prior to the call being made.

To formalize this, let $\mathcal{G}_C = (V_C, E_C)$ be a call graph with $g, h \in V_C$. Suppose a call is made to h by g with input a and that, immediately prior to this call, the state of the database is D_a. The expected outcome, having processed the input, is that the result of the action is $h_{D_a}(a)$ and the state of the database is $D_{h(a)}$. However, we actually observe $\tilde{h}_{D_a}(a)$ and $D_{\tilde{h}(a)}$. There is uncertainty as to whether the procedure has performed the action correctly, that is, whether $\tilde{h}_{D_a}(a) = h_{D_a}(a)$, and also whether the database has been left in the desired state, that is, whether $D_{\tilde{h}(a)} = D_{h(a)}$.

Definition 4. *The procedure h is said to work for input a if, for all possible database states, D_a, we have $\tilde{h}_{D_a}(a) = h_{D_a}(a)$ and $D_{\tilde{h}(a)} = D_{h(a)}$. If the two conditions do not both hold, then the procedure h is said to be in error for a.*

The error is specific to the procedure; the procedure should be able to cope with any given state of the database and any given input. For instance, if a procedure fails to leave the database in the desired state, a later procedure accessing the database will not be in error if it can handle this error. As an example, suppose a piece of code operates an accounts system for a bank. Procedure h has the function of adding a given amount, x, to a specific account y (so the input is $a = (x, y)$). It does this by adding x to every account. h performs its action correctly but does not leave the database in the desired state. h is in error for a. If procedure i is now called to read the amount in account z, although the incorrect amount is present in z, i will not be in error if it correctly reads the amount present (and leaves the database unchanged).

Definition 5. *The procedure h is said to work if it works for each input $a \in A$, where A is the set of possible inputs. If there is an input a such that the procedure h is in error for a, then the procedure is said to not work.*

Definition 5 allows us to consider the potential propagation of errors. Suppose that $\mathcal{G}_C = (V_C, E_C)$ is a call graph with $f, g, h \in V_C$ and $(f, g), (g, h) \in E_C$. Suppose g is called by f with input $\tilde{a}$ and that, in order to process this action, g calls h with input a. If h is in error for a, then an error is present when control is returned to g and, thus, when control returns from g to f: g is in error for $\tilde{a}$. The error has propagated from h to g and then to f (as the return from f will contain an error). The dependence on the database state immediately prior to the call being made (see Definition 4) means that the only way for errors to propagate is in the reverse order to calls on the call graph. In general, the error may only propagate from, on the call graph, child to parent: It depends upon whether the parent calls the child with an input for which the child is in error. This propagation of errors, in the reverse order to the calls, leads us to make the following definition.

Definition 6. *For a call graph $\mathcal{G}_C = (V_C, E_C)$ with $g, h \in V_C$, we construct the error propagation graph $\tilde{\mathcal{G}}_C = (V_C, E_R)$, where $E_R = \{(h, g) : (g, h) \in E_C\}$.*

$\tilde{\mathcal{G}}_C$ is thus the call graph with the edges reversed and it maps the potential propagation of errors, as defined by Definition 5. Notice that, while we talk here about error propagation, we are interested in actions where a change in the child on the call graph could cause a change in the parent. We view ripple effects as being such an action. Each procedure may be viewed as a random quantity having two possible states: 1 if the procedure works and 0 if the procedure does not work.

Viewing the procedures in this way enables us to formalize a relationship between the procedures. We may consider whether learning the state of a given procedure is informative about the state of another procedure of interest. This may help us distinguish between call graphs where the dominance tree fails to detect a difference. Consider the call graphs in Figs. 1a and 1b and suppose that we learn that $C000$ does not work. In both call graphs, this could have been caused by an error propagating from $B000$ and, so, we would now believe the chance of $B000$ not working to be greater than before. There is a relationship between $B000$ and $C000$ in both cases, but this is not shown on the dominance tree, Fig. 1c. In Fig. 1b, an error in $B000$ could propagate to $D^* = \{D000, D100, D200, D110\}$ and, so, learning about $C000$ influences D^*. However, in Fig. 1a, this is not the case. Notice that, if we already knew the state of $B000$ for the call graph in Fig. 1b, say that it worked, then learning that $C000$ does not work no longer provides any information about D^* as we know the error in $C000$ has not propagated from $B000$. Knowledge of the state of $B000$ separates the uncertainty, in terms of whether the procedures are working, between $C000$ and D^*. We may represent such separations using the concept of g.c.i., as we now explain.

3.2 Generalized Conditional Independence Property

Smith [25], [26] defined a tertiary relation, $(\cdot \perp\!\!\!\perp \cdot) | \cdot$, on all triples W, X, Y of uncertain quantities, that is quantities whose state is currently unknown to us, as follows:

Definition 7. *Any tertiary relation* $(\cdot \perp\!\!\!\perp \cdot) | \cdot$ *satisfying the following three properties:*

1. $(W \perp\!\!\!\perp X) | X \cup Y,$ (1)
2. $(W \perp\!\!\!\perp X) | Y$ *if and only if* $(X \perp\!\!\!\perp W) | Y,$ (2)
3. $(W \perp\!\!\!\perp X \cup Y) | Z$ *if and only if* $\begin{cases} (W \perp\!\!\!\perp Y) | Z; \\ (W \perp\!\!\!\perp X) | Y \cup Z, \end{cases}$ (3)

for any collections W, X, Y, Z of uncertain quantities is termed a generalized conditional independence (g.c.i.) property. $(W \perp\!\!\!\perp X)|Y$ is read as "W is independent of X given Y."

Equation (1) is that "once X is known (along with anything else Y), then no further information can be gained about X by observing W." Equation (2) is the symmetry relation: "If once Y is known, W is uninformative for X, then X is uninformative for W, having observed Y." Equation (3) is "if having observed Z, W is uninformative for both X and Y, then equivalently, having observed Z, W is uninformative about Y and, having observed Y and Z, W conveys no information about X." $W \perp\!\!\!\perp X$ is a shorthand for $(W \perp\!\!\!\perp X)|\emptyset$, where $\emptyset$ is the empty set.

The most familiar g.c.i. property is when the collections represent random quantities and $(\cdot \perp\!\!\!\perp \cdot) | \cdot$ is taken to be probabilistic conditional independence. For random vectors X, Y, Z, we say that X is probabilistically conditionally independent of Y given Z, written $(X \perp\!\!\!\perp Y)|Z$, if $p(x, y|z) = p(x|z)p(y|z)$, or equivalently if $p(x|y, z) = p(x|z)$, where $p(\cdot)$ denotes the probability density function; if $Z = \emptyset$, then we say that X and Y are probabilistically independent. Dawid [14], [15] developed probabilistic conditional independence as a basic intuitive concept with its own axioms. The work of Smith is a generalization of this to other inference systems which do not require full probability specifications. For example, Goldstein [16] constructs a tertiary property satisfying properties (1), (2), and (3) based on the partial quantitative specification of beliefs. Smith [25] writes that "in a Bayesian statistical or decision analysis it is common to be told that, given certain information W, a variable X will have no bearing on another Y. It is often quite easy to ascertain this type of information from a client for various combinations of variables. Such information can be gathered before it is necessary to quantify subjective probabilities which, in contrast, are often very difficult to elicit with any degree of accuracy." Pearl [24] agrees, arguing that "the notions of relevance and dependence are far more basic to human reasoning than the numerical values attached to probability judgements." By asserting properties (1), (2), and (3), the g.c.i. relation may be applied qualitatively without the need for explicit numerical specifications. The easiest way to do this is to represent the g.c.i. relation graphically using a directed graph as we now explain.

3.3 Belief Separation via the Moral Graph

A collection of g.c.i. relations may be represented graphically. The vertices of the graph are random quantities; vertices are joined by directed arrows if there is a possible direct dependency between them.

Definition 8. *A DAG, $\mathcal{G} = (V, E)$, is a directed graphical model (DGM) if, for any $X_i \in V$ and any $X_j \notin de_{\mathcal{G}}(X_i)$, the descendents of X_i on $\mathcal{G}$, we have*

$$(X_i \perp\!\!\!\perp X_j)|pa_{\mathcal{G}}(X_i), \qquad (4)$$

where $(\cdot \perp\!\!\!\perp \cdot) | \cdot$ is a g.c.i. property and $pa_{\mathcal{G}}(X_i)$ denotes the set of parents of X_i on $\mathcal{G}$.

There are a number of equivalent definitions of a DGM, for example, see Theorem 5.14 of [13]. The most familiar type of DGM, the Bayesian graphical model (BGM), occurs when $(\cdot \perp\!\!\!\perp \cdot) | \cdot$ represents probabilistic conditional independence. [29] introduces BGMs into the problem of software testing, while [1] show how fault trees can be mapped into BGMs.

Definition 8 shows that a DGM may be formed by the explicit statement of the parent sets for each vertex. However, the given g.c.i. statements are not the sole g.c.i. statements in the model because we may use properties (1), (2), and (3) to determine further g.c.i. statements. Indeed, to fully understand the g.c.i. structure of the model, we would like to be able to ask whether, for any three subsets W_1, W_2, $W_3 \subseteq V$ on the DGM, we have $(W_1 \perp\!\!\!\perp W_2)|W_3$. The answer lies by linking g.c.i. with graph separation on an associated undirected graph; graph separation satisfies properties (1), (2), and (3) (see Pearl [24, Section 3.1]) and so itself acts as a g.c.i. property. The required graph is the moral graph associated with W_1, W_2, W_3 as introduced by Lauritzen and Spiegelhalter [22].

Definition 9. *For the DAG $\mathcal{G} = (V, E)$ and any three subsets W_1, W_2, $W_3 \subseteq V$, the moral graph associated with W_1, W_2,*

W_3 is the undirected graph $\mathcal{G}_M(\cup_{i=1}^3 W_i) = (V_M(\cup_{i=1}^3 W_i), E_M(\cup_{i=1}^3 W_i))$, where

$$V_M(\cup_{i=1}^3 W_i) = \cup_{i=1}^3 \{W_i \cup an_{\mathcal{G}}(W_i)\};$$
$$E_M(\cup_{i=1}^3 W_i) = \{\{f \sim g \forall f, g \in V_M(\cup_{i=1}^3 W_i) : (f,g) \in E\}$$
$$\cup \{f \sim h \forall f, g, h \in V_M(\cup_{i=1}^3 W_i) : \{(f,g),(h,g)\}$$
$$\subseteq E \wedge (f,h),(h,f) \notin E\}\},$$

and $an_{\mathcal{G}}(W_i)$ denotes the collection of ancestors of W_i on $\mathcal{G}$. If $V_M(\cup_{i=1}^3 W_i) = V$, then we write $\mathcal{G}_M(\cup_{i=1}^3 W_i) = \mathcal{G}_M$ and term this the full moral graph.

Less formally, we construct the subgraph of $\mathcal{G}$ whose vertices are W_1, W_2, W_3 and all of their ancestors. For each individual vertex of the subgraph, we "marry" all of its parents (join them with an edge if not already joined) and then drop all arrows to form the moral graph $\mathcal{G}_M(\cup_{i=1}^3 W_i)$. Since all the parents are "married," Lauritzen and Spiegelhalter [22] coined the term moral graph. The following theorem, see [21], [27], shows that it is straightforward to use $\mathcal{G}_M(\cup_{i=1}^3 W_i)$ to determine whether $(W_1 \perp\!\!\!\perp W_2)|W_3$ on $\mathcal{G}$.

Theorem 1. *For any three subsets W_1, W_2, $W_3 \subseteq V$ on a DGM, $\mathcal{G} = (V, E)$, we have $(W_1 \perp\!\!\!\perp W_2)|W_3$ whenever W_1 and W_2 are separated by W_3 on $\mathcal{G}_M(\cup_{i=1}^3 W_i)$, the moral graph associated with W_1, W_2, W_3.*

Theorem 1 is often termed the moralization criterion. An alternative process to establish whether any three subsets W_1, W_2, W_3 satisfy $(W_1 \perp\!\!\!\perp W_2)|W_3$ on a DGM, using the concept of d-separation on the original DGM, was developed by Pearl [24]. Lauritzen et al. [21] shows this approach is equivalent to Theorem 1. The aim of Pearl [24, p. 81] as to "whether assertions equivalent to those made about probabilistic dependencies can be derived *logically* without reference to numerical quantities," may be met using Theorem 1. If W_1 and W_2 are separated by W_3 on $\mathcal{G}_M(\cup_{i=1}^3 W_i)$, then they are separated by any g.c.i. property that quantifies the network, for example, probabilistic conditional independence. For a collection of quantities of interest, V, we may assert a DGM over V and identify the independence structure of the model via Theorem 1. If we then wish to specify a full probability distribution over V, any distribution satisfying (4) will have the same independence structure irrespective of the actual numerical specifications. Such distributions are easy to find: If (4) is to hold, then the joint distribution over all the random quantities in V has the form $p(x_1, \ldots, x_n) = \prod_{i=1}^n p(x_i | pa_{\mathcal{G}}(x_i))$ (see, for example, Jensen [19, p. 20]).

3.4 Using the Call Graph to Create a DGM

In this section, we shall argue that the error propagation graph, see Definition 6, may be viewed as a DGM. First, we remark that the call graph, with the procedures viewed as the random quantities expressing whether the procedure works or not, does not constitute a DGM. Consider the call graph with vertices f, g, h and edges (f,g) and (f,h). Suppose that f is known not to be working. This could have resulted from an error propagating from either g or h or from an error in f itself. If we now learn that g works, then this will increase the belief that h is in error; procedures g and h are dependent given f. This violates property (4) which requires $(g \perp\!\!\!\perp h)|f$.

We now argue that property (4) is met on the error propagation graph. We consider vertices $g, h \in V_C$ and show that $(g \perp\!\!\!\perp h)|pa_{\tilde{\mathcal{G}}_C}(g)$ whenever $h \notin de_{\tilde{\mathcal{G}}_C}(g)$. From (1), this is immediate if $h \in pa_{\tilde{\mathcal{G}}_C}(g)$. Consider $h \in an_{\tilde{\mathcal{G}}_C}(g) \setminus pa_{\tilde{\mathcal{G}}_C}(g)$. h and g are dependent: An error in h can propagate from h to g. Any path $h \mapsto g$ on $\tilde{\mathcal{G}}_C$ must intersect some $\tilde{g} \in pa_{\tilde{\mathcal{G}}_C}(g)$. If the state of each $\tilde{g}$ is known, then knowledge of the state of h is uninformative for g. For example, if each $\tilde{g}$ is observed to work and we now observe that h is in error, then this gives us no new information: We already know from the state of $\tilde{g}$ that the error does not propagate to $pa_{\tilde{\mathcal{G}}_C}(g)$ and, hence, cannot propagate to g.

We now restrict h to the collection $V_C \setminus \{\{g\} \cup an_{\tilde{\mathcal{G}}_C}(g) \cup de_{\tilde{\mathcal{G}}_C}(g)\}$ and let $\mathcal{A} = an_{\tilde{\mathcal{G}}_C}(g) \cap an_{\tilde{\mathcal{G}}_C}(h)$. If $\mathcal{A} = \emptyset$, then g and h are independent. For example, learning that h does not work increases our belief that errors are contained in $an_{\tilde{\mathcal{G}}_C}(h)$, but errors in $an_{\tilde{\mathcal{G}}_C}(h)$ cannot propagate to g. Note that, letting $\mathcal{B} = de_{\tilde{\mathcal{G}}_C}(g) \cap de_{\tilde{\mathcal{G}}_C}(h)$, we may have $\mathcal{B} \neq \emptyset$. Consider some $\tilde{g} \in \mathcal{B}$. Learning about h is informative about $\tilde{g}$, but this does not influence g as although a potential error in h could result in $\tilde{g}$ calling g with the wrong input, or with the wrong database configuration, all that is relevant is whether g copes with these correctly. If $pa_{\tilde{\mathcal{G}}_C}(g)$ is now known, g and h remain independent since, as $\mathcal{A} = \emptyset$, $an_{\tilde{\mathcal{G}}_C}(pa_{\tilde{\mathcal{G}}_C}(g)) \cap an_{\tilde{\mathcal{G}}_C}(h) = \emptyset$.

If $\mathcal{A} \neq \emptyset$, then g and h are dependent. For example, if we learn that h does not work, then this error could have propagated from procedures contained in $\mathcal{A}$, increasing our belief for the procedures in $\mathcal{A}$ not working. Errors in $\mathcal{A}$ may also propagate to g increasing our belief in g not working. The dependence is via the collection $\mathcal{A} \subseteq an_{\tilde{\mathcal{G}}_C}(g)$. Arguing similarly to when $g \in an_{\tilde{\mathcal{G}}_C}(g)$ shows that, if $pa_{\tilde{\mathcal{G}}_C}(g)$ is known, information about $\mathcal{A}$ is irrelevant to g and tracing the passage of knowledge from h to g we see that knowledge about h is now irrelevant to g. This discussion is summarized as follows:

Lemma 1. *The error propagation graph $\tilde{\mathcal{G}}_C = (V_C, E_R)$ is a DGM. If G, H, F are three sets of procedures on $\mathcal{G}$ and F separates G from H on $\tilde{\mathcal{G}}_M(G, H, F)$, then $(G \perp\!\!\!\perp H)|F$.*

We term $\tilde{\mathcal{G}}_M(G, H, F)$ the associated moral graph to the call graph for collections G, H, F. For the call graph in Fig. 1a, the associated moral graph $\tilde{\mathcal{G}}_M(V_C \setminus A000)$ is obtained by deleting $A000$ and the three calls it makes in Fig. 1a, adding the edges $B100 \sim B200$ (as they are unmarried parents of $B000$ on the error propagation graph) and $C100 \sim C200$ and then dropping all arrows. As was intimated in Section 2.1, $B^* \cup C^*$ are separated from D^* by the empty set and, so, $B^* \cup C^* \perp\!\!\!\perp D^*$. We now explore the g.c.i. properties of the error propagation graph to strengthen our analysis of the dominance tree.

4 Program Comprehension Using the g.c.i. Relation

4.1 Strongly Directly Dominated Subtrees: "Single Call In"

In Section 2, we reviewed current practice in using the dominance tree to select potential reuse candidates. "Single Call In" subtrees, that is, subtrees whose root was strongly directly dominated, have been identified as possible sites of remodularization. In this section, we formalize the relationship between "single call in" subtrees which share a common strongly direct dominator. This enables us to strengthen the argument for the adoption of the reuse candidates, while also helping to explain the relationship of the candidates with the directly dominated vertices. The approach utilizes the g.c.i. relation discussed in Section 3. We have the following theorem; the proof is in the Appendix.

Theorem 2. *Suppose $\mathcal{G}_C = (V_C, E_C)$ is a call graph and consider any collection of vertices $h_1, h_2, \dots, h_m$ with the property that, for any $h_i \neq h_j$, there is no direct path between h_i and h_j on $\mathcal{G}_C$. Let $\mathcal{H} = \cup_{k=1}^{l}\{de_C(h_i) \cap de_C(h_{j_k})\}$. For any $i \in \{1, \dots, m\}$ and any $1 \leq l \leq m$, $1 \leq j_1 \leq j_2 \leq \dots \leq j_l \leq m$, $j_k \neq i$, with $h^* = \{h\} \cup de_C(h)$ for any $h \in V_C$,*

$$(h_i^* \perp\!\!\!\perp \cup_{k=1}^{l} h_{j_k}^*)|\mathcal{H}. \tag{5}$$

If two vertices are both strongly directly dominated by the same vertex on a dominance tree, then there is no path between them on a call graph. Moreover, with $h^\bullet = \{h\} \cup de_{\mathcal{D}_f}(h)$, $h^\bullet \subseteq h^*$ and, so, we may link Theorem 2 to the reuse candidates generated by the "single call in" subtrees on a dominance tree $\mathcal{G}_{\mathcal{D}_f}$ via the following corollary.

Corollary 1. *Suppose $\mathcal{G}_C = (V_C, E_C)$ is a call graph and f is a root node of $\mathcal{G}_C$. Additionally, consider a vertex g on $\mathcal{G}_{\mathcal{D}_f}$ which strongly directly dominates the vertices $h_1, \dots h_m$ on $\mathcal{G}_{C_f}$. Let $\mathcal{H} = \cup_{k=1}^{l}\{de_C(h_i) \cap de_C(h_{j_k})\}$. For any $i \in \{1, \dots, m\}$ and any $1 \leq l \leq m$, $1 \leq j_1 \leq j_2 \leq \dots \leq j_l \leq m$, $j_k \neq i$, with $h^\bullet = \{h\} \cup de_{\mathcal{D}_f}(h)$ for any $h \in V_{\mathcal{D}_f}$,*

$$(h_i^\bullet \perp\!\!\!\perp \cup_{k=1}^{l} h_{j_k}^\bullet)|\mathcal{H}. \tag{6}$$

Proof. Since $h_i^\bullet \setminus h_i$ forms the set of vertices dominated by h_i on $\mathcal{G}_{\mathcal{D}_f}$, then $\{h_i^\bullet \setminus h_i\} \subseteq de_C(h_i)$. The reduction of the sets h_i^* to $h_i^\bullet$ follows by the g.c.i. property (3). □

Corollary 1 thus provides us with the relationship between "single call in" subtrees whose roots share the same parent on the dominance tree. The subtrees are independent if they do not share any descendents on the call graph, so that $\mathcal{H} = \emptyset$, and conditionally independent if they do share descendents, that is $\mathcal{H} \neq \emptyset$. These shared descendents are thus present in "multiple calls in" subtrees on the dominance tree and, so, provide a connection between "single call in" and "multiple calls in" subtrees. Consider the dominance tree in Fig. 1c and the "single call in" subtrees $C000^\bullet$ and $D000^\bullet$. In the call graph in Fig. 1a, $C000$ and $D000$ do not share descendents and so, from Corollary 1, we have $C000^\bullet \perp\!\!\!\perp D000^\bullet$. However, for the call graph in Fig. 1b, $C000$ and $D000$ do share descendents: $B000^*$. In this case, applying Corollary 1 yields $(C000^\bullet \perp\!\!\!\perp D000^\bullet)|B000^*$. The difference in these two statements illustrates an advantage of the g.c.i. relation over the dominance relation and helps explain why we can capture the relationship between subtrees on the dominance tree. The dominance relation is only concerned with calls to a procedure while the g.c.i. relation also takes account of calls made by a procedure. This difference is crucial if one wishes to examine ripple effects as Figs. 1a and 1b illustrate.

4.2 Relations around "Isolated" Subtrees

Theorem 2 and, thus, Corollary 1 stress the importance of shared descendents of "single call in" subtrees. For the call graph in Fig. 1a, since $C000^* = B000^* \cup C000^\bullet$ and $D000^* = D000^\bullet$, then, from Theorem 2, $B000^* \cup C000^\bullet \perp\!\!\!\perp D000^\bullet$. In this case, $D000$ dominates all of its descendents and, hence, does not call any other subtree on the dominance tree.

If, for any $h_u \in V_{\mathcal{D}_f}$, we have $de_C(h_u) = de_{\mathcal{D}_f}(h_u)$, then the subtree $h_u^\bullet = h_u^*$ does not call any other subtree on $\mathcal{G}_{\mathcal{D}_f}$. We introduce the term "isolated" subtree to describe $h_u^\bullet$. In terms of the call graph, this means that, once h_u has been called, execution remains solely in the subtree $h_u^\bullet$ until h_u is exited. This suggests that we may wish to consider this subtree as a single unit. Notice that this may include subtrees where multiple procedures call the root h_u. We shall term subtrees that make calls to other subtrees on the dominance tree "nonisolated" subtrees. We have the following theorem; the proof is in the Appendix.

Theorem 3. *Suppose that $\mathcal{G}_C = (V_C, E_C)$ is a call graph and $h_u \in V_C$ is such that $de_C(h_u) = de_{\mathcal{D}_f}(h_u)$ for some dominance tree $\mathcal{G}_{\mathcal{D}_f} = (V_{\mathcal{D}_f}, E_{\mathcal{D}_f})$. If $g_1, \dots, g_m$ are any collection of vertices on $\mathcal{G}_{\mathcal{D}_f}$ with the property that, for each i, there is no direct path between each g_i and h_u on $\mathcal{G}$, then*

$$h_u^\bullet \perp\!\!\!\perp \cup_{i=1}^{m} g_i^*. \tag{7}$$

If $\{g_1, \dots, g_m\} \subseteq \{an_C(h_u) \cap V_{\mathcal{D}_f}\}$, with $G^\dagger \setminus h_u^\dagger = \{\cup_{i=1}^{m} g_i^*\} \setminus h_u^\bullet$, then

$$(de_{\mathcal{D}_f}(h_u) \perp\!\!\!\perp \{G^\dagger \setminus h_u^\dagger\})|h_u. \tag{8}$$

Theorem 3 shows us that "isolated" subtrees are either independent or conditionally independent of the other vertices on the call graph. It is irrelevant whether the root of the "isolated" subtree is strongly directly or just directly dominated on the dominance tree. Note that, on the dominance tree in Fig. 1c, $B000^\bullet = B000^*$ for both the call graph in Fig. 1a and that in Fig. 1b. For Fig. 1a, we have that $D000^\bullet = D000^*$ and, so, from relation (7), $B000^\bullet \perp\!\!\!\perp D000^\bullet$. $C000$ is an ancestor of $B000$ on the call graph and, from relation (8), we find that $(\{B100, B200\} \perp\!\!\!\perp C000^\bullet)|B000$. On the call graph in Fig. 1b, both $C000$ and $D000$ are ancestors of $B000$ and, so, from relation (8), we find that $(\{B100, B200\} \perp\!\!\!\perp C000^\bullet \cup D000^\bullet)|B000$.

Assessing whether $de_{\mathcal{D}_f}(h_u) = de_C(h_u)$ is simple; identifying these subtrees on the dominance tree enables it to provide more information about the calling structure without altering its layout. First, there will be a reduction in the number of call graphs that can produce a given dominance tree, while, second, there will be an added ability of the dominance tree to represent more detailed information. For example, a "single call in" "isolated" subtree is independent of its neighboring subtrees.

4.3 Modifying the Dominance Tree to Highlight "Isolated" Subtrees: The Moral Dominance Tree

The shading of vertices on the dominance tree enables us to identify easily "single call in" and "multiple calls in" subtrees. The shading also shows where parental loss has occurred in the abstraction of the calling structure from call graph to dominance tree. Strongly directly vertices have the same, unique parent on both graphs while the directly dominated vertices had at least two parents on the call graph of which at most one can be a parent on the dominance tree. We may modify the shading to indicate whether directly dominated vertices are dominated by one of their parents from the call graph or by an ancestor. Without altering the layout of the tree, so that Corollary 1 remains applicable, such a modification provides more information about the calling structure.

Theorem 3 shows the importance of "isolated" or "nonisolated" subtrees when determining the relationships between subtrees on the dominance tree; such subtrees are not highlighted on the dominance tree but may be formally identified by determining whether, for each $h \in V_{\mathcal{D}_f}$, $de_{\mathcal{D}_f}(h) = de_C(h)$. Thus, if $h \in V_{\mathcal{D}_f}$ is the root of a "nonisolated" subtree, then the abstraction of the call graph results in h losing some of its descendents: $de_{\mathcal{D}_f}(h) \subset de_C(h)$. In an analogous way to using vertex shading to highlight loss of parents in the abstraction, we may use differing vertex shapes on the dominance tree to represent those vertices for which $ch_C(h) \not\subseteq ch_{\mathcal{D}_f}(h)$, $ch_C(h) \subseteq ch_{\mathcal{D}_f}(h)$ only, or $de_C(h) = de_{\mathcal{D}_f}(h)$. We propose modifying the dominance tree into the moral dominance tree.

Definition 10. *The moral dominance tree corresponding to a root node, f, is the graph $\mathcal{G}_{\mathcal{D}_f} = (f^*, E_{\mathcal{D}_f})$ formed from $\mathcal{G}_{C_f} = (f^*, E_{C_f})$, the subgraph of the call graph $\mathcal{G}_C = (V_C, E_C)$, where $f^* = \{f\} \cup de_C(f)$. For any two vertices $g, h \in f^*$, $(g, h) \in E_{\mathcal{D}_f}$ if g directly dominates h on $\mathcal{G}_{C_f}$. If g strongly directly dominates h on $\mathcal{G}_{C_f}$, then the vertex h is unshaded and h is shaded if g only directly dominates it. Two shadings are used to distinguish vertices directly dominated by one of their parents on $\mathcal{G}_{C_f}$ and those directly dominated by a nonparent. If $de_C(h) = de_{\mathcal{D}_f}(h)$, then the vertex is a rectangular box with rounded corners. If only $ch_C(h) \subseteq ch_{\mathcal{D}_f}(h)$, then the vertex is an ellipse. If neither of these occur, then the vertex is a rectangular box.*

The moral dominance tree has the same layout as the dominance tree, see Definition 3; it differs in the shape and shading of the vertices. Figs. 1d and 1e show the respective moral dominance trees for the call graphs in Figs. 1a and 1b. Notice that, unlike the corresponding dominance tree (Fig. 1c), the two call graphs lead to different moral dominance trees. In Fig. 1d, $D000^\bullet$ is an "isolated" subtree: $D000$ is unshaded and is a rectangular box with rounded corners. We immediately deduce that $D000^\bullet \perp\!\!\!\perp B000^\bullet \cup C000^\bullet$. However, in Fig. 1e, $D000^\bullet$ is not an "isolated" subtree: It makes calls to other subtrees. As $D000$ appears in an ellipse, we infer that these calls are made by procedures in $de_{\mathcal{D}_f}(D000)$ and not by $D000$. The differing vertex shapes introduce more of the calling structure into the visual summary without destroying the tree representation. The same is true with the vertex shadings which help illustrate how "multiple calls in" subtrees are accessed by "nonisolated" subtrees: Compare the shading of $B000$ in Figs. 1d and 1e.

The results of Sections 4.1 and 4.2 provide a formal relationship between different types of subtrees on the dominance tree. The dominance relation identifies "single call in" subtrees which are either independent or conditionally independent to adjoining "single call in" subtrees dependent upon whether they share descendents (present in "multiple calls in" subtrees). We modify the vertex shading to illustrate whether the parent of the root of a "multiple calls in" subtree is also a parent on the call graph. Theorem 3 shows that we should also consider whether subtrees "call out" to other subtrees on the dominance tree; we add this information to the graphic using differing vertex shapes.

4.4 Additional Benefits of the g.c.i. Relation

The problem of multiple root nodes. The power of the g.c.i. relation is that it can be used to assess the relationship between any collections of vertices by constructing the corresponding moral graph. It can, for example, determine the relationship between different root nodes of a call graph, a facet that the dominance relation is unable to do. Relationships obtained from the moral graph are valid for the entire graphical model. Contrast this to the dominance relation for call graphs with multiple root nodes. Here, multiple dominance trees are constructed and procedures may appear on many dominance trees. Relationships obtained on one dominance tree may not hold on the call graph; procedures can be strongly dominated on one tree and only directly dominated on another, they may have different direct dominators on different dominance trees.

Failure to capture potential reuse candidates. In Fig. 2b, $C001$ and $D001$ are the only "single call in" subtrees dominated by $A000$ and Theorem 2 may be used to deduce that $(C^* \perp\!\!\!\perp D^*) | B^*$. The moral dominance tree does not alter the shape of the dominance tree and the conditional independence of C^* and D^* is thus not apparent on the moral dominance tree corresponding to Fig. 2a. We now explore whether we can modify the call graph to better capture its structure on the moral dominance tree.

Consider the collection of vertices $\{A000, D001, D000\}$. Each pair of vertices in the collection are joined by an edge. If we try to add any other vertex to the collection, then this is no longer the case. For example, if we add $D200$, then we have $A000 \not\sim D200$. The collection $\{A000, D001, D000\}$ is termed a clique. To quote Whittaker [28, p. 59], "a graph or subgraph is complete if all vertices are joined with either directed or undirected edges. A clique is a subset of vertices which induce a complete subgraph but for which the addition of a further vertex renders the induced subgraph incomplete." Cliques are central to the study of conditional independencies because any vertex in a clique on the moral graph is dependent upon all of the other vertices in the clique. Suppose that the graph $\mathcal{G} = (V, E)$ is a DAG and the collection $G \subseteq V$ forms a clique on $\mathcal{G}$. There is a unique $g \in G$ such that $(g, h) \in E$ for all $h \in G^{\dagger}$, where $G^{\dagger} = G \setminus \{g\}$. We term g the clique-parent of G. For example, in the clique $\{A000, D001, D000\}$ on the call graph in Fig. 2a, the clique-parent is $A000$. Note that any clique on the call graph is also a clique on the error propagation graph. The dominance relation may struggle with a clique on a call graph because of the dependence within the clique. A natural extension of the dominance relation is to seek the collection of vertices, H, for which, for all $h \in H$, $G^{\dagger}$ is an $(f, g)\text{-separator}$, where f is a root node on the call graph. If G comprises of just two vertices, then $G^{\dagger}$ is a single vertex, $g^{\dagger}$ say, and H comprises the set of procedures $g^{\dagger}$ dominates on $\mathcal{G}_{\mathcal{D}_f}$. This provides a formal way of extending the separator set from a single vertex to a collection of vertices. Indeed, in a similar way to how cycles are collapsed on the call graph, we may consider recursively collapsing all cliques on the call graph to just two vertices: g and $G^{\dagger}$ before creating the dominance tree. To illustrate this, consider the call graph in Fig. 2a. We examine cliques with at least three vertices. There are two such cliques which have $A000$ as a clique-parent: $\{A000, C001, C000\}$ and $\{A000, D001, D000\}$. We merge $C001$ and $C000$ into a single vertex $\{C001, C000\}$ whose children are the combined children of $C001$ and $C000$. We similarly merge $D001$ and $D000$ together; the merged vertex having as children the union of the children of $D001$ and $D000$. On the resultant call graph, $\{D001, D000\}$ is the clique-parent of the clique $\{\{D001, D000\},\ D100, D200\}$ and, so, $D100$, $D200$ may be merged to a single vertex. All other cliques on the call graph contain just two vertices. The modified call graph and moral dominance tree are shown in Fig. 2c and Fig. 2d respectively. Notice that there are only two out of 11 shaded vertices in Fig. 2d compared with the seven out of 14 in Fig. 2b which, intuitively, suggests the usefulness of Fig. 2d over Fig. 2b for program comprehension of the original call graph, Fig. 2a. The merging of the vertices in Fig. 2c enables the dominance relation to obtain the three subsets B^*, C^*, D^* discussed in Section 2.4. The subtrees C^* and D^* have a different shape on Fig. 2d reflecting the g.c.i. property $(C100 \perp\!\!\!\perp C200) | C110$, whereas $D100$ and $D200$ are dependent. This contrasts with the scenario in Fig. 2b. The similarity in structure between Figs. 1b and 2a is now apparent. Indeed, removing the cliques in Fig. 1b results in merging the vertices $D100$ and $D200$ so that the resultant call graph has the same shape as that in Fig. 2c.

5 Conclusion

Dominance tree analysis may be used to identify subtrees which may be considered as potential reuse candidates. The subtrees considered are those we termed "single call in" subtrees. The dominance tree does not explain the relation between "single call in" subtrees and vertices who are only directly dominated.

To address this, we introduce a g.c.i. relation over the call graph. This supports the argument for the potential reuse candidates obtained from the dominance tree by identifying "single call in" subtrees as being either independent or conditionally independent of other "single call in" subtrees. The conditional independence occurred when "single call in" subtrees made calls to the same "multiple call in" subtrees. The "multiple call in" subtrees have a root which is only directly dominated and, so, the conditional independence relation not only supports the dominance tree analysis, but strengthens it by explaining the relationship between the strongly directly and directly dominated vertices.

We argue that it is not just "single call in" subtrees that should be highlighted as potential reuse candidates, but also "isolated" subtrees, subtrees which make no calls to any other subtree on the dominance tree. As such, we propose modifying the dominance tree to the moral dominance tree which provides a greater understanding of the relationships between individual branches and also highlights areas where further investigation, in particular, the "nonisolated" "multiple calls in" subtrees, using the g.c.i. relation is required.

The g.c.i. relation is a tool for investigating any collection of vertices. It provides a formal theoretical framework for the previous heuristic approach, thus enhancing the argument for the adoption of potential reuse candidates and developing a formal relationship between the candidates. Additionally, we are able to understand collections where the dominance relation exhibited a lack of understanding. We also considered how the dominance relation could be improved so that it could handle cliques. Combining the dominance tree analysis with the g.c.i. relation provides us with a more detailed understanding of the relationships within the calling structure and, thus, our level of comprehension.

Appendix

Proofs of Theorems

Proof of Theorem 2. Note that for any $h \in V_C$, h^* is also the collection of h and its ancestors on the error propagation graph. From Lemma 1, to show (5), we construct the associated moral graph $\tilde{\mathcal{G}}_M(h^* \cup H^*)$, where $H^* = \cup_{k=1}^{l} h_{j_k}^*$ and consider separations on this graph. Notice that $V_M(h_i^* \cup H^*) = h_i^* \cup H^*$. Let $A = \cup_{k=1}^{l}\{de_C(h_i) \cap de_C(h_{j_k})\}$, $B = \cup_{k=1}^{l}\{de_C(h_i) \cap de_C^c(h_{j_k})\}$, and $C = \cup_{k=1}^{l}\{de_C^c(h_i) \cap de_C(h_{j_k})\}$; A, B, and C are mutually incompatible.

If $A = \emptyset$, then the subgraphs h_i^* and H^* are unconnected on $\mathcal{G}_C$ and, thus, on $\tilde{\mathcal{G}}_C$. If they are connected on

$\tilde{\mathcal{G}}_M(h_i^* \cup H^*)$, the path must have been formed by the marriage of some $h_{i1} \in de_C(h_i)$ and some $h_{*1} \in \cup_{k=1}^{l} de_C(h_{j_k})$. Since $A = \emptyset$, $de_C(h_i) = B$, and $\cup_{k=1}^{l} de_C(h_{j_k}) = C$.

If $A \neq \emptyset$, then the subgraphs h_1^* and H^* are connected on $\mathcal{G}_C$ and, thus, on $\tilde{\mathcal{G}}_C$. The connecting vertices are A. There is no arc between any $h_{i1} \in B$ and any $h_{*1} \in C$ (and vice versa). Any path between B and C which does not pass through an element of A must evolve through the marriage of some $h_{i1} \in B$ and some $h_{*1} \in C$.

In either case, we require the addition of an arc between some $h_{i1} \in B$ and some $h_{*1} \in C$. This will occur, see Definition 9, if there exists $h_2 \in h_i^* \cup H^*$ such that $\{(h_{i1}, h_2), (h_{*1}, h_2)\} \subseteq E_R$ or, equivalently, $\{(h_2, h_{i1}), (h_2, h_{*1})\} \subseteq E_C$. If $h_2 \in A$, then $\{h_{i1}, h_{*1}\} \subset A$: a contradiction. If $h_2 \in h_i^* \setminus A$, then $h_{*1} \in de_C(h_i)$: a contradiction. If $h_2 \in H^* \setminus A$, then $h_{i1} \in \cup_{k=1}^{l} de_C(h_{j_k})$: a contradiction. Thus, there is no such $h_2 \in h_i^* \cup H^*$ and the results follow. □

Proof of Theorem 3. Once more, for any $h \in V$, we let $h^* = h \cup de_C(h)$. Let $G^* = \cup_{i=1}^{m} g_i^*$. From Lemma 1, we need to consider separations on $\tilde{\mathcal{G}}_M(h_u^* \cup G^*)$. Notice that $V_M(h_u^* \cup G^*) = h_u^* \cup G^*$. Since $de_C(h_u) = de_{\mathcal{D}_f}(h_u)$, then the only calls from $V_{\mathcal{D}_f} \setminus h_u^*$ to h_u^* on $\mathcal{G}_C$ are to h_u only.

If there is no direct path between each g_i and h_u, then each $g_i \in V_{\mathcal{D}_f} \setminus h_u^*$ and $h_u \notin de_C(g_i)$. Thus, $g_i^* \cap h_u^* = \emptyset$ and the subgraphs h_u^* and G^* are unconnected on $\mathcal{G}_C$ and, thus, on $\tilde{\mathcal{G}}_C$. For them to be connected on $\tilde{\mathcal{G}}_M(h_u^* \cup G^*)$, the path must have been formed by the marriage of some $h_{u1} \in h_u^*$ and some $g_{*1} \in G^*$. We may show this cannot occur in an identical way to the proof of Theorem 2. Property (7) thus follows.

Property (8) also follows by observing that, if each $g_i \in an_C \cap V_{\mathcal{D}_f}$, then the subgraphs h_u^* and G^* are unconnected on $\mathcal{G}_C$, but only at h_u. Following the proof of Theorem 2, we show that there can be no marriage between some $h_{u1} \in h_u^*$ and some $g_{*1} \in H^* \setminus h_u^*$. □

ACKNOWLEDGMENTS

This work was supported by grant GR\M76775 from the UK Engineering and Physical Sciences Research Council.

REFERENCES

[1] A. Bobbio, L. Portinale, M. Minichino, and E. Ciancamerla, "Improving the Analysis of Dependable Systems by Mapping Fault Trees into Bayesian Networks," *Reliability Eng. and System Safety,* vol. 71, no. 3, pp. 249-260, 2001.

[2] E. Burd and M. Munro, "Enriching Program Comprehension for Software Reuse," *Proc. Fifth Int'l Workshop Program Comprehension,* pp. 130-137, 1997.

[3] E. Burd and M. Munro, "A Method for the Identification of Reusable Units through the Reengineering of Legacy Code," *J. Systems and Software,* vol. 44, no. 2, pp. 121-134, 1998.

[4] E. Burd and M. Munro, "Evaluating the Use of Dominance Trees for C and COBOL," *Proc. 1999 Int'l Conf. Software Maintenance,* pp. 401-410, 1999.

[5] E. Burd and M. Munro, "Supporting Program Comprehension Using Dominance Trees," *Annals Software Eng.,* vol. 9, pp. 193-213, 2000.

[6] E. Burd, M. Munro, and C. Wezeman, "Analysing Large COBOL Programs: The Extraction of Reusable Modules," *Proc. 1996 Int'l Conf. Software Maintenance,* pp. 238-243, 1996.

[7] E. Burd, M. Munro, and C. Wezeman, "Extracting Reusable Modules from Legacy Code: Considering Issues of Module Granularity," *Proc. Third Working Conf. Reverse Eng.,* pp. 189-197, 1996.

[8] G. Canfora, A. Cimitile, A. De Lucia, and G.A. Di Lucca, "Decomposing Legacy Systems into Objects: An Eclectic Approach," *Information and Software Technology,* vol. 43, no. 6, pp. 401-412, 2001.

[9] G. Canfora, A. De Lucia, G.A. Di Lucca, and A.R. Fasolino, "Recovering the Architectural Design for Software Comprehension," *Proc. Third Workshop Program Comprehension (WPC '94),* pp. 30-38, 1994.

[10] A. Cimitile, A. De Lucia, G.A. Di Lucca, and A.R. Fasolino, "Identifying Objects in Legacy Systems," *Proc. Fifth Int'l Workshop Program Comprehension,* pp. 138-147, 1997.

[11] A. Cimitile, A. De Lucia, G.A. Di Lucca, and A.R. Fasolino, "Identifying Objects in Legacy Systems Using Design Metrics," *J. Systems and Software,* vol. 44, no. 3, pp. 199-211, 1999.

[12] A. Cimitile and G. Visaggio, "Software Salvaging and the Call Dominance Tree," *J. Systems and Software,* vol. 28, no. 2, pp. 117-127, 1995.

[13] R.G. Cowell, A.P. Dawid, S.L. Lauritzen, and D.J. Spiegelhalter, *Probabilistic Networks and Expert Systems.* Springer, 1999.

[14] A.P. Dawid, "Conditional Independence in Statistical Theory (with Discussion)," *J. Royal Statistics Soc. B,* vol. 41, no. 1, pp. 1-31, 1979.

[15] A.P. Dawid, "Conditional Independence for Statistical Operations," *Annals of Statistics,* vol. 8, no. 3, pp. 598-617, 1980.

[16] M. Goldstein, "Influence and Belief Adjustment (with Discussion)," *Influence Diagrams, Belief Nets and Decision Analysis,* R.M. Oliver and J.Q. Smith, eds., pp. 143-174, Wiley, 1990.

[17] M.S. Hecht, *Flow Analysis of Computer Programs.* Amsterdam: North-Holland, 1977.

[18] IEEE, *IEEE Standard Glossary of Software Engineering Terminology.* New York: IEEE Press, 1983.

[19] F.V. Jensen, *An Introduction to Bayesian Networks.* London: UCL Press 1996.

[20] S.L. Lauritzen, *Graphical Models.* Oxford Science Publications, 1996.

[21] S.L. Lauritzen, A.P. Dawid, B.N. Larsen, and H.-G. Leimer, "Independence Properties of Directed Markov Fields," *Networks,* vol. 20, no. 5, pp. 491-505, 1990.

[22] S.L. Lauritzen and D.J. Spiegelhalter, "Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems (with Discussion)," *J. Royal Statistics Soc. B,* vol. 50, no. 2, pp. 157-224, 1988.

[23] H.A. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhl, "A Reverse-Engineering Approach to Subsystem Structure Identification," *J. Software Maintenance: Research and Practice,* vol. 5, no. 4, pp. 181-204, 1993.

[24] J. Pearl, *Probabilistic Inference in Intelligent Systems.* San Mateo, Calif.: Morgan Kaufman, 1988.

[25] J.Q. Smith, "Influence Diagrams for Statistical Modelling," *Annals of Statistics,* vol. 17, no. 2, pp. 654-672, 1989.

[26] J.Q. Smith, "Statistical Principles on Graphs (with Discussion)," *Influence Diagrams, Belief Nets and Decision Analysis,* R.M. Oliver and J.Q. Smith, eds., pp. 89-120, Wiley, 1990.

[27] D.J. Spiegelhalter, "Discussion on 'Statistical Principles on Graphs' by Smith," *Influence Diagrams, Belief Nets and Decision Analysis,* R.M. Oliver and J.Q. Smith, eds., pp. 113-116, Wiley, 1990.

[28] J. Whittaker, *Graphical Models in Applied Multivariate Statistics.* Chichester: Wiley, 1990.

[29] D.A. Wooff, M. Goldstein, and F.P.A. Coolen, "Bayesian Graphical Models for Software Testing," *IEEE Trans. Software Eng.,* vol. 28, no. 5, pp. 510-525, May 2002.

Simon C. Shaw is a research associate in the Statistics and Probability Group in the Department of Mathematical Sciences, University of Durham, United Kingdom. His research interests include Bayes (linear) methods and Bayesian methods for software testing. He is particularly concerned with the analysis of collections of (second-order) exchangeable sequences where the sequences may be infinite or finite; uses of (generalized) conditional independence and graphical models.

Michael Goldstein is a professor of statistics in the Department of Mathematical Sciences, University of Durham, United Kingdom. His research interests are concerned with foundations, methodology, and applications of the Bayesian approach to statistics and decision analysis. In particular, he has developed applications of the Bayesian approach for uncertainty analysis for computer models of large scale physical systems and for assessing software reliability through efficient test design, for each of which he has received both commercial and UK Engineering and Physical Sciences Research Council support.

Malcolm Munro is a professor of software engineering in the Department of Computer Science at the University of Durham, United Kingdom. His main research focus is software visualization, software maintenance and evolution, and program comprehension. The concern of the research is to establish how legacy systems evolve over time and to discover representations (visualizations) of those systems to enable better understanding of change. He has been actively involved with the IEEE International Conference on Software Maintenance and the International Workshop on Program Comprehension. He has led a number of UK EPSRC funded projects including Release (Reconstruction of Legacy Systems), VVSRE (Visualising Software in a Virtual Reality Environment), GUSTT (Guided Slicing and Targeted Transformation), and Jigsaw (Distributed and Dynamic Visualisation Generation). He is also involved in research in SaaS (Software as a Service) and the application of Bayesian Networks to software testing and program comprehension.

Elizabeth Burd received the PhD degree in software engineering. She is a senior lecturer in the Research Institute of Software Evolution at the University of Durham, United Kingdom. Her research interests are in the areas of software evolution, software reuse, maintenance and reverse engineering. She is currently serving as a committee member of a number of internal conferences in these areas. She has collaborated with a number of industrial companies during the process of her research including British Telecommunication, BAe, and Logica. She has also been an active researcher within an ESPRIT consortium. She is a member of the IEEE and the IEEE Computer Society.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.