Moral dominance relations for program comprehension

Simon C Shaw[1], Michael Goldstein[1], Malcolm Munro[2,3], Elizabeth Burd[2]

[1] *Department of Mathematical Sciences, University of Durham, Science Laboratories, South Road, Durham, DH1 3LE, UK*

[2] *Research Institute in Software Evolution, Department of Computer Science, University of Durham, Science Laboratories, South Road, Durham, DH1 3LE, UK*

[3] *Corresponding author. E-mail address: malcolm.munro@durham.ac.uk Tel.: +44-191-374-2634*

ABSTRACT

Dominance trees have be used as a means for reengineering legacy systems into potential reuse candidates. The dominance relation shapes the form of the reuse candidates which are identified as the strongly directly dominated subtrees. We review the approach, illustrating how the dominance tree fails to show the relationship of the strongly directly dominated nodes to the directly dominated nodes. We propose introducing a relation of generalised conditional independence which strengthens the argument for the adoption of the potential reuse candidates suggested by the dominance tree by explaining their relationship with the directly dominated nodes. This leads to an improved dominance tree, the moral dominance tree, which helps aid program comprehension available from the tree. We also argue that the generalised conditional independence relation identifies potential reuse candidates that are missed by the dominance relation.

Keywords: Dominance tree; generalised conditional independence; directed graphical model; reuse candidate; reengineering; program comprehension.

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 in order to adapt to this change. 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', [12]. Program comprehension, in this setting, involves acquiring knowledge about programs, as well as any remaining documentation and operating procedures. We aim to both understand the software through visualisations of models of it, such as the call structure, and to identify areas of the software which may be remodularised into separate modules as a means of aiding the maintenance process by localising the impacts of change. Further, these identified areas are potentially reusable and as such we refer to them as potential reuse candidates.

In this paper we investigate current approaches to the identification of potential reuse candidates via an abstraction of the calling structure called the dominance tree. We then illustrate potential difficulties with the approach and argue how the adoption of a relation of conditional independence can strengthen the case for the adoption of the potential reuse candidates suggested by the dominance tree whilst also aiding comprehension of the calling structure in areas unexplained by the dominance tree.

2 Program comprehension using dominance trees

2.1 The call graph

The calling structure of a piece of software provides a high level description of the flow of the program. This structure describes the procedural units and the relationships between them. The procedural unit of C is termed a function; it is termed a paragraph in COBOL and a method in Java. In this paper we use the generic term 'procedure' and the relationships between procedures are termed 'calls'. We visualise the calling structure by presenting it as a graph. All relevant graph theory notation used in this paper is summarised in Appendix I.

Definition 1 *A call graph is a directed graph, $\mathcal{G} = (V, E)$. The finite set of nodes, V, consists of the procedures that may be called in the program. For any two procedures f, $g \in V$ if there is a potential call to g by f then the arc (f, g) appears on the graph. The complete collection of arcs is denoted by E.*

Without loss of generality, we shall assume that the call graph is a directed acyclic graph (DAG). The representation as a DAG may be obtained from the call graph by collapsing every strongly connected subgraph into one node; see Section 2 of [1] for further details.

Definition 2 *A root node of a call graph $\mathcal{G} = (V, E)$ is a procedure which is not called by any other procedure.*

Since a call graph is a DAG, it must have at least one root node. A root node is often called an entry/exit point. Figure 1 shows an example of a very simple call graph; it has a single root node $A000$.

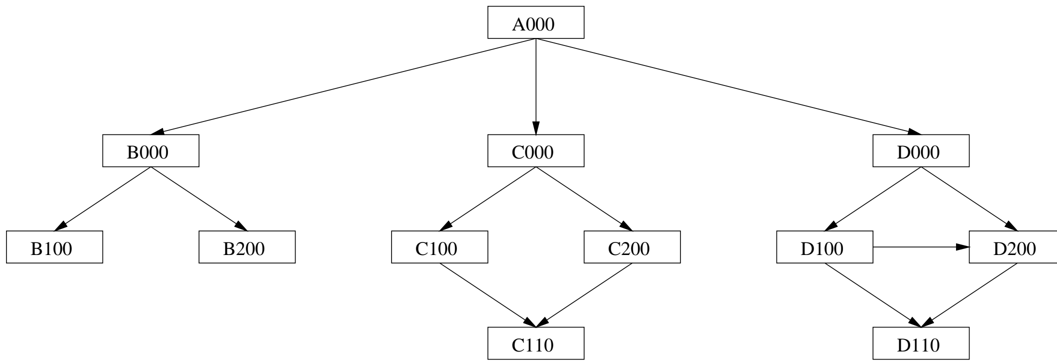


Figure 1: A simple call graph. Procedure $A000$ calls procedures $B000$, $C000$ and $D000$. Procedure $B000$ calls procedures $B100$ and $B200$ and so on.

The layout of the nodes in Figure 1 and the paucity of calls make it straightforward to examine the potential flow of code in the program. For example, by removing the procedure $A000$ and the three calls it makes from the graph we are left with a subgraph of $\mathcal{G}$ which consists of three disconnected pieces of code: the 'B-code', $B^* = \{B000, B100, B200\}$; the 'C-code', $C^* = \{C000, C100, C200, C110\}$; and the 'D-code, $D^* = \{D000, D100, D200, D110\}$. Intuitively, it would seem that these three collections are unconnected and may be assessed separately. A software maintainer interested only in B^* need not understand C^* or D^* for once $B000$ has been called, the execution of the program exists purely in the B^* until $B000$ is exited. Such a conclusion seems intuitively clear from the call graph, but is there a way we can formalise it and identify these collections of procedures where each collection is separate from any other? Moreover, the identification of possible collections and understanding of the call graph of Figure 1 was aided by the simplicity of the call graph. It will be less clearcut in call graphs that may have thousands of procedures and calls.

One approach is to make a further abstraction of the call structure by converting the call graph into a directed tree. This may be achieved by using the dominance relation; the directed tree is termed the dominance tree.

2.2 The dominance tree

The dominance tree aims to assist the program comprehension by reducing information overload during the early stages of comprehension and by identifying sections of code which may be remodularised into single modules. The dominance tree is a directed tree over the collection of procedures on the call graph $\mathcal{G} = (V, E)$. It is formed using the relation of strong direct and direct dominances, [11].

Definition 3 *If $f \in V$ is a root node of the call graph and procedures $g, h \in de_{\mathcal{G}}(f)$, the descendents of f on $\mathcal{G}$, then procedure g dominates h if and only if every path $f \rightarrow h$ on $\mathcal{G}$ intersects g. We say that g directly dominates h if and only if all procedures that dominate h dominate g. g strongly directly dominates h if and only if g directly dominates h and is the only procedure that calls h.*

The dominance relation may be viewed as a graph separation (see Appendix I) property. g dominates h if and only if g separates f from h on $\mathcal{G}$, that is g is a (f, h)-separator. Notice that whilst the separation property may be applied to any three collections of nodes, the dominance relation applies to three nodes, one of which is a root node. The direct dominance relation identifies, for each node, a single dominator from the collection of dominators of that node.

Definition 4 *The dominance tree corresponding to a root node f is the graph $\mathcal{G}_{\mathcal{D}_f} = (\{f\} \cup de_{\mathcal{G}}(f), E_{\mathcal{D}_f})$ formed from the call graph $\mathcal{G} = (V, E)$. For any two nodes g, $h \in de_{\mathcal{G}}(f)$, $(g, h) \in E_{\mathcal{D}_f}$ if g directly dominates h. The node h is shaded if g only directly dominates h.*

Since $\mathcal{G}_{\mathcal{D}_f}$ is a tree, there is an easy way to provide the visual representation: we may place each node immediately below its parent. The dominance tree may aid program comprehension by reducing the complexity of visualisation of the call graph. If $\mathcal{G}$ has a single root node, f, then $\{f\} \cup de_{\mathcal{G}}(f) = V$ and thus $\mathcal{G}_{\mathcal{D}_f}$ includes all of the procedures of $\mathcal{G}$; we write $\mathcal{G}_{\mathcal{D}_f} = \mathcal{G}_{\mathcal{D}}$. Work on dominance trees have been carried out by [5, 9, 3, 4]. Figure 2 gives the dominance tree corresponding to the call graph of Figure 1.

Figure 1 has a single root node and so there is a single dominance tree, $\mathcal{G}_{\mathcal{D}}$. We shade the nodes which are not strongly directly dominated. For example, $D000$ strongly directly

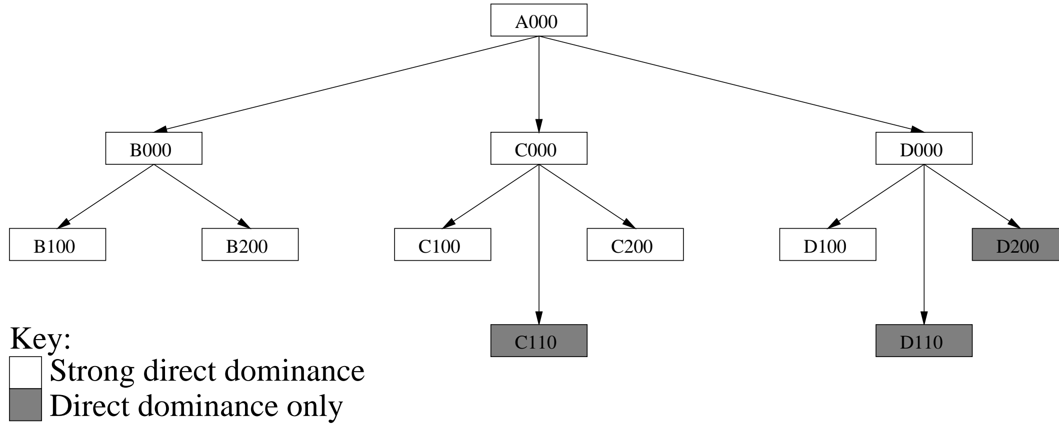


Figure 2: The dominance tree corresponding to the call graph of Figure 1.

dominates $D100$, whilst $D110$ is only directly dominated by $D000$. Nodes that are only directly dominated have become disinherited from some, possibly all, of their parents: they had at least two parents and may not be directly dominated by one of their parents. In Figure 2, the node $D110$ is directly dominated by $D000$, but $D000$ is not a parent of $D110$ on Figure 1. Thus, $E_{\mathcal{D}} \nsubseteq E$: the dominance tree is not merely the call graph with some edges removed. The nodes 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 call graph to dominance tree at the only directly dominated nodes. Intuitively, the greater the proportion of shaded nodes, the more problematic program comprehension may be from the dominance tree.

2.3 Identifying reuse candidates

One of the aims of the dominance tree is to identify potential reuse candidates within the software which may then be reengineered into separate modules. This modularisation helps make the software more flexible and maintainable.

Burd & Munro [1] describe a 10 step method for reengineering legacy systems into potential reuse candidates. The identification of potential reuse candidates is made by a dominance tree analysis and a case study of the approach may be found in [2]. In this subsection, we review the approach and suggest possible limitations to the understanding gained from the approach.

Burd & Munro [2] write that 'the directly dominates and strongly directly dominates relations define where re-modularisation can occur. For instance, where directly dominates

relations are identified this means that calls are made to other nodes within the branch of the tree'. In Figure 2, the node $C110$ is only directly dominated which indicates that calls are made to it by either $C100$ or $C200$. From Figure 1, we can confirm that both $C100$ and $C200$ call $C110$.

We wish to consider subtrees on the dominance tree. For a dominance tree, $\mathcal{G}_{\mathcal{D}_f}$, and any node $h \in V_{\mathcal{D}_f}$, we are interested in the subtree consisting of the collection of nodes $\{h\} \cup de_{\mathcal{D}_f}(h)$, the node h and all its descendents on $\mathcal{G}_{\mathcal{D}_f}$. We denote this subtree by h^*, that is $h^* = \{h\} \cup de_{\mathcal{D}_f}(h)$. The dominance relation means that the only calls from $V_{\mathcal{D}_f} \setminus h^*$ to h^* on $\mathcal{G}$ are to h itself. If h is strongly directly dominated then there is a single call to h and this is shown on $\mathcal{G}_{\mathcal{D}_f}$; $\mathcal{G}_{\mathcal{D}_f}$ illustrates how h^* is accessed by the other nodes, $V_{\mathcal{D}_f} \setminus h^*$. In this case, we term h^* a 'single call in' subtree.

We now demonstrate, through a series of examples, the strengths and weaknesses of analysis based on the dominance tree.

Example 1: All children are strongly directly dominated

The strong direct dominance relation thus identifies 'single call in' subtrees and Burd & Munro [1] identify these subtrees as areas of potential reuse. Figure 1, and the resulting dominance tree, Figure 2, are discussed by Burd & Munro ([1]; Figure 4). The three subtrees $B000^* = B^*$, $C000^* = C^*$, and $D000^* = D^*$ are identified as potential reuse candidates. The strong roots of these subtrees ($B000$, $C000$, and $D000$ respectively) are all strongly directly dominated by $A000$ and so these are all 'single call in' subtrees.

The dominance relation identifies that, for example, once $B000$ has been called by $A000$, execution cannot switch to either C^* or D^* until $B000$ is exited. The analogous statement applies for $C000$ and $D000$. Moreover, since $B^* \cup C^* \cup D^* = V \setminus A000$, we may deduce from the dominance tree that once $B000$ has been called by $A000$ execution remains solely in B^* until $B000$ is exited. The similar statement applies for $C000$ and $D000$. We determine from the dominance tree that the subtrees B^*, C^*, D^* constitute three collections of code for which the calling structure does not deviate from the collection it enters; there are no interrelationships between B^*, C^*, and D^* and we consider these as three potential reuse candidates.

Example 2: All children are either strongly directly dominated or are service candidates

Section 3.2 of Burd & Munro [2] considers the reuse candidates available for the call graph obtained by adding a call between the procedures $B000$ and $C110$ on Figure 1. This creates a new path $A000 \to C110$ via $B000$ which does not pass through $C000$. $C110$ is no longer dominated by $C000$; it is directly dominated by $A000$. The resultant dominance tree is shown in Figure 3, see also Figure 3 of [2].

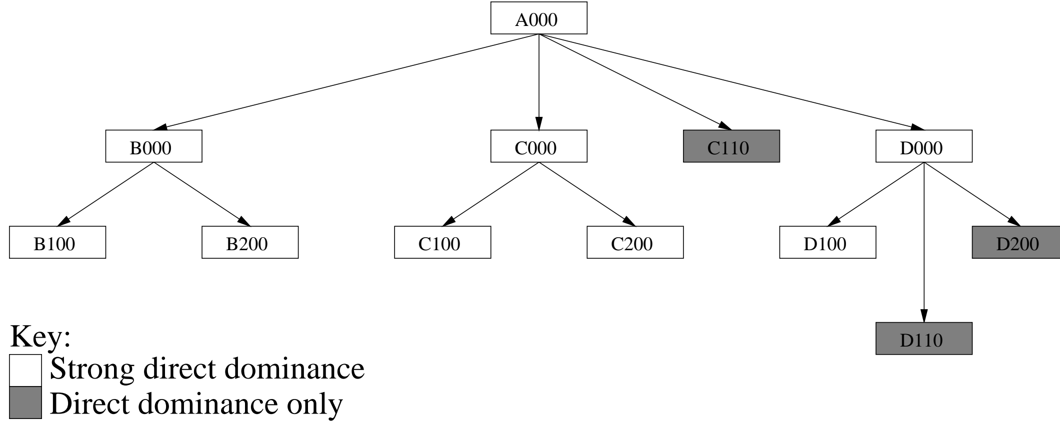


Figure 3: The dominance tree resulting from the call graph of Figure 1 with an additional call between $B000$ and $C110$.

The three subtrees $B000^* = B^*$, $C000^* = C^* \setminus C110$, $D000^* = D^*$ are all 'single call in' subtrees and so we know, from Figure 3, that there are no calls made between these subtrees. We may identify these as potential reuse candidates. However, $B^* \cup C000^* \cup D^* = V \setminus \{A000, C110\}$: we don't know the relationship between the three 'single call in' subtrees and the directly dominated node, $C110$.

Consider a dominance tree $\mathcal{G}_{\mathcal{D}_f}$ and any node $h \in V_{\mathcal{D}_f}$ which is only directly dominated. Then h has at least two parents on $\mathcal{G}$. Suppose $g \in V_{\mathcal{D}_f}$ directly dominates h, then g must dominate every node in $pa_{\mathcal{G}}(h)$. Thus, $pa_{\mathcal{G}}(h) \subseteq g^* \setminus h^*$ and there is at least one node $\tilde{g} \in de_{\mathcal{D}_f}(g) \setminus h^*$ which calls h.

Hence, in Figure 3, at least one of the three candidates B^*, $C^* \setminus C110$, and D^* will access $C110$. $C110$ is termed a service candidate: a single procedure that can be accessed by one or more of the reuse candidates. In this case, $C110$ is accessed by B^* and $C^* \setminus C110$. If a change is made to $C110$, then the effects of this change may ripple up $C^* \setminus C110$ or transfer to B^*, as Burd & Munro ([2]; p404) point out 'ripple effects are additionally restricted to the two candidates in this case $B000$ and $C000$'.

This last statement however cannot be deduced from Figure 3: the dominance tree does not exhibit how a service candidate is called by the potential reuse candidates which makes ripple effects hard to map. The dominance tree only tells us that at least one of the candidates access the service candidate; we need to return to the call graph to determine this interaction.

The limit of the dominance tree is that if a node h is only directly dominated by g then it is called by at least one node $\tilde{g} \in de_{\mathcal{D}_f}(g) \setminus h^*$. We do not know whether g itself calls h on $\mathcal{G}$. This leads to a lack of uniqueness of the dominance tree: different call graphs lead to the same dominance tree. The same dominance tree is produced by setting $pa_{\mathcal{G}}(h) = g^* \setminus h^*$ rather than the actuality of $pa_{\mathcal{G}}(h) \subseteq g^* \setminus h^*$. The lack of uniqueness becomes more apparent the larger $g^* \setminus h^*$ is. This could lead to difficulties in identifying reuse candidates and mapping ripple effects. Consider that instead of adding the arc $(B000, C110)$ to the call graph of Figure 1, we added the arc $(A000, C110)$. The dominance tree is identical to that shown in Figure 3 but in this case the only access to the service candidate $C110$ is from $C^* \setminus C110$. We might argue that the reuse candidates of Figure 2 remain valid here, namely B^*, C^*, and D^*. Any member of B^* does not access C^* or D^* and similarly for members of C^* and D^*. Ripple effects of a change to $C110$ are restricted to C^* candidate and the root node. This difference in ripple effects is not apparent on the dominance tree and requires further study of the call graph, suggesting a failure of the dominance tree to map the ripple effects.

As a further example of the potential difficulty of mapping ripple effects, consider adding the calls to Figure 1 so that every node calls $C110$. This is possible since $A000^* \setminus C110^* = V \setminus C110$. The dominance tree is still identical to Figure 3 but ripple effects from $C110$ are now no longer restricted. These illustrations of the lack of uniqueness of the dominance tree suggest that the dominance tree may not be a good vehicle for investigating ripple effects.

Example 3: Failure to isolate reuse candidates

A further example considered in [2] concerns the call graph obtained by adding a call between $B000$ and $C000$ on Figure 1. The result is that $C000$ is no longer strongly directly dominated by $A000$ for it is called by more than one node. The resultant dominance tree for this scenario is given by Figure 4, see also Figure 4 of [2].

The subtree $C000^* = C^*$ is no longer a 'single call in' subtree since the strong root of this subtree, $C000$, is only directly dominated. It is called by at least two members of $A000^* \setminus C000^* = V \setminus C^*$. From Figure 4, we can not deduce whether $A000$ calls $C000$. The

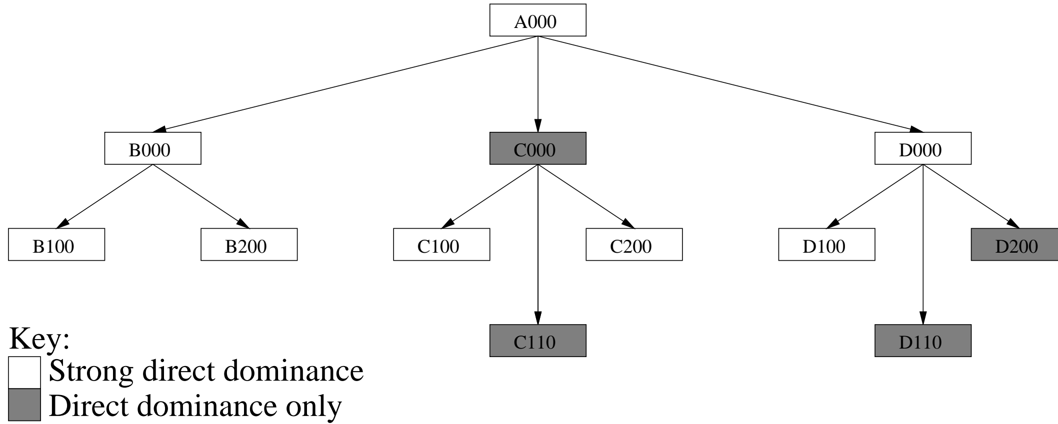


Figure 4: The dominance tree resulting from the call graph of Figure 1 with an additional call between $B000$ and $C000$.

dominance relation means that the nodes in $V \setminus C^*$ may only call $C000$ of the nodes contained in C^*. We term C^* a 'multiple calls in' subtree. As with the previous two examples, B^* and D^* remain 'single call in' subtrees and so do not call one another. Their relationship with C^* is not available on the dominance tree. An intuitive viewing of the call graph suggests two initial reuse candidates: $B^* \cup C^*$ and D^* but the dominance tree does not suggest an automated way of obtaining these candidates. As Burd & Munro [2] point out 'this represents a failure to properly isolate candidates at an appropriate level of granularity'. Notice that having called $B000$, execution does not switch between C^* and the single nodes $B100$ and $B200$. We might regard C^* as being a separate module within the module $B^* \cup C^*$. This isolation is not apparent on the dominance tree.

Further, an additional call between $D000$ and $C000$ on Figure 1 yields the identical dominance tree, Figure 4, but in this instance we would suggest the potential reuse candidates to be B^* and $C^* \cup D^*$. The same dominance tree is also obtained by adding the calls $(B000, C000)$ and $(D000, C000)$ when we might suggest no reuse candidate as the code is all interlinked via $C000$. Observe that there may still be reuse candidates, for once $C000$ is called, execution remains within C^*.

Example 4: The problem of multiple root nodes

The dominance relation is determined from a specific root node. Where a call graph has multiple root nodes multiple dominance trees must be generated and the same procedures

may appear on different root nodes. Burd & Munro ([2]; Section 4) found this problem in case studies of 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.' For example, consider adding a procedure, $C001$, to Figure 1 which calls $B200$ and $C100$ but is not called itself. We also add a call between $B000$ and $C110$. The resulting call graph is given in Figure 5.

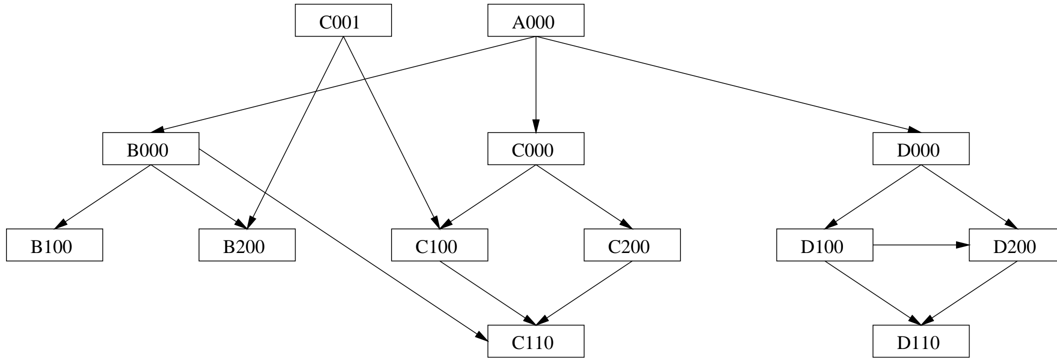


Figure 5: The call graph obtained by adding the procedure $C001$ which calls $B200$ and $C100$ and a call between $B000$ and $C110$ to the call graph of Figure 1.

Figure 5 thus has two root nodes and so will yield two dominance trees: $\mathcal{G}_{\mathcal{D}_{A000}}$ from the root node $A000$ and $\mathcal{G}_{\mathcal{D}_{C001}}$ from the root node $C001$. Figure 6 shows the dominance trees.

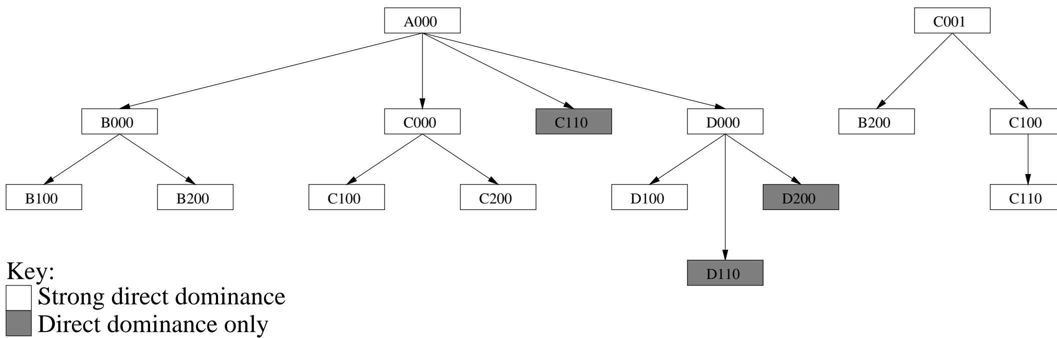


Figure 6: The dominance tree resulting from the call graph of Figure 5.

Notice that $\mathcal{G}_{\mathcal{D}_{A000}}$ is identical to Figure 3. $C110$ is strongly directly dominated by $C100$ on $\mathcal{G}_{\mathcal{D}_{C001}}$ but only directly dominated by $A000$ on $\mathcal{G}_{\mathcal{D}_{A000}}$. $\mathcal{G}_{\mathcal{D}_{C001}}$ suggests $\{C100, C110\}$ as a reuse candidate and $\mathcal{G}_{\mathcal{D}_{A000}}$ suggests $C^* \setminus C110$ as a reuse candidate. It is not clear how we

should combine the two; the dominance relation provides no guidance for the relationship between $\mathcal{G}_{\mathcal{D}_{A000}}$ and $\mathcal{G}_{\mathcal{D}_{C001}}$.

Example 5: Failure to capture potential reuse candidates

We consider the call graph obtained from Figure 1 by adding two procedures: the procedure $C001$ which calls $C000$ and $C200$ and is called by $A000$, and the procedure $D001$ which calls $D000$ and $D200$ and is called by $A000$. This call graph is shown in Figure 7.

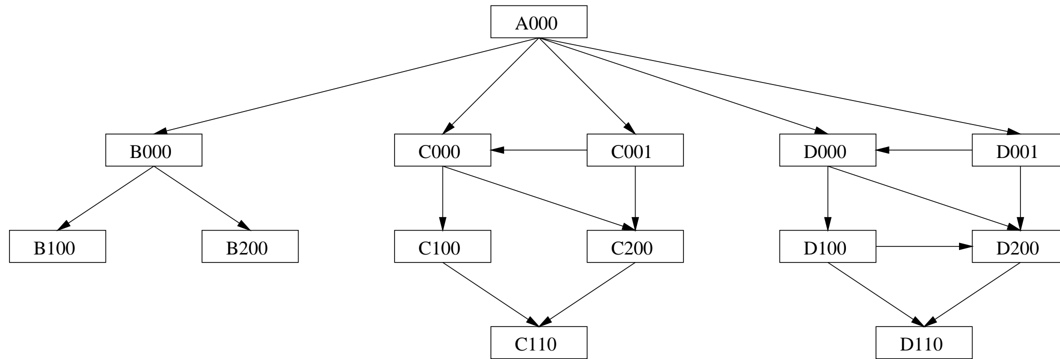
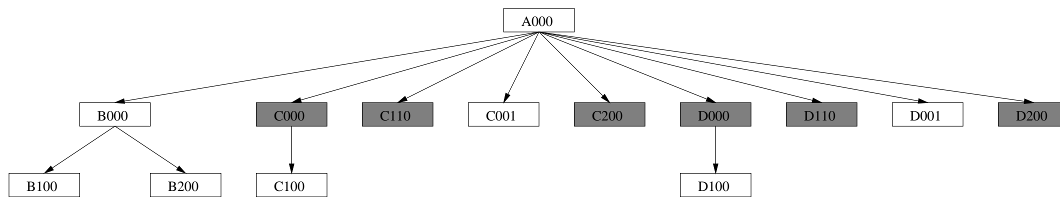


Figure 7: The call graph obtained by adding the procedure $C001$ which calls $C000$ and $C200$ and is called by $A000$ and the procedure $D001$ which calls $D000$ and $D200$ and is called by $A000$ to the call graph of Figure 1.

The straightforward layout of the call graph may lead one to suggest three reuse candidates: B^*, $C^* \cup C001$, and the $D^* \cup D001$. Execution never transfers between these three subtrees. The corresponding dominance tree is shown in Figure 8.



Key:
Strong direct dominance
Direct dominance only

Figure 8: The dominance tree resulting from the call graph of Figure 7.

Observe how the $C^* \cup C001$ and $D^* \cup D001$ decompose in the identical way on Figure 8 despite the difference in the calling structure. There are children of $A000$ on Figure 8 which

are only directly dominated and are not service candidates. The change from strong direct dominance to only direct dominance of much of C^* between Figure 2 and Figure 8 is caused by the node $C001$ and likewise $D001$ plays a similar role for D^*; the dominance tree does not illustrate the separations of Figure 7. Recall that following Definition 3 we remarked how the dominance relation is a graph separation applied to individual nodes. Graph separation may be applied to collections of nodes as well as single nodes. For example, on Figure 7, $\{C100, C200, C110\}$ are separated from $A000$ by $\{C000, C001\}$ and only $A000$ (out of the remaining nodes) calls the set $\{C000, C001\}$. This separation seems to identify the set $C^* \cup C001$ as a potential reuse candidate. It may be argued that the dominance relation restricts our identification of reuse candidates by considering separations of individual nodes rather than separations involving collections of nodes.

These examples suggest that we need a more informative graphical representation to support program comprehension which clarifies the sense in which we may identify potential reuse candidates from the dominance tree and supports other candidates which are failed to be detected on the dominance tree. We now develop such a representation.

3 Generalised conditional independence representations for the software calling structure

3.1 Procedures and uncertainty in the calling structure

For simplicity of exposition, we regard a piece of software as consisting of a database, D, and a collection of procedures which may be called and which operate on the database. We view the database as encoding the state of the program (eg. the variables). Having been called, each procedure is viewed as processing an input in order to perform an action. Following the completion of this action, the 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 write to the database. The result of the action is thus functionally dependent on the state of the database immediately prior to the call being made.

Suppose that on a call graph $\mathcal{G} = (V, E)$, a call is made to $f \in V$ by $g \in V$ with an input a. Further, assume that immediately prior to the call to f by g, the database is in state D_a. Having processed the input, the result of the action is $f_{D_a}(a)$ and the state of the database is $D_{f(a)}$ and control is returned to g. There is uncertainty however as to whether the procedure has operated correctly or whether there is an error in the procedure. Having

processed the input a, we observe the result of the action to be $\tilde{f}_{D_a}(a)$ and the state of the database to be $D_{\tilde{f}(a)}$. We have uncertainty as to whether the procedure has performed the action correctly, that is whether $\tilde{f}_{D_a}(a) = f_{D_a}(a)$ and also whether the database has been left in the desired state, that is whether $D_{\tilde{f}(a)} = D_{f(a)}$.

Definition 5 *The procedure f is said to work for input a if, for all possible database states, D_a, we have*

$$\tilde{f}_{D_a}(a) \;=\; f_{D_a}(a) \quad \textit{and} \quad D_{\tilde{f}(a)} \;=\; D_{f(a)}. \tag{1}$$

If the two conditions do not both hold, then the procedure f is in error for a.

Notice how this definition makes the error specific to the procedure. Thus, if an earlier procedure has caused an error in the database, f may still work for a if it can cope with this error. For example, suppose we have a piece of software operating an accounts system for a bank. Suppose procedure g has the function of adding a given amount to a specific account but instead adds that amount to each account. Then g performs its action correctly (the account in question does have the given amount added) but does not leave the database in the correct state (as every account has had the given amount added). If procedure f is then called with the task of reading the amount in a different account it would not be in error if it correctly read this value, even though the value is incorrect.

For each procedure, f, we define the set of possible inputs by A. We make the following definition.

Definition 6 *The procedure f is said to work if it works for each input $a \in A$. If there is an input a such that the procedure f is in error for a, then the procedure is said to not work.*

Definition 6 allows us to consider the potential propagation of errors. Suppose that procedure g does not work, that is there is an input $\tilde{a}$ for which g is in error for. g receives input from the procedures that call it. Assume $(f, g) \in E$ and f is called with input a and that this causes g to be called with input $\tilde{a}$. Since g is in error for $\tilde{a}$, this will cause either the result of the action to be incorrect or the database to be corrupted. There is an error present when control is returned to f: the error will propagate from g to f, from child to parent. In general, the error may only potentially propagate from child to parent: it depends upon whether the child is called with an input for which it is in error for. If $(f, g) \notin E$ then an error in g cannot directly propagate to g.

Definition 7 *For a call graph $\mathcal{G} = (V, E)$ with f, $g \in V$ we construct the error propagation graph $\tilde{\mathcal{G}} = (V, E_R)$ where*

$$E_R \quad = \quad \{(g, f) : (f, g) \in E\}. \tag{2}$$

The error propagation graph $\tilde{\mathcal{G}} = (V, E_R)$ maps the potential physical propagation of errors. It is the call graph with the arcs reversed. Notice that whilst we talk here about error propagation, we are interested in actions where a change in the child node on the call graph could cause a change in the parent action. We view ripple effects as being such an action.

We may view each procedure as a well defined random quantity having two possible states: 1 if the procedure works and 0 if the procedure does not work. The set V, of procedures of the call graph, may also be considered as a collection of random quantities. If we learn the state of a procedure, $g \in V$, for example $g = 1$ indicating that the procedure g is working, then this may enable us to gain information about the state of another procedure $h \in V$.

Recall the simple example of a call graph given in Figure 1 and the potential propagation of errors. We have already noted that once a call has been made to $D000$, calls are restricted to the set $\{D100, D200, D110\}$ until execution finishes and procedure $D000$ is exited. Suppose that $D100$ is in error. This error may propagate from $D100$ directly to $D000$, for example by $D000$ calling $D100$ with an input for which $D100$ is in error for. The error may then propagate to $A000$, but only via $D000$. If the state of $D000$ is already known, we would not expect to learn anything further about $A000$ from $D100$. Thus, knowledge of the state of $D000$ separates the uncertainty, in terms of whether the procedure is working, between $A000$ and $D100$. We may represent such separations using the concept of conditional independence as follows.

3.2 Generalised Conditional Independence

A random quantity X is defined to be probabilistically independent of Y if knowledge of the value of Y does not affect the uncertainty about X: there is no influence between X and Y. X is probabilistically dependent of Y if knowledge about Y does affect the uncertainty about X. Suppose that X and Y are (possibly vector valued) random quantities with joint probability density function $p(\cdot)$. Adopting the notation of Dawid [7], we write $X \perp\!\!\!\perp Y$ to denote that X and Y are probabilistically independent, $p(x, y) = p(x)p(y)$, or equivalently that $p(x|y) = p(x)$. where $p(x|y)$ is the conditional density of X given $Y = y$. For random vectors X, Y, Z, we say that X is 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 $p(x|y, z) = p(x|z)$.

We may interpret $X \perp\!\!\!\perp Y$ as meaning that any information we receive about Y does not alter our uncertainty about X, whilst $(X \perp\!\!\!\perp Y)|Z$ may be understood as having observed Z, any information we learn about Y does not alter our beliefs about X.

Dawid [7], [8] developed probabilistic conditional independence as a basic intuitive concept with its own axioms. He showed that 'many of the important concepts of statistics (sufficiency, ancillarity, etc.) can be regarded as expressions of conditional independence, and that many results and theorems concerning these concepts are just applications of some simple general properties of conditional independence.' Smith [17] discusses a generalised version of the conditional independence property writing 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 [16] agrees, arguing that ' the notions of relevance and dependence are far more basic to human reasoning than the numerical values attached to probability judgements'.

Smith [17], [18] extends [7], [8] and shows that any tertiary relation $(\cdot \perp\!\!\!\perp \cdot)|\cdot$ satisfying the following three properties, for any collections W, X, Y, Z, will behave computationally as a generalised conditional independence (g.c.i.) property.

1. $(W \perp\!\!\!\perp X)|X \cup Y$; (3)
2. $(W \perp\!\!\!\perp X)|Y$ if and only if $(X \perp\!\!\!\perp W)|Y$; (4)
3. $(W \perp\!\!\!\perp X \cup Y)|Z$ implies and is implied by the pair of conditions $\begin{cases}(W \perp\!\!\!\perp Y)|Z; \\ (W \perp\!\!\!\perp X)|Y \cup Z.\end{cases}$ (5)

Equation (3) expresses the property that 'once X is known (along with anything else Y), then no further information can be gained about X by observing W.' Equation (4) is the symmetry relation: 'if once Y is known, W is uninformative for X, then X is uninformative for W, having observed Y.' Equation (5) may be read as '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.'

We may construct the g.c.i. relation qualitatively and examine its implications in an identical manner to those for probabilistic conditional independence, or use the relation for other properties which represent types of lack of influence quantitatively without requiring the use of the full probabilistic conditional independence. Goldstein [10] constructs a tertiary

property satisfying equations (3) - (5) based on the partial quantitative specification of beliefs.

3.3 Directed graphical model

A collection of conditional independence relations may be represented graphically as follows. The nodes of the graph are random quantities; nodes are joined by directed arrows if there is a possible direct dependency between the nodes.

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

$$X_i \perp\!\!\!\perp X_j | pa_{\mathcal{G}}(X_i), \tag{6}$$

where $(\cdot \perp\!\!\!\perp \cdot)|\cdot$ is a generalised conditional independence property satisfying relations (3) - (5). $pa_{\mathcal{G}}(X_i)$ represents the set of parents of X_i on $\mathcal{G}$.

There are a number of equivalent definitions of a directed graphical model, for example see Theorem 5.14 of [6]. The most familiar representation is when $(\cdot \perp\!\!\!\perp \cdot)|\cdot$ represents probabilistic conditional independence. In this case, the directed graphical model is termed a Bayesian belief network. The Bayesian belief network represents the independencies embedded in $p(x_1, \ldots, x_n)$, the joint distribution over all the random quantities in V, that follow from the definition of the parent sets. The Bayesian belief network allows immediate construction of $p(x_1, \ldots, x_n)$ for, see for example Jensen ([13]; p20)

$$p(x_1, \ldots, x_n) \quad = \quad \prod_{i=1}^{n} p(x_i | pa_{\mathcal{G}}(x_i)). \tag{7}$$

3.4 Belief separation via the moral graph

One of the uses of directed graphical models is to understand the independence relationships expressed in a model additional to those, see relation (6), explicitly stated in the creation of the model. Smith ([18]; p90) writes that a use of the directed graphical model is 'to help the decision analyst or statistician to understand and use a model's c.i. [conditional independence] structure. He uses graphs directly to derive rigorously both the relationships embedded between variables and the forms of optimal policies implicit within a given model structure.'

This understanding is achieved by linking conditional independence with graph separation on an associated undirected graph; graph separation satisfies the conditions (3) - (5) (see

Pearl ([16]; Section 3.1)) and so itself acts as a generalised conditional independence property. The associated undirected graph is the moral graph, defined as follows.

Definition 9 *On the directed acyclic graph $\mathcal{G} = (V, E)$ for subsets $W_1, W_2, W_3 \subseteq V$, the moral graph $\mathcal{G}_M(\bigcup_{i=1}^3 W_i) = (V_M(\bigcup_{i=1}^3 W_i), E_M(\bigcup_{i=1}^3 W_i))$ is the undirected graph where*

$$V_M(\bigcup_{i=1}^{3} W_i) = \bigcup_{i=1}^{3} \{W_i \cup an_{\mathcal{G}}(W_i)\}; \tag{8}$$

$$E_M(\bigcup_{i=1}^{3} W_i) = \{\{(f,g),(g,f) \forall g,f \in V_M(\bigcup_{i=1}^{3} W_i) : (f,g) \in E\} \cup \{(g,h),(h,g) \tag{9}$$

$$\forall f,g,h \in V_M(\bigcup_{i=1}^{3} W_i) : \{(g,f),(h,f)\} \subseteq E \wedge (g,h),(h,g) \notin E\}\}. \tag{10}$$

If $V_M(\bigcup_{i=1}^3 W_i) = V$ then we write $\mathcal{G}_M(\bigcup_{i=1}^3 W_i) = \mathcal{G}_M$ and term this the full moral graph.

Less formally, we draw the subgraph of $\mathcal{G}$ with nodes W_1, W_2, W_3 and all their ancestors; for each node we 'marry' all of its parents (join them with an edge if not already joined); drop all arrows to form the moral graph $\mathcal{G}_M(\bigcup_{i=1}^3 W_i)$. For further details on moral graphs see [14].

Separations upon the moral graph are then used to identify conditional independences within the model structure as the following theorem, see Cowell *et al.* ([6]; p71), explains.

Theorem 1 *For any three collections of nodes W_1, W_2, W_3 within a Bayesian belief network $\mathcal{G} = (V, E)$, construct the moral graph $\mathcal{G}_M(\bigcup_{i=1}^3 W_i)$. Then $W_1 \perp\!\!\!\perp W_2 | W_3$ whenever W_1 and W_2 are separated by W_3 on the moral graph.*

As Smith ([17]; Section 3) illustrates, all the theory we develop about how information is transferred in probabilistic conditional independence holds for a generalised conditional independence property and so we may apply Theorem 1 to a directed graphical model rather than a Bayesian belief network and so meet the aim of Pearl ([16]; p81) as to 'whether assertions equivalent to those made about probabilistic dependencies can be derived *logically* without reference to numerical quantities.' If W and X are separated by Y on the moral graph of a directed graphical model then they are separated when we attempt to quantify the network by any approach which respects the generalised conditional independence properties.

3.5 Using the call graph to create a directed graphical model

In Subsection 3.1 we argued that each procedure may be viewed as a random quantity, the uncertainty being whether the procedure is correct, as defined in Definition 6. We argued

how if procedure f called procedure g then there was a potential of an error in g propagating to f. This potential propagation of error means that there is a direct influence between f and g: if we learn that g is in error then this causes us to increase our belief in the procedure f being in error because of the possible propagation. We argue that the potential error propagation may be used as a means to constructing a directed graphical model over the procedures of the software.

We wish to examine the effect of error propagation upon changing our beliefs about the state of a procedure. Thus, we are concentrating upon evidence drawn from the call graph, an object that is readily available. We are not considering other forms of relationships we could construct. For example, we could attempt to track areas of code written by specific programmers: learning that they have written one procedure correctly is likely to increase our belief in their competence and thus reduce our judgment about the chance that the other procedures they have written are in error. Alternatively, we could consider information flow across procedures with similar functionality or across areas of code that share similar features (such as joins between new and old areas of the code). Attempting to model these would require a detail knowledge of the code and may not even be available.

Consider the simple call graph $\mathcal{G}$ as given by the left hand graph of Figure 9. The right hand graph is the corresponding error propagation graph $\tilde{\mathcal{G}}$.

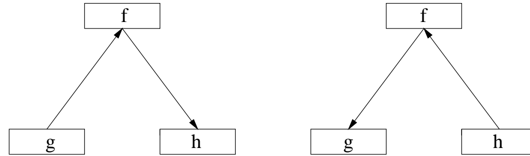


Figure 9: Left, a simple call graph with three nodes. Right, the error propagation graph. The procedures g and h are dependent, but if we know whether f works or not then the procedures are independent.

If an error is detected in h it could propagate to its parent f when h completes its action and control returns to f and then to g by the return of the control from f to g. It could not propagate directly to g. This direction of propagation is exhibited by reversing the arcs on the call graph. The result is the error propagation graph, shown as the right hand graph on Figure 9. Observe how the error propagation effects our beliefs about the states of f and g. Suppose h is found to be in error. The potential propagation causes us to increase our belief in procedure f not working and this increase is passed on to yield an increased belief in procedure g not working. Procedures g and h are not independent. However, if the state

of f was already known, that is whether or not f was working, then procedures g and h are independent. For example, if we observe f to work then the additional observation of h does not change our beliefs about g. If an error is detected in h then f working merely confirms that this error does not propagate to f. We have the relationship $(g \perp\!\!\!\perp h)|f$. It is worth pointing out here that we have not explicitly defined what tertiary property we are using for $(\cdot \perp\!\!\!\perp \cdot)|\cdot$. We are thinking in terms of either probabilistic conditional independence or second-order belief separation depending upon the level of quantification we are prepared to give but at this stage we may act qualitatively rather than quantitatively.

As a second example, we consider the call graph $\mathcal{G}$ as given by the left hand graph of Figure 10; the corresponding error propagation graph $\tilde{\mathcal{G}}$ is the right hand graph.

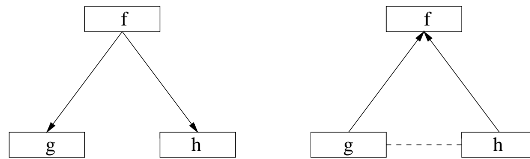


Figure 10: Left, a simple call graph with three nodes. The procedures g and h are independent, but if we know f then the procedures are dependent. The call graph with the arrows reversed, right, captures this.

The procedures g and h are independent. If g is not working, this gives us no information about the state of h. Although a potential error in g could result in h being called with the wrong input, or with the wrong database set-up, all that is relevant is whether h copes with these correctly. Now suppose that the procedure f is known not to be working. Are the procedures g and h still independent? f not working could have resulted from an error propagating from either g or h or from an error in f itself. If we learn that g works, then this will increase the belief that h is in error; procedures g and h are dependent given f. This conditional dependency of the procedures g and h given f may be captured by reversing the arcs of the call graph as shown in the right hand figure of Figure 10, for g and h are unmarried parents of f on this graph and so are joined on $\tilde{\mathcal{G}}_M$, as illustrated by the dotted line. This creates a path between g and h which does not pass through f and so f is not a (g, h)-separator on $\tilde{\mathcal{G}}_M$.

The left hand graph of Figure 11 provides a third example. The corresponding error propagation graph is the right hand graph.

The procedures g and h are dependent. If g is not working then the chance that f is not working is increased. Any errors in f may propagate to h and so the chance of h being

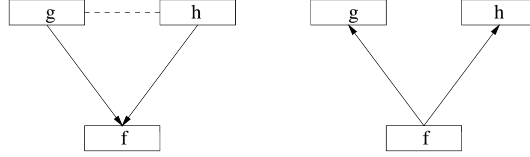


Figure 11: Left, a simple call graph with three nodes. The procedures g and h are dependent, but if we know f then the procedures are independent. The call graph with the arrows reversed, right, captures this.

in error increases. However, if f is known then the procedures g and h are independent. For example, if f is known to be working, then observing that g is not working gives no information about the state of h. Again f separates g and h on $\tilde{\mathcal{G}}_M$, but a moral graph constructed from the call graph sees an arc, see the dotted line on Figure 11, added between g and h so that f does not separate g and h on $\mathcal{G}_M$.

The three examples given in Figures 9 - 11 may be viewed as illustrating the following lemma. A full proof requires showing that equation (6) is satisfied on $\tilde{\mathcal{G}}$.

Lemma 1 *The error propagation graph $\tilde{\mathcal{G}} = (V, E_R)$ is a directed graphical model. 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 $(G \perp\!\!\!\perp H)|F$.*

In terms of the call graph, we term $\tilde{\mathcal{G}}_M(G, H, F)$ the associated moral graph. By reversing the arcs of the call graph, we may investigate conditional independences between the procedures. Identifying conditionally independent sets allows us to assess the influence of one collection of procedures upon another and use this to obtain potential reuse candidates.

4 Program comprehension using the conditional independence relation

4.1 Strongly directly dominated subtrees: ‘single call in’

In Section 2 we reviewed the use of the dominance tree in selecting potential reuse candidates and how Burd & Munro [2] identify the subtrees of $\mathcal{G}_{\mathcal{D}_f}$ whose strong root was strongly directly dominated as potential reuse candidates. We termed these subtrees ‘single call in’ subtrees. These are identified as possible sites of remodularisation because if we had two nodes h_1, h_2 strongly directly dominated by g then the subtrees h_1^*, h_2^* of $\mathcal{G}_{\mathcal{D}_f}$ have the property that there are no calls between the nodes in h_1^* and the nodes in h_2^* on $\mathcal{G}$. We now show how these ‘single call in’ subtrees may be viewed in terms of conditional

independence statements. This enables us to strengthen the argument for the adoption of the reuse candidates whilst also helping to explain the relationship of the candidates with the directly dominated nodes. We shall make use of the following theorem; the proof is in Appendix II.

Theorem 2 *Suppose $\mathcal{G} = (V, E)$ is a call graph and consider any collection of nodes h_1, h_2, $\ldots$, 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 $\mathcal{G}$. For any $i \in \{1, \ldots, m\}$, and any $1 \leq l \leq m$, $1 \leq j_1 \leq j_2 \leq \cdots \leq j_l \leq m$, $j_k \neq i$, for which $\bigcup_{k=1}^{l} \{de_{\mathcal{G}}(h_i) \cap de_{\mathcal{G}}(h_{j_k})\} = \emptyset$ then*

$$\{h_i \cup de_{\mathcal{G}}(h_i)\} \perp\!\!\!\perp \bigcup_{k=1}^{l} \{h_{j_k} \cup de_{\mathcal{G}}(h_{j_k})\}. \tag{11}$$

For any $i \in \{1, \ldots, m\}$, and any $1 \leq l \leq m$, $1 \leq j_1 \leq j_2 \leq \cdots \leq j_l \leq m$, $j_k \neq i$, for which $\bigcup_{k=1}^{l} \{de_{\mathcal{G}}(h_i) \cap de_{\mathcal{G}}(h_{j_k})\} \neq \emptyset$ then

$$\{h_i \cup de_{\mathcal{G}}(h_i)\} \perp\!\!\!\perp \bigcup_{k=1}^{l} \{h_{j_k} \cup de_{\mathcal{G}}(h_{j_k})\} | \bigcup_{k=1}^{l} \{de_{\mathcal{G}}(h_i) \cap de_{\mathcal{G}}(h_{j_k})\}. \tag{12}$$

Theorem 2 applies to any collection of nodes $\{h_1, \ldots, h_m\}$ with the property that there is no path between any h_i and h_j on the call graph. Notice that if two nodes are both strongly directly dominated by the same node on a dominance tree then there is no path between them on the call graph. Furthermore, it is straightforward to see that any subtree h^* on the dominance tree has the property that $h^* \subseteq \{h \cup de_{\mathcal{G}}(h)\}$. We may link Theorem 2 to the reuse candidates generated by the 'single call in' subtrees via the following corollary.

Corollary 1 *Suppose $\mathcal{G} = (V, E)$ is a call graph and f is a root node of $\mathcal{G}$. Additionally, consider a node g on $\mathcal{G}_{\mathcal{D}_f}$ which strongly directly dominates the nodes h_1, $\ldots$ h_m. For any $i \in \{1, \ldots, m\}$, and any $1 \leq l \leq m$, $1 \leq j_1 \leq j_2 \leq \cdots \leq j_l \leq m$, $j_k \neq i$, for which $\bigcup_{k=1}^{l} \{de_{\mathcal{G}}(h_i) \cap de_{\mathcal{G}}(h_{j_k})\} = \emptyset$ then*

$$h_i^* \perp\!\!\!\perp \bigcup_{k=1}^{l} h_{j_k}^*. \tag{13}$$

For any $i \in \{1, \ldots, m\}$, and any $1 \leq l \leq m$, $1 \leq j_1 \leq j_2 \leq \cdots \leq j_l \leq m$, $j_k \neq i$, for which $\bigcup_{k=1}^{l} \{de_{\mathcal{G}}(h_i) \cap de_{\mathcal{G}}(h_{j_k})\} \neq \emptyset$ then

$$h_i^* \perp\!\!\!\perp \bigcup_{k=1}^{l} h_{j_k}^* | \bigcup_{k=1}^{l} \{de_{\mathcal{G}}(h_i) \cap de_{\mathcal{G}}(h_{j_k})\}. \tag{14}$$

Proof - The results follow by observing that since $h_i^* \setminus h_i$ forms the set of nodes dominated by h_i on $\mathcal{G}_{\mathcal{D}_f}$, then $\{h_i^* \setminus h_i\} \subseteq de_{\mathcal{G}}(h_i)$. The reduction of the sets $h_i \cup de_{\mathcal{G}}(h_i)$, of equations (11) and (12), to h_i^* follows by the c.i. property (5). □

Corollary 1 thus provides the relationship between the subtrees on the dominance tree $\mathcal{G}_{\mathcal{D}_f}$ whose strong root is strongly directly dominated by g. We illustrate the results of Corollary 1 by returning to the examples of Subsection 2.3.

Example 1 revisited: All children are strongly directly dominated

The first example was drawn from Figure 1. The resulting dominance tree is Figure 2. The node $A000$ strongly directly dominates $B000$, $C000$ and $D000$. The subtrees $B000^* = B^*$, $C000^* = C^*$, and $D000^* = D^*$ are all 'single call in' subtrees. Without reference to Figure 1, Figure 2 reveals that there are no calls between these three subtrees. Corollary 1 enables us to deduce immediately that these subtrees are either conditionally independent or independent of one another. In Figure 2 there are no other available subtrees and so these subtrees must be independent. Applying Corollary 1 we establish, using relation (13), that $B^* \perp\!\!\!\perp C^* \cup D^*$, $C^* \perp\!\!\!\perp B^* \cup D^*$ and $D^* \perp\!\!\!\perp B^* \cup C^*$. The reuse candidates suggested by [1] are independent. Notice that the three independence statements may be reduced to illustrate pairwise independence either by using relation (13) or via the conditional independence property (5).

Example 2 revisited: All children are either strongly directly dominated or are service candidates

Notice that if g has a child h_d on $\mathcal{G}_{\mathcal{D}_f}$ which it only directly dominates, then h_d is a descendent, on $\mathcal{G}$, of at least one of the h_i. This situation was present in the second example of Subsection 2.3 when we created a service candidate by adding a call between $B000$ and $C110$ to Figure 1. The resultant dominance tree was given by Figure 3. We identify the three 'single call in' subtrees, $B000^* = B^*$, $C000^* = C^* \setminus C110$, and $D000^* = D^*$ but also the addition of the service candidate, $C110$, and from Figure 3 we may deduce that it is called by at least one of the subtrees B^*, $C000^*$ and D^*. The original call graph is required to determine how this service candidate is accessed by the 'single call in' subtrees. If it

is called by at least two of the 'single call in' subtrees then we will have, from Corollary 1, a relationship of conditional independence between the 'single call in' subtrees. In this instance, $C110 \in de_{\mathcal{G}}(B000) \cap de_{\mathcal{G}}(C000)$. Using Corollary 1 with $D000$, we find, from relation (13), that $D000^* \perp\!\!\!\perp B000^* \cup C000^*$. Indeed, Theorem 2 shows that $D^* \perp\!\!\!\perp B^* \cup C^*$. If we wish to consider the relationship between B^* and $C^* \setminus C110$ then $C110 \in de_{\mathcal{G}}(B000) \cap de_{\mathcal{G}}(C000)$ and so relation (14) is activated: we have a conditional independence relation, $(B^* \perp\!\!\!\perp C^* \setminus C110)|C110$.

In the discussion of this example in Subsection 2.3, we pointed out the lack of uniqueness of the dominance tree by illustrating that alternative call graphs lead to the same dominance tree. The relationship between the service candidate and the reuse candidates was not explained. The conditional independence relation however captures this. Suppose we added only the call $A000$ to $C110$ to Figure 1. The dominance tree is Figure 3 and Corollary 1 may be used to obtain the conditional independence statements $B^* \perp\!\!\!\perp \{C^* \setminus C110\} \cup D^*$, $\{C^* \setminus C110\} \perp\!\!\!\perp B^* \cup D^*$ and $D^* \perp\!\!\!\perp B^* \cup \{C^* \setminus C110\}$. The service candidate is only accessed by one of the reuse candidates and this supports our intuition of using the C^* as the reuse candidate in this case, for the conditional independence statements between B^*, C^* and D^* are identical to those in Figure 2. In the other alternative of allowing every procedure to call $C110$ then $C110 \in de_{\mathcal{G}}(B000) \cap de_{\mathcal{G}}(C000) \cap de_{\mathcal{G}}(D000)$ and the conditional independence statements derived from Corollary 1 capture this as: $B^* \perp\!\!\!\perp \{C^* \setminus C110\}|C110$, $B^* \perp\!\!\!\perp D^*|C110$ and $\{C^* \setminus C110\} \perp\!\!\!\perp D^*|C110$. Notice how the conditional independence relation can capture the role of the service candidate automatically; the visual aid of the call graph is not needed.

This illustrates a difference between the dominance relation and the conditional independence relation. The dominance relation is only concerned with calls to a procedure whilst the conditional independence statement takes account of calls made by a procedure. The subtrees with strongly directly dominated nodes as the strong root are, at least, conditionally independent of the adjoining subtrees which are also strongly directly dominated by the same node but whether they are conditionally independent or independent is not available on the dominance tree but this information has a vital role when considering potential ripple effects. In Figure 2, $de_{\mathcal{G}}(B000) = de_{\mathcal{D}_f}(B000)$ but this was not the case in Figure 3 and this was highlighted by the independence of B^* from C^* in Figure 2 but only conditional independence of B^* and C^* in Figure 3. In the latter, B^* called other subtrees, in the former it did not.

4.2 Relations around 'isolated' subtrees

If for any node $h_u \in V_{\mathcal{D}_f}$, we have $de_{\mathcal{G}}(h_u) = de_{\mathcal{D}_f}(h_u)$ then the subtree h_u^* does not call any other subtree on $\mathcal{G}_{\mathcal{D}_f}$. We call h_u^* an 'isolated' subtree. In terms of the call graph, this means that once h_u has been called, execution remains solely in the subtree h_u^* 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 strong root, h_u. We have the following theorem; the proof is in Appendix II.

Theorem 3 *Suppose that $\mathcal{G} = (V, E)$ is a call graph and $h_u \in V$ is such that $de_{\mathcal{G}}(h_u) = de_{\mathcal{D}_f}(h_u)$ for some dominance tree $\mathcal{G}_{\mathcal{D}_f} = (V_{\mathcal{D}_f}, E_{\mathcal{D}_f})$. If $g_1, \ldots, g_m$ are any collection of nodes 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*

$$h_u \cup de_{\mathcal{G}}(h_u) \perp\!\!\!\perp \bigcup_{i=1}^{m} \{g_i \cup de_{\mathcal{G}}(g_i)\}. \tag{15}$$

If $\{g_1, \ldots, g_m\} \subseteq \{an_{\mathcal{G}}(h_u) \cap V_{\mathcal{D}_f}\}$ then

$$de_{\mathcal{G}}(h_u) \perp\!\!\!\perp \{G^\dagger \setminus h_u^\dagger\} | h_u, \tag{16}$$

where $G^\dagger \setminus h_u^\dagger = \{\bigcup_{i=1}^{m} \{g_i \cup de_{\mathcal{G}}(g_i)\}\} \setminus \{h_u \cup de_{\mathcal{G}}(h_u)\}$.

Theorem 3 enables us to identify subtrees on the dominance tree which are independent of subtrees where the two strong roots are not on a common path on the call graph. For example, in the first example of Subsection 2.3, the 'single call in' subtrees B^*, C^*, and D^* are also 'isolated' subtrees and so the independence of these subtrees may also be established via relation (15), whilst relation (16) may be used to deduce, for instance, that $\{B^* \setminus B000\} \perp\!\!\!\perp A000 | B000$. In the second example of Subsection 2.3, the 'single call in' subtree D^* is also an 'isolated' subtree but B^* and $C^* \setminus C110$ are not for they do not dominate their shared descendent $C110$.

Theorem 3 may be applied to any node h_u with the property that $de_{\mathcal{G}}(h_u) = de_{\mathcal{D}_f}(h_u)$; whether h_u is strongly directly dominated or only directly dominated is irrelevant. For example, in the third example of Subsection 2.3, we may check that $de_{\mathcal{G}}(C000) = de_{\mathcal{D}_f}(C000)$ and $de_{\mathcal{G}}(D000) = de_{\mathcal{D}_f}(D000)$. Thus, the D^* subtree does not call $C000$ and we have $C^* \perp\!\!\!\perp D^*$.

Assessing whether $de_{\mathcal{G}}(h_u) = de_{\mathcal{D}_f}(h_u)$ is simple, and we may choose to identify these subtrees on the dominance tree. This will provide us with more information about the calling structure without altering the layout of the dominance tree.

We may check that $de_{\mathcal{G}}(C000) = de_{\mathcal{D}_f}(C000)$ but we cannot tell from Figure 4 which nodes are responsible for the direct dominance of $C000$. Assessing whether $de_{\mathcal{G}}(h_u) = de_{\mathcal{D}_f}(h_u)$ is simple, and we may choose to identify these subtrees on the dominance tree. This will provide us with more information about the calling structure without altering the layout of the dominance tree. Firstly, it restricts the number of potential call graphs that could produce the dominance tree. In the second example of Subsection 2.3, adding the call $B000$ to $C110$ on Figure 1 means that neither $B000$ nor $C000$ satisfy the requirement for they do not dominate $C110$. Adding only the call $A000$ to $C110$ on Figure 1 means that $B000$ now dominates all of its descendents on the call graph: the dominance trees would no longer be the same. Secondly, these differences will indicate the ability of the dominance tree to represent more detailed information, for example the independence (and not conditional independence) of D^* in this example.

4.3 Modifying the dominance tree to highlight 'isolated' subtrees: the moral dominance tree

The 'single call in' subtrees are easy to identify on the dominance tree by the shading of the nodes, unshaded nodes representing strongly directly dominated nodes. 'single call in' subtrees have a strongly directly dominated node as the strong root. The node shading may be viewed as representing parental loss in the abstraction of the calling structure to create the dominance tree from the call graph. Strongly directly dominated nodes have not lost any parents, whilst directly dominated nodes had at least two parents on the call graph. We may modify the shading to indicate whether directly dominated nodes are dominated by one of their parents from the call graph or by an ancestor. This provides more information about the call graph without decreasing the simplicity of the dominance tree.

The 'isolated' subtrees on the dominance tree are those subtrees for which the strong root has not lost any descendents in the simplification of the calling structure. They are identified by establishing the truth of the equality $de_{\mathcal{G}}(h) = de_{\mathcal{D}_f}(h)$. This does not rely on any visual representation of the call graph. Notice that $ch_{\mathcal{G}}(h) = ch_{\mathcal{D}_f}(h)$ does not imply that $de_{\mathcal{G}}(h) = de_{\mathcal{D}_f}(h)$ but the converse is true. We may use shapes to identify 'isolated' subtrees by using different shaped nodes to represent those nodes for which $ch_{\mathcal{G}}(h) \neq ch_{\mathcal{D}_f}(h)$, $ch_{\mathcal{G}}(h) = ch_{\mathcal{D}_f}(h)$ only, or $de_{\mathcal{G}}(h) = de_{\mathcal{D}_f}(h)$.

We make the following amendments to Definition 4 to obtain a revised dominance tree, which we term the moral dominance tree.

Definition 10 *The moral dominance tree corresponding to a root node f is the graph $\mathcal{G}_{\mathcal{D}_f} = (\{f\} \cup de(f), E_{\mathcal{D}_f})$ formed from the call graph $\mathcal{G} = (V, E)$. For any two nodes g, $h \in de(f)$, $(g, h) \in E_{\mathcal{D}_f}$ if g either directly or strongly directly dominates h. If g strongly directly dominates h, then the node h is unshaded and h is shaded if g only directly dominates it. Two shadings are used to distinguish nodes directly dominated by one of their parents on $\mathcal{G}$ and those directly dominated by a non-parent. If $de_{\mathcal{G}}(h) = de_{\mathcal{D}_f}(h)$ then the node is a rectangular box with rounded corners. If only $ch_{\mathcal{G}}(h) = ch_{\mathcal{D}_f}(h)$ then the node is an ellipse. If neither of these occur, then the node is a rectangular box.*

The dominance tree of Definition 10 has the identical shape to Definition 4; we merely alter the shapes and shadings of the nodes. An example helps clarify this.

Example 3 revisited: Failure to isolate reuse candidates

This third example of Subsection 2.3 concerned the addition of a call between $B000$ and $C000$ to Figure 1. The dominance tree corresponding to Definition 4 is given in Figure 4; the moral dominance tree corresponding to Definition 10 is given in Figure 12.

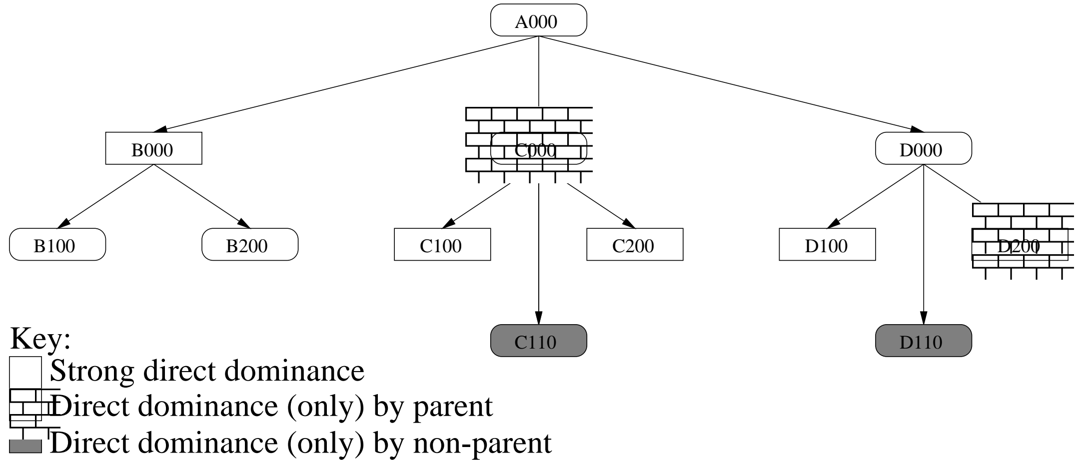


Figure 12: The moral dominance tree resulting from the call graph of Figure 1 with an additional call between $B000$ and $C000$ deduced from Definition 10.

Burd & Munro [2] argued that Figure 4 failed to isolate the reuse candidates for it is not clear from Figure 4 how C^* is accessed. The strong dominance of $C000$ means that $C^* \subset de_{\mathcal{G}}(B000) \cup de_{\mathcal{G}}(D000)$ but nothing further. The conditional independence relation captures this; we obtain from Corollary 1 and Theorem 3 that $D^* \perp\!\!\!\perp B^* \cup C^*$, $D^* \perp\!\!\!\perp B^*$ and

$D^* \perp\!\!\!\perp C^*$. Thus of the two candidate subtrees, headed by $B000$ and $D000$, which may access C^* we find that only B^* does. This information was automatic from the structure of the call graph and had no reliance on any visual representation. Observe how this is represented on Figure 12 by the node $D000$ being displayed in a rectangular box with rounded corners: we deduce that $de_{\mathcal{G}}(D000) = \{D100, D200, D110\}$ and D^* is thus, via Theorem 3, independent of all other nodes on the same or lower level on Figure 12. The shape of the boxes in the B^* subtree on Figure 12 reveal that the only member of B^* which calls $C000$ is $B000$ for $B000$ appears in a rectangular box. The conditional independence relation supports a reuse candidate of $B^* \cup C^*$ and this may be viewed on Figure 12. From Theorem 3 we may obtain the relationship that $\{B100, B200\} \perp\!\!\!\perp C^*$ allowing scope for comprehension within the reuse candidates that was not available from the dominance relation but is captured by a conditional independence relation.

The shape of the boxes surrounding the nodes help us to visualise whether the relationships between subtrees headed by strongly directly dominated nodes are conditional independences or independences; the latter being represented by rectangular boxes with rounded corners. The choice between ellipses and rectangular boxes help restrict the number of different call graphs that could yield the same dominance tree, the same is true with the modified shadings between Definition 4 and Definition 10. On Figure 12, we see that the node $C000$ is directly dominated by $A000$ and the shading indicates that $A000$ is a parent of $C000$ on the call graph. The node $C110$ is directly dominated by $C000$ but the lighter shading indicates that $C110$ was not called by $C000$ on the call graph. We could not have detected this on Figure 4. In fact, although this will not always be the case, only a single call graph can generate Figure 12.

4.4 Using the conditional independence relation to aide comprehension where the dominance relation fails

Example 4 revisited: The problem of multiple root nodes

The power of the conditional independence relation is that it may be used to assess the relationship between any collection of nodes. The fourth example of Subsection 2.3 concerned the problem of multiple root nodes. From Corollary 1, we identify that $B^* \perp\!\!\!\perp \{C^* \setminus C110\} | C110$ on Figure 5. Notice that although $B200$ and $C100$ share a parent on Figure 5, they are not married on the moral graph $\tilde{\mathcal{G}}_M(\{B000, C000\})$ since $C001$ is not a node on this moral graph. The dominance relation may only be considered in relation to

a root node and so provides no information about the relationship between different root nodes. However, we can do this with the conditional independence relation. For example, $A000 \perp\!\!\!\perp C001 | \{B200, C100\}$. We have a determinable relationship between the two root nodes, a feature that was not possible with the dominance relation.

Example 5 revisited: Failure to capture potential reuse candidates

The fifth example of Subsection 2.3 was given in Figure 7 and suggested that the dominance relations dependence upon single node separators rather than collection of nodes may limit its ability to identify potential reuse candidates. The visual representation of Figure 7 suggests three clear reuse candidates: B^*, $C^* \cup C001$, and D^*. However, this intuition was not supported by the dominance tree given in Figure 8. We may redraw the dominance tree in line with Definition 10; the resultant moral dominance tree is shown in Figure 13.

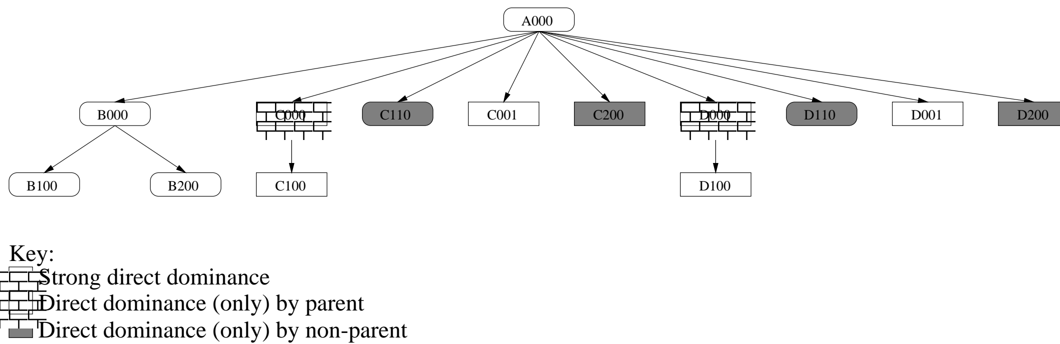


Figure 13: The moral dominance tree resulting from the call graph of Figure 7 deduced from Definition 10.

Notice that the strictly directly dominated node $B000$ is contained in a rectangular box with rounded corners. This illustrates that $B000$ dominates all of its descendents on Figure 7 and provides an immediate visual identification of the conditional independence statement $B^* \perp\!\!\!\perp \{C^* \cup C001 \cup D^* \cup D001\}$. Thus, Figure 13 draws our attention to the adoption of B^* as a reuse candidate completely separate of any other collection of nodes: once execution enters B^* it does not switch to any other node before exiting whilst it may only enter B^* via $B000$. This information could not have been deduced from inspection of Figure 8. Figure 13 illustrates that dominance trees derived from Definition 10 cannot always be mapped back to a unique call graph: the $C^* \cup C001$ and $D^* \cup D001$ decompose in an identical way on Figure 13 despite the difference in the calling structure between these two (there is a call between $D100$

and $D200$ but no call between $C100$ and $C200$). Figure 13 does provide us with guidance as to how to understand the remaining code. We may look first at the relationships between the subtrees headed by strongly directly dominated nodes, for we know from Corollary 1 that these are either conditionally independent or independent. This includes the subtrees consisting of just single nodes, namely $C001$ and $D001$. The conditional independence relation will home in on the relationship of these strongly directly dominated nodes with the directly dominated nodes. Applying Corollary 1 we find that $C^* \cup C001 \perp\!\!\!\perp \{B^* \cup D^* \cup D0001\}$ and $D^* \cup D001 \perp\!\!\!\perp \{B^* \cup C^* \cup C0001\}$. This provides an argument for the adoption of the three reuse candidates suggested by the visual representation of the call graph, but did not rely on the layout of Figure 7 and would have derived the identical results for a more confused layout of Figure 7. Observe how even though the dominance relation failed to identify the reuse candidates it did help the program comprehension process for it highlighted an area of code that was insufficiently explained by the dominance relation and was in need of further investigation. We understood B^* from Figure 13 but not the $C^* \cup C001 \cup D^* \cup D001$. We can immediately home in on these nodes and in particular the nodes $C001$ and $D001$ with the conditional independence relation.

The dominance relation fails in this example, Figure 7, because it looks for a single node separator. We could investigate further ways of modifying the dominance tree to detect this structure. For example, we noted that the dominance relation may be viewed as a graph separation property but graph separation may be applied to a collection of nodes rather than single nodes. Consider the collection of nodes $\{A000, D000, D001\}$ on Figure 7. If we ignore the direction of the arcs, we see that each of these nodes is joined, by an arc, to every other node in the collection. If we try to add any other node to this collection, for example $D200$, this is no longer the case as there is no arc between $A000$ and $D200$. The collection $\{A000, D000, D001\}$ is called a clique. We may identify a number of cliques on Figure 7. For instance the collection $\{A000, C000, C001\}$ or the collection $\{C000, C001, C200\}$. Cliques are central to the study of conditional independences for any node in a clique on the moral graph is always dependent upon the other nodes in the clique. If a collection of nodes on a dominance tree, $\mathcal{G}_{\mathcal{D}_f}$, form a clique on the call graph, $\mathcal{G}$, then they will be present in a clique on the associated moral graph, $\tilde{\mathcal{G}}_M(f)$, and so cannot be separated from one another.

In this paper, we have linked the results of the dominance relation with conditional independence statements. The dominance relation is good at identifying conditionally inde-

pendent areas of the code ('single call in' subtrees) but struggles to cope with dependencies in the call graph (mapping the relationships with the only directly dominated nodes). It may be unable to cope with cliques for the nodes are inseparable from other members of the clique. We could modify the dominance relation to handle cliques. Notice that since the call graph is acyclic, any clique on the call graph must contain a unique node which calls all the other nodes in the clique. We call this node the clique-parent. In the clique $\{A000, D000, D001\}$ the clique-parent is $A000$, whilst in the clique $\{C000, C001, C200\}$ the clique-parent is $C001$. For any clique, we may form the collection of nodes which are not the clique-parent. We shall denote this collection by G. We then seek the collection of nodes, H, for which for all $h \in H$, G is a (f, h)-separator. Compare this with the dominance relation: g dominates h if g is a (f, h)-separator. We merely expand the separator set in a formal way.

As an illustrative example, suppose we consider the cliques $\{A000,\ C000,\ C001\}$ and $\{A000,\ D000,\ D001\}$ on Figure 7. We find that for each $h_c \in \{C100,\ C200,\ C110\}$, $\{C000, C001\}$ is an $(A000, h_c)$-separator and that for each $h_d \in \{D100,\ D200,\ D110\}$, $\{D000, D001\}$ is an $(A000, h_d)$-separator. We may redraw Figure 13 as Figure 14 to represent these revised separations.

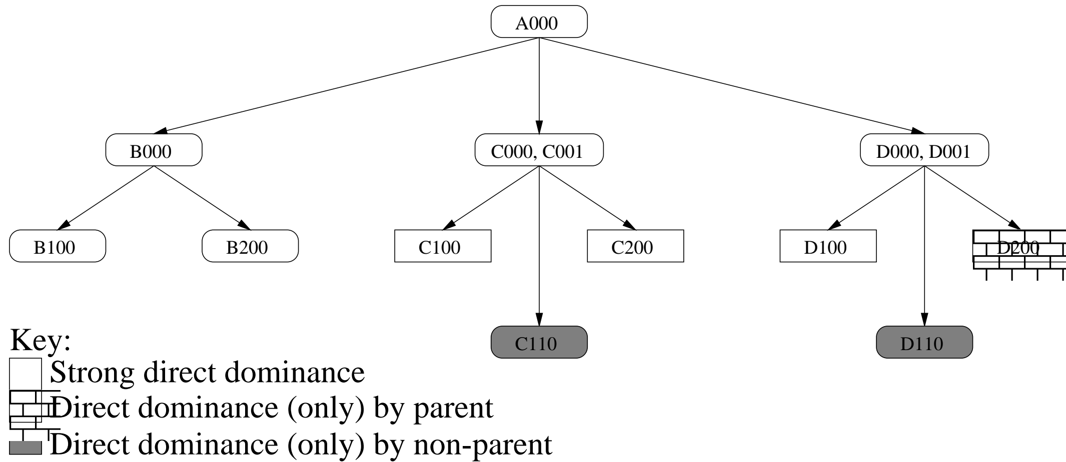


Figure 14: The moral dominance tree resulting from the call graph of Figure 7 deduced from Definition 10 and taking account of the cliques $\{A000, C000, C001\}$ and $\{A000, D000, D001\}$.

We consider that $\{C000, C001\}$ strongly directly dominates $C100$ and $C200$ for the only nodes that call these two nodes are contained in $\{C000, C001\}$. $C110$ is only directly dominated by $\{C000, C001\}$ for this node is called by nodes other than $\{C000, C001\}$. The similar sentiments may be applied to $D100$, $D200$ and $D110$. Notice the similarity of Figure

14 with the moral dominance tree that would be obtained from Figure 1 (compare Figure 2). Figure 14 shows three 'isolated' subtrees: B^*, $C^* \cup C001$ and $D^* \cup D001$ which are independent as our earlier analysis reveals. We could have dealt with further cliques on the call graph which would have given us additional information.

This suggestion of handling cliques in an automated way may provide a valuable tool in producing a dominance tree in situations where available reuse candidates, supported by a conditional independence relation, are not visualised. Figure 14 provides a much more preferable visualisation of the call graph than Figure 13.

5 Conclusion

The dominance tree analysis may be used to identify subtrees which may be considered as potential reuse candidates. The subtrees considered are those whose strong root is strongly directly dominated. We termed these 'single call in' subtrees. The dominance tree does not explain the relation between 'single call in' subtrees and nodes who are only directly dominated. We address this problem by introducing a generalised conditional independence 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 strong 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 strongly directly dominated nodes and directly dominated nodes. 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 is required. The conditional independence relation is a tool for investigating any collection of nodes in the software and so we are able to understand collections where the dominance relation exhibited a lack of understanding. Finally, we considered how the dominance relation could be improved so that it could handle cliques. Combining the dominance tree analysis with the conditional independence relation provides us with a more detailed understanding of the relationships within the calling structure and thus our level of

comprehension.

Acknowledgements

This work has been supported by grant GR\M76775 from EPSRC.

References

[1] E. Burd and M. Munro. A method for the identification of resuable units through the reengineering of legacy code. *The Journal of Systems and Software*, 44:121–134, 1998.

[2] E. Burd and M. Munro. Evaluating the Use of Dominance Trees for C and COBOL. *Proceedings of the International Conference on Software Maintenance, ISCM'99*, 00:401–410, 1999.

[3] E. Burd, M. Munro, and C. Wezeman. Analysing Large COBOL Programs: the extraction of reusable modules. *Proceedings of the International Conference on Software Maintenance, IEEE Press*, 1996.

[4] E. Burd, M. Munro, and C. Wezeman. Extracting Reusable Modules from Legacy Code: Considering issues of module granularity. *Proceedings of the 3rd Working Conference on Reverse EngineeringI, IEEE Press*, 1996.

[5] A. Cimitile, A. De Lucia, G. Di Lucca, and Fasolino A. Identifying Objects in Legacy Systems. *International Workshop on Program Comprehension, IEEE Press*, 1997.

[6] Robert. G. Cowell, A. Philip. Dawid, Steffan. L. Lauritzen, and David. J. Spiegelhalter. *Probabilistic Networks and Expert Systems.* Springer, New York, 1999.

[7] A. P. Dawid. Conditional independence in statistical theory (with discussion). *J. R. Statist. Soc. B*, 41:1–31, 1979.

[8] A. P. Dawid. Conditional independence for statistical operations. *Ann. Statist.*, 8:598–617, 1980.

[9] J-F. Girard and R. Koschke. Finding Components in a Hierarchy of Modules: a step towards architectural understanding. *International Conference on Software Maintenance, IEEE Press*, 1997.

[10] M Goldstein. Influence and Belief Adjustment. In R. M. Oliver and J. Q. Smith, editors, *Influence Diagrams, Belief Nets and Decision Analysis*, pages 143–174. Wiley, 1990.

[11] M. S. Hetch. *Flow Analysis of Computer Programs.* North-Holland, Amsterdam, 1977.

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

[13] Finn. V. Jensen. *An introduction to Bayesian networks.* University College London Press, London, 1996.

[14] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems (with discussion). *J. R. Statist. Soc. B*, 50:157–224, 1988.

[15] Steffan. L. Lauritzen. *Graphical Models.* Oxford Science Publications, Oxford, 1996.

[16] Judea Pearl. *Probabilistic Inference in Intelligent Systems.* Morgan Kaufman, San Mateo, California, 1988.

[17] J. Q. Smith. Influence diagrams foe statistical modelling. *Annals of Statistics*, 17:654–672, 1989.

[18] J. Q. Smith. Statistical Principles on Graphs. In R. M. Oliver and J. Q. Smith, editors, *Influence Diagrams, Belief Nets and Decision Analysis*, pages 89–120. Wiley, 1990.

Appendix I: Graph theory and notation

In this appendix, we briefly review graph theory and the notation used in the paper. The approach mirrors that of [15] and further details may be found there.

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. Thus, we shall consider graphs with no loops.

An edge $(f, g) \in E$ is said to be **directed**, denoted $f \to g$, if $(f, g) \in E \wedge (g, f) \notin E$. An edge $(f, g) \in E$ is said to be **undirected**, denoted $f \sim g$, if $(f, g) \in E \wedge (g, f) \in E$. If $(f, g) \notin E$, we write $f \not\to g$ and if $(f, g) \notin E \wedge (g, f) \notin E$, we write $f \not\sim g$.

The graph $\mathcal{G}$ is said to be **directed** if there is no $(f, g) \in E$ such that $f \sim g$. Similarly, $\mathcal{G}$ is said to be **undirected** if there is no $(f, g) \in E$ such that $f \to g$.

If $f \to g$ then f is said to be a **parent** of g and g is said to be a **child** of f. We denote by $pa_{\mathcal{G}}(g)$ the set of parents of g on $\mathcal{G}$, and $ch_{\mathcal{G}}(f)$ denotes the set of children of f on $\mathcal{G}$.

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 $g \mapsto f$ we say that f and g **connect** and write $f \rightleftharpoons g$. If either $f \mapsto g$ or $g \mapsto f$ we say that there is a **direct path** between f and g

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**.

The vertices f such that $f \mapsto g$ and $g \not\mapsto f$ are the **ancestors** of g on $\mathcal{G}$, denoted $an_{\mathcal{G}}(g)$. The vertices g such that $f \mapsto g$ and $g \not\mapsto f$ are the **descendents** of f on $\mathcal{G}$, denoted $de_{\mathcal{G}}(f)$.

A subset $H \subseteq V$ is said to be an (f, g)**-separator** if all paths from f to g intersect H. Thus, for a directed graph, for each path $f \mapsto g \ \exists \ h \in H$ such that $f \in an_{\mathcal{G}}(h)$ and $g \in de_{\mathcal{G}}(h)$. If F, G, H are non-overlapping subsets of V then H is said to **separate** F from G if H is an (f, g)-separator for every $f \in F$, $g \in G$

Appendix II: Proofs of theorems

Proof of Theorem 2 - For any node $h \in V$, let $h^\dagger = h \cup de_{\mathcal{G}}(h)$. We construct the associated moral graph $\tilde{\mathcal{G}}_M(h_i^\dagger \cup H^\dagger)$, where $H^\dagger = \bigcup_{k=1}^{l} h_{j_k}^\dagger$, and from Lemma 1, we need to consider separations on this graph. Notice that $V_M(h_i^\dagger \cup H^\dagger) = h_i^\dagger \cup H^\dagger$. Let $A = \bigcup_{k=1}^{l} \{de_{\mathcal{G}}(h_i) \cap de_{\mathcal{G}}(h_{j_k})\}$, $B = \bigcup_{k=1}^{l} \{de_{\mathcal{G}}(h_i) \cap de_{\mathcal{G}}^c(h_{j_k})\}$ and $C = \bigcup_{k=1}^{l} \{de_{\mathcal{G}}^c(h_i) \cap de_{\mathcal{G}}(h_{j_k})\}$; A, B, and C are mutually incompatible.

If $A = \emptyset$ then the subgraphs $h_i^\dagger$ and $H^\dagger$ are unconnected on $\mathcal{G}$ and thus on $\tilde{\mathcal{G}}$. If they are connected on $\tilde{\mathcal{G}}_M(h_i^\dagger \cup H^\dagger)$, the path must have been formed by the marriage of some $h_{i1} \in de_{\mathcal{G}}(h_i)$ and some $h_{*1} \in \bigcup_{k=1}^{l} de_{\mathcal{G}}(h_{j_k})$. Since $A = \emptyset$, $de_{\mathcal{G}}(h_i) = B$ and $\bigcup_{k=1}^{l} de_{\mathcal{G}}(h_{j_k}) = C$.

If $A \neq \emptyset$ then the subgraphs $h_1^\dagger$ and $H^\dagger$ are connected on $\mathcal{G}$ and thus on $\tilde{\mathcal{G}}$. The connecting nodes 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 equation (10), if there exists $h_2 \in h_i^\dagger \cup H^\dagger$ such that $\{(h_{i1}, h_2), (h_{*1}, h_2)\} \subseteq E_R$, or equivalently $\{(h_2, h_{i1}), (h_2, h_{*1})\} \subseteq E$. If $h_2 \in A$ then $\{h_{i1}, h_{*1}\} \subset A$: a contradiction. If $h_2 \in h_i^\dagger \setminus A$ then $h_{*1} \in de_{\mathcal{G}}(h_i)$: a contradiction. If $h_2 \in H^\dagger \setminus A$ then $h_{i1} \in \bigcup_{k=1}^{l} de_{\mathcal{G}}(h_{j_k})$:

a contradiction. Thus, there is no such $h_2 \in h_i^\dagger \cup H^\dagger$ and the results follow. □

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

If there is no direct path between each g_i and h_u then each $g_i \in V_{\mathcal{D}_f} \setminus h_u^\dagger$ and $h_u \notin de_{\mathcal{G}}(g_i)$. Thus, $g_i^\dagger \cap h_u^\dagger = \emptyset$ and the subgraphs $h_u^\dagger$ and $G^\dagger$ are unconnected on $\mathcal{G}$ and thus on $\tilde{\mathcal{G}}$. For them to be connected on $\tilde{\mathcal{G}}_M(h_u^\dagger \cup G^\dagger)$, the path must have been formed by the marriage of some $h_{u1} \in h_u^\dagger$ and some $g_{*1} \in G^\dagger$. We may show this cannot occur in an identical way to the proof of Theorem 2. Property (15) thus follows.

Property (16) also follows by observing that if each $g_i \in an_{\mathcal{G}} \cap V_{\mathcal{D}_f}$ then the subgraphs $h_u^\dagger$ and $G^\dagger$ are unconnected on $\mathcal{G}$ but only at h_u. Following the proof of Theorem 2 we show that there can be no marriage between some $h_{u1} \in h_u^\dagger$ and some $g_{*1} \in H^\dagger \setminus h_u^\dagger$. □