Moralising the call graph as a means of program comprehension

Simon C Shaw & Michael Goldstein
Department of Mathematical Sciences
University of Durham, UK

in collaboration with:

Elizabeth Burd & Malcolm Munro
Centre for Software Maintenance
University of Durham, UK

January 2001

1 Graph theory and notation

In this section, we briefly review graph theory and the notation we shall use. The approach mirrors that of Lauritzen (1996) 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 $pa(g)$ the set of parents of g, and $ch(f)$ denotes the set of children of f. If $f \sim g$, then f and g are said to be neighbours. The complete set of neighbours for f is denoted by $ne(f)$. If $f \not\sim g$ then f and g are said to be **non-adjacent**. We may make similar definitions for groups of vertices, F. Namely,

$$pa(F) = \cup_{f \in F} pa(f) \setminus F \quad (1)$$

$$ch(F) = \cup_{f \in F} ch(f) \setminus F \quad (2)$$

$$ne(F) = \cup_{f \in F} ne(f) \setminus F \quad (3)$$

Thus, for example, $pa(F)$ is the collection of parents of the vertices in F that are not themselves in F.

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 $f \rightleftharpoons g$.

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, denoted $an(g)$. The vertices g such that $f \mapsto g$ and $g \not\mapsto f$ are the **descendents** of f, denoted $de(f)$.

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 $f \in an(h)$ and $g \in de(h)$. 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 $f \in F$, $g \in G$

We want to use the graph as a visual aid and we draw it as follows. Each vertex is represented by a circle and we connect the circles associated with vertices f, g with a line if $f \sim g$. If $f \mapsto g$ then we join the associated circles by an arrow from f to g.

2 The call graph

We view a piece of software as working on a database and consisting of a number of procedures which may be called and which operate on the database. The calling structure of a piece of software provides a high level description of the flow of the program. It may be represented graphically by a call graph.

Definition 1 *A call graph is a directed acyclic graph (DAG), $\mathcal{G} = (V, E)$. 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.*

The call graph is thus a full representation of the calling structure of the program: the arcs represent potential calls as opposed to actual calls. It may be that later on in the comprehension process, we want to refine the graph, perhaps by constructing the call graph for subsets of inputs of interest. Any such graph will be a subgraph of $\mathcal{G}$. The code may need restructuring before the construction of the call graph. For example, to ensure that the graph is acyclic, every strongly connected subgraph is collapsed to a single vertex. Section 2 of Burd & Munro (1998) lists three conditions that must be satisfied by a piece of COBOL code before the call graph is formed.

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 $A000$.

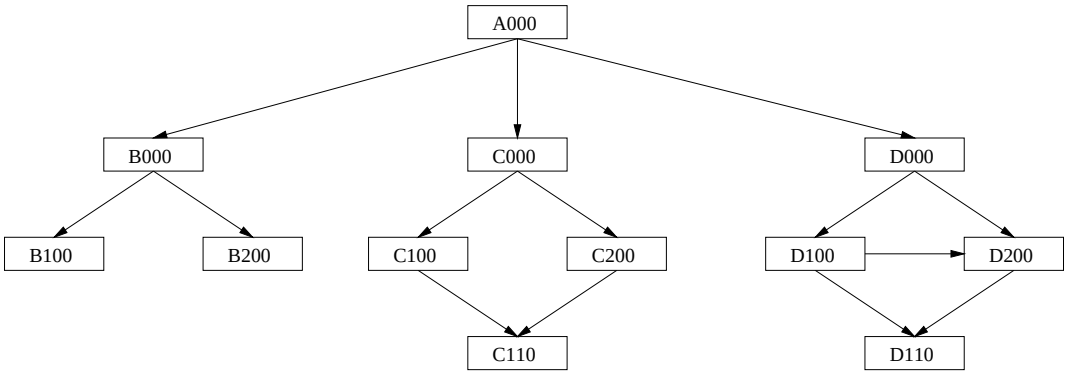

Figure 1: A simple call graph. Procedure $A000$ calls procedures $B000$, $C000$ and $D000$. Procedure $B000$ calls procedures $B100$ and $B200$ and so on.

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', $\{B000, B100, B200\}$; the 'C-code', $\{C000, C100, C200, C110\}$; and the 'D-code, $\{D000, D100, D200, D110\}$. Intuitively, it would seem that these three collections are unconnected and may be assessed separately. A software maintainer interested only in the 'B-code' need not understand the 'C-code' or the 'D-code' for once $B000$ has been called, the execution of the program exists purely in the 'B-code' 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 and constructing a directed tree called the dominance tree.

3 The dominance tree

The dominance tree aims to assist the program comprehension by reducing information overload during the early stages of comprehension but have the additional use of identifying sections of code which may be remodularised into a single module. The dominance tree provides a visualisation of the code that is at least as simple as the call graph for it is a directed tree over the collection of procedures on the call graph $\mathcal{G} = (V, E)$.

It is formed by using the relations of strong and direct dominance as identified by Hetch (1977).

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

Thus, we see that g dominates h if and only if g separates f from h on $\mathcal{G}$. Notice that this relationship is not symmetric since $\mathcal{G}$ is directed: there is no path from any procedure to f since f is a root node. Note also that for every $g \in de(f)$, f dominates g so that there exists at least one dominating node for each g. The direct dominance relation identifies, for each node, a single dominator from the collection of dominators of that node. To show this, suppose g and h dominate i but g doesn't dominate h and h doesn't dominate g. Then, there is a path $f \to g$ which doesn't intersect h and a path $g \to i$ which doesn't intersect h and so combining these two yields a path $f \to i$ which does not intersect h and so h is not a dominator of i: a contradiction. Further, if g directly dominates a node i then either $g \in pa(i)$ and g dominates h for all $h \in pa(i) \setminus \{g\}$ on $\mathcal{G}$ or $g \notin pa(i)$ and g dominates h for all $h \in pa(i)$.

Definition 4 *The 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.*

If $\mathcal{G}$ has a single root node, f, then $\{f\} \cup de(f) = V$ and the dominance tree $\mathcal{G}_{\mathcal{D}_f}$ includes all the procedures and we write $\mathcal{G}_{\mathcal{D}_f} = \mathcal{G}_{\mathcal{D}}$. The advantage of the dominance tree is that there is a single path from the root node to any other procedure on the dominance tree and so the number of arcs have been reduced from the call graph. The disadvantage is that if $\mathcal{G}$ has more than one root node then the dominance tree corresponding to a root node f will not contain all the procedures that may be called in the software. Moreover, the same procedures are likely to appear on different dominance trees and it may be hard to ascertain the relationships between different dominance trees. Burd & Munro (1998; 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.'

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, $\mathcal{G}_{\mathcal{D}}$. Notice that we shade the nodes which are only directly dominated. Thus, $D000$ strongly directly dominates $D100$, whilst $D110$ is only directly dominated by $D000$. Direct dominance corresponds to nodes that have become disinherited from their parents: they had at least two parents and

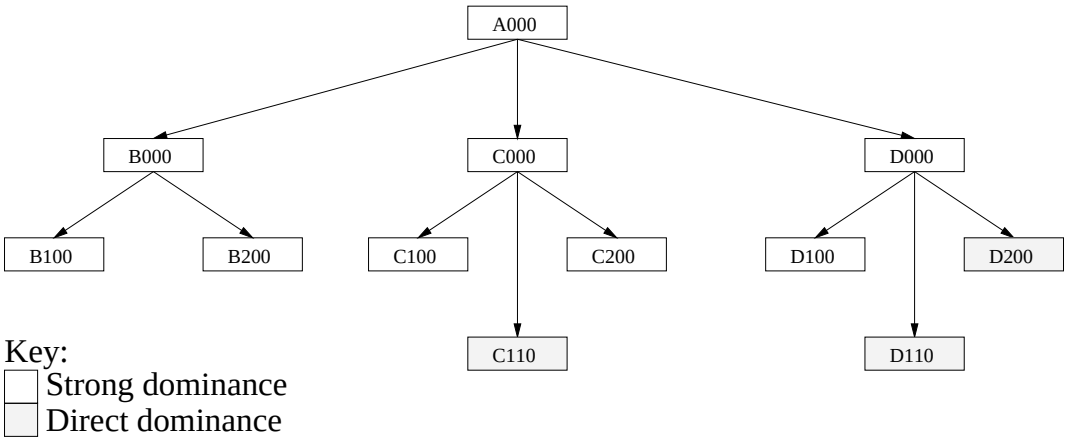

Figure 2: The dominance tree corresponding to the call graph of Figure 1.

may not be directly dominated by one of their parents. For example, in Figure 2, the node $D110$ is directly dominated by $D000$, but $D000$ is not a parent of $D110$. Thus, $E_{\mathcal{D}} \not\subseteq E$: the dominance tree is not merely the call graph with some edges removed.

Notice that the dominance tree is identical to the call graph if and only if the call graph is a tree. In this case, every node is strongly directly dominated. Where only directly dominated nodes are present on the dominance tree, disinheritance of that nodes parents has occurred. The directly dominated nodes 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. This may create problems in comprehension and intuitively, the greater the proportion of shaded nodes, the more problematic program comprehension may be from the dominance tree.

4 Identifying reuse candidates

One of the aims of the dominance tree is to use it as a means for identifying 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 (1999) 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'. For example, in Figure 2, $C110$ is called by both $C100$ and $C200$. Burd & Munro (1999) evaluates the use of dominance trees using two factors to assess the suitability of dominance trees. Firstly, that the results of the dominance tree analysis will identify candidates suitable for reconstruction into reuse candidates and secondly that the reuse candidates generated, and the restructuring performed, represents an improvement to the structure of the code.

The subtrees headed by the nodes $B000$, $C000$ and $D000$ have the property that the strong roots of these subtrees ($B000$, $C000$, and $D000$ respectively) are all strongly directly dominated by $A000$ and are the only nodes directly dominated by $B000$. As we highlighted in the description of the call graph, once $B000$ is called, execution can only take place in the 'B-code'. Likewise, once $C000$ is called, execution only takes place in the 'C-code' and having called $D000$, we confine ourselves to execution in the 'D-code' until $D000$ is exited. It is only after execution ceases at these strong roots that we may switch between the branches.

If a change happens to say procedure $C110$, then the ripple effects of this are blocked from the 'B-code' and the 'D-code' by the head of the 'C-code', $C000$. This intuition leads us to the following definition.

Definition 5 *Suppose $\mathcal{G} = (V, E)$ is a call graph and $f \in \mathcal{G}$ is a root node. If there is a node $g \in V_{\mathcal{D}_f}$ such that g strongly directly dominates h for each $h \in ch(g)$ on $\mathcal{G}_{\mathcal{D}_f}$, then the subtree of $\mathcal{G}_{\mathcal{D}_f}$ consisting of the nodes $\{h\} \cup de(h)$ and all arcs between them is a potential reuse candidate.*

If we apply this definition to Figure 2, then we may identify the collections $\{B000, B100, B200\}$, $\{C000, C100, C200, C110\}$ and $\{D000, D100, D200, D110\}$ as potential reuse candidates.

We now consider alternative call graphs and their associated dominance trees. We shall consider the effect of additional calls to Figure 1 in a similar vein to the analysis in Section 3.2 of Burd & Munro (1999).

Firstly, consider the addition of a call between the procedures $B000$ and $C110$. This creates a new path $A000 \rightarrow C110$ which does not pass through $C000$. If we remove $A000$ and all arcs from it on the call graph then we no longer get three separate subgraphs: the 'D-code' is separate but the call between $B000$ and $C110$ connects the 'B-code' with the 'C-code'. The additional call creates a path $A000 \rightarrow C110$ via $B000$ which does not pass through $C000$. Thus, $C110$ is no longer dominated by $C000$; it is directly dominated by $A000$. The resultant dominance tree is shown in Figure 3.

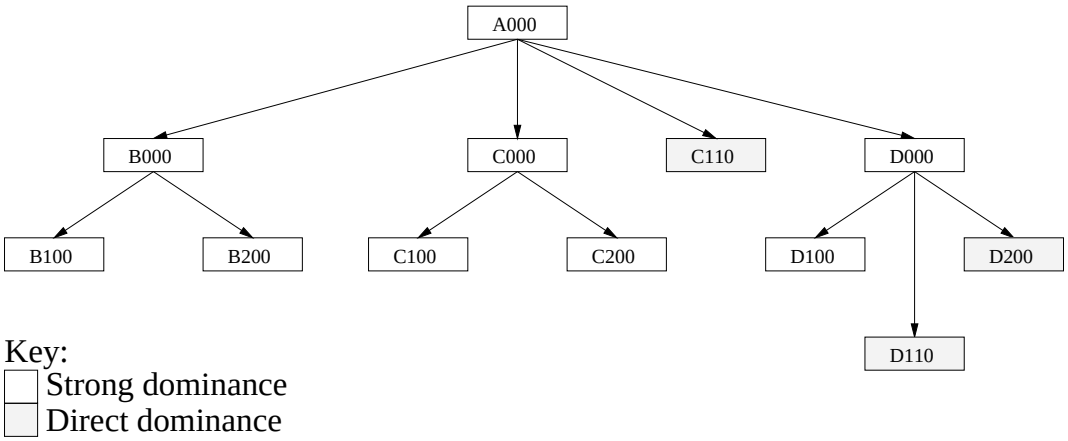

Figure 3: The dominance tree resulting from the call graph of Figure 1 with an additional call between $B000$ and $C110$.

How may we interpret reuse candidates on this dominance tree? The children of $A000$ are no longer all strongly directly dominated and so it would appear that Definition 5 no longer applies. In the 'B-code', execution only switches to the single procedure $C110$ in the 'C-code' and removing $C110$ and all edges connecting it from the call graph as well separates the 'B-code' from the remaining 'C-code'. If a change is made to $C110$, then the effects of this change may ripple either up the 'C-code' or to the 'B-code', but if we know there is no change made to $C110$ then we conceive the 'B-code' and the collection $\{C000, C100, C200\}$ as being separate. Burd & Munro (1999) argue that the collections $\{B000, B100, B200\}$, $\{C000, C100, C200\}$ and $\{D000, D100, D200, D110\}$ are potential reuse candidates and the node $C110$ is a service candidate; a service candidate being a single procedure that can

be accessed by one of more of the reuse candidates. This leads us to extend Definition 5 to the following.

Definition 6 *Suppose $\mathcal{G} = (V, E)$ is a call graph and $f \in \mathcal{G}$ is a root node. If there is a node $g \in V_{\mathcal{D}_f}$ such that g either strongly directly dominates h for each $h \in ch(g)$ on $\mathcal{G}_{\mathcal{D}_f}$, or any $h \in ch(g)$ which is only directly dominated has no children, then the subtree of $\mathcal{G}_{\mathcal{D}_f}$ consisting of the nodes $\{h\} \cup de(h)$ and all arcs between them is a potential reuse candidate if $de(h) \neq \emptyset$ and a service candidate if $de(h) = \emptyset$.*

Notice that the dominance tree does not exhibit how a service candidate is called by the other reuse candidates. This results in a loss of information which may make ripple effects hard to map.

Consider that instead of adding the arc $(B000, C110)$ to the call graph of Figure 1, we added the arc $(A000, C110)$. The only effect again concerns paths to $C110$: the dominance tree is identical to that shown in Figure 3, for there is no a path $A000 \rightarrow C110$ which passes through no other nodes. In this instance there is no direct link between the initial reuse sets $\{B000, B100, B200\}$ and $\{C000, C100, C200, C110\}$ and we may wish to consider the reuse candidates to be identical to those corresponding to the dominance tree of Figure 2. Thus, the addition of the arc $(A000, C110)$ to the call graph of Figure 1 makes no difference to the potential reuse candidates. These two differences are not apparent on the dominance tree itself and require further study of the call graph.

The effect of a change to $C110$ is also not apparent on the dominance tree. For the addition of the arc $(B000, C110)$ to the call graph of Figure 1 then as Burd & Munro point out 'ripple effects are additionally restricted to the two candidates in this case $B000$ and $C000$'. The fact that the 'D-code' does not call the reuse candidate $C110$ and so remains separate from the 'B-code' and the 'C-code' can only be detected by referring back to the call graph. Suppose instead, that to the call graph of Figure 1 we add calls $(\beta, C110)$ for each $\beta \in \{A000, B000, B100, B200, C000, D000, D100, D200, D110\}$, then every procedure now calls $C110$, but once more the resulting dominance tree is identical to that given in Figure 3. However, every node calls $C110$ and any change to $C110$ would immediately ripple across the whole graph. Adding only the arc $(A000, C110)$ to the call graph of Figure 1 sees the same dominance tree, but the ripple effects restricted to the candidate $C000$. This suggests that the dominance tree may not be a good vehicle for investigating ripple effects for whilst we know that calls will be made to $C110$, we do not know which of the reuse candidates call it.

As a second example, suppose that to the original call graph of Figure 1, we add the arc $(C000, D000)$. The result is that the node $D000$ is no longer strongly directly dominated by $A000$ since it is now called by more than one node. The resultant dominance tree for this scenario is given by Figure 4.

The consequence of the change is that upon execution of $C000$, we could execute $D000$ and so the sets $\{C000, C100, C200, C110\}$ and $\{D000, D100, D200, D110\}$ should not be regarded as separate reuse candidates. However, the relation between these candidates is not clear on the dominance tree and Burd & Munro point this out when considering the analogous scenario caused by adding a call between $B000$ and $C000$ on the call graph of Figure 1 by stating that 'the result of this change, unlike the first, could potentially indicate a failing of the dominance tree approach as a means of identification of reusable candidates. This is because, the approach has apparently failed to identify a possible connection between the two reuse candidates. Thus, this represents a failure to properly isolate candidates ... and therefore to control a ripple effect between them.' Again, it is possible to illustrate this more by adding calls to the call graph that don't alter the dominance tree. For example, the dominance tree of Figure 4 also results from adding the arcs $(B000, D000)$ and $(C000, D000)$

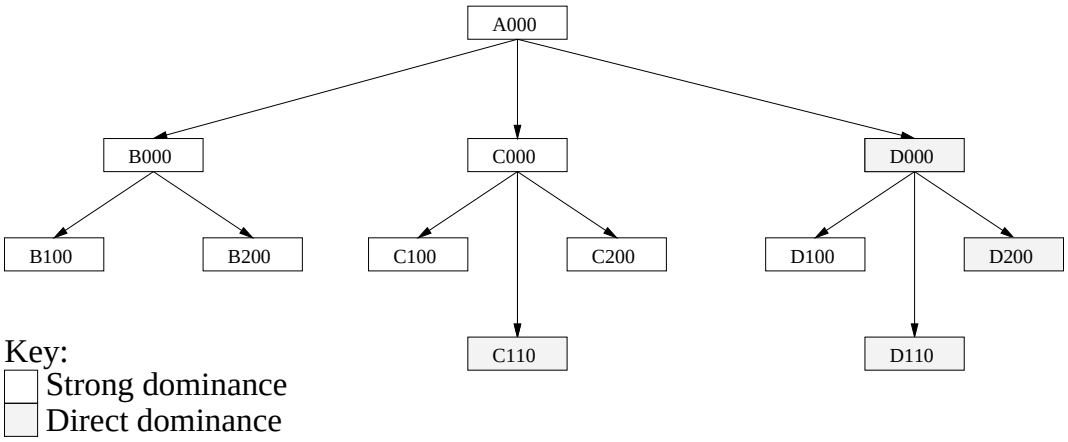

Figure 4: The dominance tree resulting from the call graph of Figure 1 with an additional call between $C000$ and $D000$.

to the call graph of Figure 1 so that the execution of the program may switch between each of the reuse candidates. Likewise, the addition of the arc $(C110, D000)$ to the call graph of Figure 1 also results in the dominance tree of Figure 4. In this case, might we consider using the sets $\{B000, B100, B200\}$, $\{C000, C100, C200\}$ and $\{D000, D100, D200, D110\}$ as reuse candidates?

5 Procedures and uncertainty

We view each procedure as having the following form. It processes an input in order to perform an action and the result of this action is then returned. The return is thus, minimally, a sign for the program to continue. We denote the procedure by f and the input by a. We remark here that a may be vector valued. Immediately prior to f being called by a, the database is in state D_a. The procedure is then called and the action performed. For the input a, the desired output is $f_{D_a}(a)$ and the desired state of the database after the performance of f on a is $D_{f(a)}$. Notice that the desired output is functionally dependent upon the current state of the database. The function should be able to handle the input according to any state of the database, and the output may depend upon the state of the database. A simple example may help to clarify this.

Suppose that we are considering the bank account details of customers of a bank. The database thus consists of a record for each customer. The fields of each record contain information such as the customers address and the balance of his account. Suppose that f is a procedure that takes two arguments, a_1 and a_2, as its input. a_1 denotes the name of an account holder and a_2 a numeric value. The aim of f is to add the amount a_2 to the account of a_1. The input is thus the two dimensional vector $a = (a_1, a_2)$ and the output $f_{D_a}(a)$ will be an error message if there is no account corresponding to a_1 in D_a; an error message if a_2 is not a numeric quantity; a message to continue if the account corresponding to a_1 has been found and a_2 is numeric. Thus, f should be able to handle the cases when a_1 is and isn't in the database and also when data is entered in the incorrect format. The desired state of the database after the execution of f, $D_{f(a)}$, is that it is equal to D_a if a_1 was not in D_a (or a_2 was incorrectly entered) and if a_1 is found, then the only change should be the addition of a_2 to the variable corresponding to a_1's account balance.

In reality, after the execution of f, the output is $\tilde{f}_{D_a}(a)$ and the state of the database is $D_{\tilde{f}(a)}$. 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 $D_{\tilde{f}(a)} = D_{f(a)}$.

In the example of the bank database, we may see that if we enter an account name for an account we expect to find in the database with a numerical quantity then we may see $\tilde{f}_{D_a}(a)$ return a message indicating we may proceed and we may be confident that $\tilde{f}_{D_a}(a) = f_{D_a}(a)$. However, if the action of f has merely been to add the sum to every existing account, then we would not have $D_{\tilde{f}(a)} = D_{f(a)}$. The procedure f has caused an error in the database which may only manifest itself later.

Definition 7 *The procedure f is said to work for input a if, for all possible database states, D_a, we have*

$$\tilde{f}_{D_a}(a) \;=\; f_{D_a}(a) \quad \textit{and} \quad D_{\tilde{f}(a)} \;=\; D_{f(a)}. \tag{4}$$

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, if in the bank record database, the procedure f to add a specified sum to the account of the account holder adds the amount to every account holder, then the database is in error. However, if the procedure, g which checks the account balance for a specified account holder correctly gives the balance held in the database for an alternative recognised account holder then this procedure is not in error, although the returned sum is not the correct balance.

For each procedure f there is a set of possible inputs, A. We make the following definition.

Definition 8 *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.*

Thus, we may attach to each procedure a well defined random quantity, namely the 'Procedure working?' quantity which has two possible states, 1 if the procedure works and 0 if the procedure does not work. Thus, attached to each procedure f *in* V, the collection of procedures, is the random quantity *f-working*? If we learn about the state of a random quantity *f-working*? this may enable us to gain information about the state of a further random quantity *g-working*? for an alternative procedure $g \in V$. Intuitively, a random quantity X is judged to be independent of Y if knowledge about Y does not affect the uncertainty about X: there is no influence between X and Y. X is judged to be dependent of Y if knowledge about Y does affect the uncertainty about X. In a similar way, a notion of conditional independence can be established. How does this fit into our aims for program comprehension and the call graph? Consider the simple example we gave in Figure 1. We noted that once a call had been made to $D000$, the calls are restricted to the set $\{D100, D200, D110\}$ until execution finishes and the procedure $D000$ is exited and then execution may switch to any other section of the program. Intuitively, we could argue that if we know the state of $D000$-*working*? then there is nothing further we could learn about $A000$-*working*? from learning about the collection $\{D100$-*working*?, $D200$-*working*?, $D110$-*working*?$\}$ since if there is an error in any of these procedures it could only possibly propagate to $A000$ via $D000$. The value to program comprehension is that for a given collection of procedures of interest we may determine which, if any, procedures they are independent of, when the independence is considered for the associated random quantities. The most familiar notion of independence and conditional independence is that concerning probability distributions.

6 Probabilistic Conditional Independence

Suppose that X and Y are random quantities; a random quantity being any quantity whose value is currently unknown to us. Note that X and Y may themselves be vector valued and so could be considered as collections of random quantities. We assume that we are able to attach to each random quantity a probability density function $p(\cdot)$. Thus, $p(x, y)$ denotes the joint density of (X, Y), and the marginal densities of X and Y are denoted, respectively, by $p(x)$ and $p(y)$. denote, respectively, the marginal densities of X and Y. Adopting the notation of Dawid (1979), we write $X \perp\!\!\!\perp Y$ to denote that X and Y are probabilistically independent. In terms of the density functions, this requires that

$$p(x, y) \quad = \quad p(x)p(y). \tag{5}$$

Equation (5) expresses the notion $X \perp\!\!\!\perp Y$ as a property of the probability densities, which are numerical quantities. However, this quantitative statement can be viewed in an intuitive, qualitative sense. We may interpret $X \perp\!\!\!\perp Y$ as meaning that any information we receive about Y does not alter our uncertainty about X. For example, if X is 'the score on a single roll of a die' and Y is 'the number of heads obtained in 2 tosses of a coin' then learning about the value of Y does not alter our beliefs about X. An equivalent expression to equation (5) of probabilistic independence which better captures this intuition is that $X \perp\!\!\!\perp Y$ requires that

$$p(x|y) \quad = \quad p(x), \tag{6}$$

where $p(x|y)$ is the conditional density of X given $Y = y$.

This notion of independence and the use of conditional densities leads us to extend the notion to more than two random quantities. For random quantities X, Y, Z (again, possibly vector valued), we say that X is conditionally independent of Y given Z, written $X \perp\!\!\!\perp Y|Z$ if

$$p(x, y|z) \quad = \quad p(x|z)p(y|z). \tag{7}$$

Equation (7) reduces to equation (5) in the case where $Z = \emptyset$, the empty set. Thus, as Smith (1989) points out, we may write $X \perp\!\!\!\perp Y$ as a shorthand for $X \perp\!\!\!\perp Y|\emptyset$. For general Z, we may once more understand $X \perp\!\!\!\perp Y|Z$ intuitively. Having observed Z, any information we learn about Y does not alter our beliefs about X. An equivalent representation of equation (7) quantifies this intuition:

$$p(x|y, z) \quad = \quad p(x|z). \tag{8}$$

We have thus defined conditional independence in terms of equalities on the probability density functions which suggests the property to be a quantitative feature of these density functions, identified only by testing the equality of an equation such as (8). Pearl (1988; p79) calls this the 'most striking inadequacy of traditional theories of probability' arguing that 'people tend to judge the three-place relationship of conditional dependency (i.e. X influences Y, given Z) with clarity, conviction, and consistency ... the notions of relevance and dependence are far more basic to human reasoning than the numerical values attached to probability judgements'. We will later argue that we are able to make such basic statements of dependencies for the random quantities relating whether procedures in the call graph are working. Dawid (1979, 1980) was the first to treat 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.' Treating

conditional independence as a basic quantity is also supported by Smith (1989; p656) who 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.'

Smith (1989) is interested in a generalised conditional independence property as opposed to the probabilistic conditional independence property we have discussed up until this point. He highlights three properties of probabilistic conditional independence given by Dawid (1979) and extends it to any tertiary property $(\cdot \perp\!\!\!\perp \cdot | \cdot)$ on collections of objects which obeys the three properties. For any collections W, X, Y, Z, we have

$$1.\, W \perp\!\!\!\perp X | (X \cup Y); \tag{9}$$

$$2.\, W \perp\!\!\!\perp X | Y \text{ if and only if } X \perp\!\!\!\perp W | Y; \tag{10}$$

$$3.\, W \perp\!\!\!\perp (X \cup Y) | Z \text{ implies and is implied by the pair of conditions } \begin{cases} W \perp\!\!\!\perp Y | Z; \\ W \perp\!\!\!\perp X | (Y \cup Z). \end{cases} \tag{11}$$

Equation (9) 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 (10) is the symmetry relation: 'if once Y is known, W is uninformative for X, then X is uninformative for W, having observed Y.' The final property, equation (11), 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.'

Smith (1989; 1990) shows that any tertiary relation obeying these properties will behave computationally as a conditional independence property. Hence, we may construct the 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.

For example, Goldstein (1981, 1990, 1994, 1999) has long advocated that there is a need to develop statistical methods based upon partial prior specifications to enable the handling of problems where the level of complexity means that it is beyond our ability or inclination to construct a full probability specification. Goldstein (1990) constructs a tertiary property based on the orthogonality of certain adjusted belief structures (see Goldstein (1986, 1988) for further details).

The advantage of this generalised conditional independence property is that provided properties (9) - (11) are satisfied for a set of random quantities, as Smith (1989; p656) writes 'all the theory we develop about how "information" is transferred between those uncertain quantities will also hold'. At the outset, we may only be willing to specify our qualitative judgments of independence but through the use of a generalised conditional independence property we need not restrict ourselves to requiring a full probabilistic either at the outset or at all. We may focus attention on making independence statements, the very thing both Smith and Pearl argue we are willing and able to do. Thus, the aim of Pearl (1988; p81) as to 'whether assertions equivalent to those made about probabilistic dependencies can be derived *logically* without reference to numerical quantities' may be accomplished if such a system exists for probabilistic conditional independencies. The solution to this lies in graphical representations and, in particular, Bayesian belief networks and the associated moral graphs. Construction of the Bayesian belief network allows us to represent graphically certain independencies present in the full probability distribution over all the random quantities

of interest. The moral graph uses the Bayesian belief network to provide us with a logical and tractable way to identify further independence statements between the collections of random quantities as we now explain.

7 Graphical representations: Bayesian belief network

A collection of conditional independence properties may be represented graphically. The nodes of the graph are random quantities. Nodes are joined by directed arrows if there is a possible direct dependency between the nodes. The direction of the arc represents the causal influence, for as Whittaker (1990; p71) writes 'in many, if not most, studies of several interacting variables there is a striking lack of symmetry in the roles played by variables that corresponds to a notion of causality'. For example, there is a possible direct dependency between an individual having lung cancer and that individual being a smoker, smoking being the possible cause of the cancer. Pearl (1988; p116) gives a similar example: 'if the sound of a bell is functionally determined by the outcomes of two coins, we will have the network $coin1 \rightarrow bell \leftarrow coin2$, without connecting $coin1$ to $coin2$. This network reflects the natural perception of causal influences; the arrows indicate that the sound of the bell is determined by the coin outcomes, which are mutually independent'. The most familiar graphical representation of a collection of conditional independence properties is the Bayesian belief network.

Definition 9 *Given a probability distribution $p(x_1, \ldots, x_n)$ and any ordering d of the variables, a Bayesian belief network of $p(x_1, \ldots, x_n)$ is the DAG $\mathcal{G} = (V, E)$ with node set $V = \{X_1, \ldots, X_n\}$. The set of vertices E is created by designating as parents of X_i, for each i, any minimal set $pa(X_i) \subseteq \{X_1, \ldots, X_{i-1}\}$ of ancestors satisfying*

$$p(x_i|pa(x_i)) \quad = \quad p(x_i|x_1, \ldots, x_{i-1}). \tag{12}$$

The immediate question arises of whether we can identify when a DAG is a Bayesian belief network. The following lemma, see Pearl (1988; p120), answers this question.

Lemma 1 *Given a probability distribution $p(x_1, \ldots, x_n)$ and a DAG $\mathcal{G} = (V, E)$ where $V = \{X_1, \ldots, X_n\}$, a necessary and sufficient condition for $\mathcal{G}$ to be a Bayesian belief network of $p(x_1, \ldots, x_n)$ is that each variable X_i is conditionally independent of all its non-descendents given its parents $pa(X_i)$, and that no proper subset of $pa(X_i)$ satisfy this condition.*

The Bayesian belief network thus represents the independencies embedded in $p(x_1, \ldots, x_n)$ that follow from the definition of the parent sets. On the other hand, inspection of the Bayesian belief network allows us to immediately reconstruct $p(x_1, \ldots, x_n)$ as

$$p(x_1, \ldots, x_n) \quad = \quad \prod_{i=0}^{n-1} p(x_{n-i}|x_{n-i-1}, \ldots, x_1) \tag{13}$$

$$\quad = \quad \prod_{i=1}^{n} p(x_i|pa(x_i)). \tag{14}$$

For further details on this see Jensen (1996; p20).

Having represented $p(x_1, \ldots, x_n)$ as a Bayesian belief network, we are interested in how it can be used for deducing new independence relationships from those explicitly used to

construct the DAG. The way this is achieved is by linking probabilistic conditional independence with graph separation on an associated undirected graph; graph separation satisfies the conditions (9) - (11) (see Pearl (1988; Section 3.1)) and so acts as a generalised conditional independence property. An undirected graphical model is said to have the global Markov property if whenever a collection of nodes Y separates collections W and X on the graph then $W \perp\!\!\!\perp X|Y$; see Cowell *et al.* (1999; p67) for further details on the global Markov property. If we are interested in assessing whether $W \perp\!\!\!\perp X|Y$ for our Bayesian belief network we construct an undirected graph called the moral graph.

Definition 10 *On the DAG $\mathcal{G} = (V, E)$ for subsets $W \subseteq V$, $X \subseteq V$, $Y \subseteq V$, the moral graph $\mathcal{G}_M(W, X, Y) = (V_M(W, X, Y), E_M(W, X, Y))$ is the undirected graph where*

$$V_M(W, X, Y) = \{W\} \cup an(W) \cup \{X\} \cup an(X) \cup \{Y\} \cup an(Y); \quad (15)$$

$$E_M(W, X, Y) = \{\{(f, g), (g, f) \forall g, f \in V_M(W, X, Y) : (f, g) \in E\} \cup \{(g, h), (h, g) \quad (16)$$

$$\forall f, g, h \in V_M(W, X, Y) : \{(g, f), (h, f)\} \subseteq E \wedge (g, h), (h, g) \notin E\}\}. \quad (17)$$

If $V_M(W, X, Y) = V$ then we write $\mathcal{G}_M(W, X, Y) = \mathcal{G}_M$ and term this the full moral graph.

Less formally, we draw the subgraph of $\mathcal{G}$ with nodes W, X, Y and all their ancestors; 'marry' all parents (join them with an edge if not already joined); drop all arrows to form the moral graph $\mathcal{G}_M(W, X, Y)$. For further details on moral graphs see Lauritzen & Spiegelhalter (1988).

The moral graph satisfies the global Markov property; see Lauritzen *et al.* (1990; Section 6). The following theorem, see Cowell *et al.* (1999; p71), is then used to determine the conditional independence properties of our Bayesian belief network.

Theorem 1 *For any three collections of nodes W, X, Y within a Bayesian belief network $\mathcal{G} = (V, E)$, construct the moral graph $\mathcal{G}_M(W, X, Y)$. Then $W \perp\!\!\!\perp X|Y$ whenever W and X are separated by Y on the moral graph.*

8 Viewing the call graph as a Bayesian belief network

In the previous section, we reviewed the use of a Bayesian belief network and its associated moral graph to exhibit conditional independencies present in the specification of the probability distribution $p(x_1, \ldots, x_n)$ over the nodes. However, as was discussed in Section 6, our independence and dependence statements are more basic than the specification of $p(x_1, \ldots, x_n)$. As Pearl (1988; p79) writes, 'the language used for representing probabilistic information should allow assertions about dependency relationships to be expressed qualitatively, directly, and explicitly ... once asserted, these dependency relationships should remain a part of the representation scheme, impervious to variations in numerical inputs'.

The representation comes from constructing a DAG as previously, but the parent sets, $pa(X_i)$, are now specified directly by the modeller as opposed to being a feature of a quantified probability distribution, $p(x_1, \ldots, x_n)$. Pearl (1988; p23) writes about how this should be done:

> The parents of X_i are those variables judged to be *direct causes* of X_i or to have *direct influence* on X_i. The informal notions of causation and influence replace the formal notion of directional conditional independence. An important feature of the network representation is that it permits people to express directly the fundamental, qualitative relationships of direct influence; the network augments these with derived relationships of *indirect influence* and preserves them, even if the numerical assignments are just sloppy estimates.

Later, we may seek to quantify the numerical assignments, and notice that by the generalised conditional independence property of Smith (1989), we may choose this using any system that satisfies the three properties (9) - (11). For example, we could elect to make a full probability specification, resulting in the Bayesian belief network discussed in the previous section. An alternative involves only a second order specification. This leads to a Bayes linear graphical model as described in Goldstein & Wilkinson (2000; Section 2.3).

We now reach the stage when we discuss how a graphical representation of conditional independence between procedures in our software may help program comprehension. Smith (1990) argues that there are three uses of graphical representations of conditional independence statements. Firstly, to enable quick and efficient calculations of marginal and conditional distributions of interest from $p(x_1, \ldots, p_n)$. This is done through local computation as expressed in Lauritzen & Spiegelhalter (1988). Goldstein & Wilkinson (2000) consider the principles of local computations when using a partial prior specification. The second to help in the elicitation of a model structure. If we constructed a graphical model over a piece of software and we wished to then test the software, we may wish to go down this route. It is the third use that we are interested in in this paper. Smith (1990; p90) writes

> The third use of graphical methods, which is linked but distinct from the second, 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. Since this analysis works only on the conditional independence statements in a model, the results obtained will be completely distribution-free.

We shall consider the propagation of errors in the call graph and use this as a means of deriving independencies between sets of random quantities *f-working*? for each procedure $f \in V$ on a call graph $\mathcal{G} = (V, E)$. These sets enable us to track the potential propagation of errors in the code and hence map the ripple effects.

Before we explore the dependence between the random quantities *f-working*? for each procedure $f \in V$ on a call graph $\mathcal{G} = (V, E)$, we make the following definition to link dependencies and independencies in the random quantities with the associated procedures.

Definition 11 *Suppose* $F = \{f_1, \ldots, f_l\}$, $G = \{g_1, \ldots, g_m\}$, *and* $H = \{h_1, \ldots, h_n\}$ *are three subsets of* V *where* V *is the collection of procedures of a call graph* $\mathcal{G} = (V, E)$. *The collection* G *is said to be procedurally conditionally independent of* H *given* F, *written* $G \perp\!\!\!\perp H|F$, *if for the collections* F*-working*? $= \{f_1$*-working*?$, \ldots f_l$*-working*?$\}$, G*-working*? $= \{g_1$*-working*?$, \ldots g_m$*-working*?$\}$, *and* H*-working*? $= \{h_1$*-working*?$, \ldots h_n$*-working*?$\}$ *we have* G*-working*?$\perp\!\!\!\perp H$*-working*?$|F$*-working*? *for a chosen generalised conditional independence property.*

Having made this definition, we now explore how we may use the call graph to suggest dependencies and independencies between collections of procedures. We make it explicit here that we are concerned with tracing the possible path of errors in the call graph, that is if a procedure f on the call graph is found to be in error for some input x, to which other procedures could this error be propagated and so affect our judgments of whether a second procedure g is in error in light of this information. We are ruling out other forms of relationships we could construct in the code. 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. We thus restrict our attention to dependencies that we can derive from the call graph, an object that is readily available.

Definition 12 *For a call graph $\mathcal{G} = (V, E)$, with f, $g \in V$, we judge g-working? to be a direct cause of f-working? if and only if $(f, g) \in E$*

We shall motivate this definition through three examples.

In the first case, consider a simple call graph $\mathcal{G} = (V, E)$ where $V = \{f, g, h\}$ and $E = \{(g, f), (f, h)\}$. This is illustrated as the left hand figure of Figure 5. Suppose that the

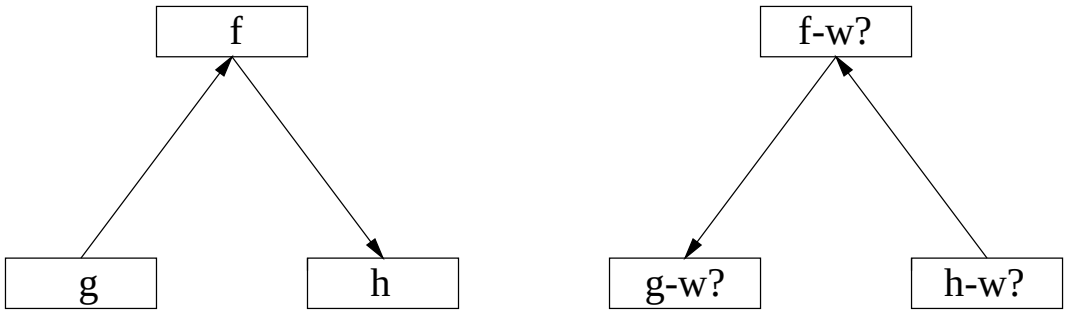

Figure 5: Left, a simple call graph with three nodes. If an error is detected in a child node, it could propagate to the parent node. This direction of propagation is exhibited by reversing the arcs in the call graph as shown in the figure on the right, where f-?, g-?, h-? represent, respectively, the random quantities *f-working?*, *g-working?* and *h-working?*. The procedures g and h are dependent, but if we know f then the procedures are independent.

procedure h does not work, so that the state of its associated random quantity *h-working?* is 0. There is at least one possible input for which h either yields the incorrect output or causes a mistake in the database. Suppose we call procedure f with input a. This input may cause procedure h to be called with input $b(a)$, the input being dependent upon the initial input a into f. It is possible that $b(a)$ may be an input that yields the error in h, so that when h returns to f, an error is present and so procedure f is in error. The error has directly propagated from h to f; the cause of an error in f could be h. We could view this as there being a potential physical propagation of an error from h to f. However, the reverse is not true since the execution of h must cease before the execution of f. Procedures f and h are dependent: knowledge of an error in h will cause us to increase our belief that f is in error for f may call h with the input that yields the error, whilst if f is in error, than we may judge that this error may be caused by a procedure it calls, namely h, which propagates the error. There is a dependence between f and h and we may reflect this by creating a new graph with a link between f and h; the propagation argument suggests this arc should be directed from h to f, the reverse of the direction of the arc on the call graph. A similar argument shows that g and f should be viewed as dependent. An error in h can propagate to g but not directly but via f. The dependence is indirect and if f is known, h and g are independent, that is *g-working?* $\perp\!\!\!\perp$ *h-working?* $|$ *f-working?* The moral graph of the right hand figure captures this for it is simply the figure with the arrows dropped. There is a single path from *g-working?* to *h-working?* and that is through *f-working?*.

In the second case, consider the call graph $\mathcal{G} = (V, E)$ where $V = \{f, g, h\}$ and $E = \{(f, g), (f, h)\}$; the call graph is shown as the left hand figure of Figure 6. The procedures

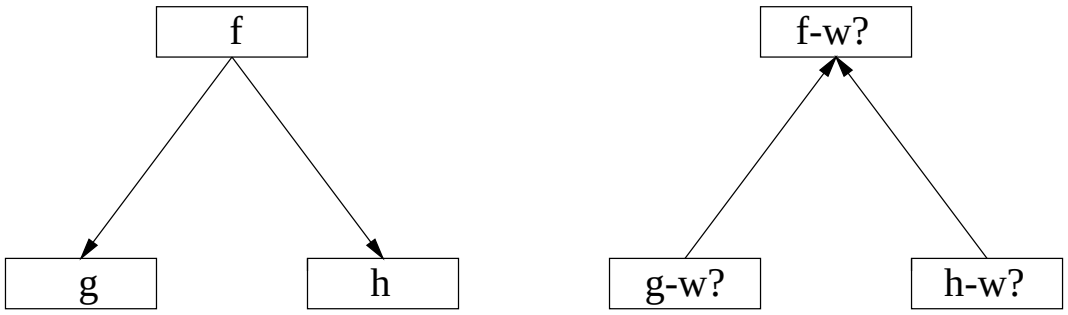

Figure 6: Left, a simple call graph with three nodes. The procedures g and h are independent, but if we know f then the procedures are dependent. The call graph with the arrows reversed, right, captures this.

g and h are independent. If g is not working, this gives us no information about the state of the *h-working*? variable. 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 conditionally 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 6, for random quantities *g-working*? and *h-working*? are parents of *f-working*? on this graph and are not joined by an arrow, so we marry them. This creates a path from *g-working*? to *h-working*? which does not intersect *f-working*?.

As a third case, we consider the call graph $\mathcal{G} = (V, E)$ where $V = \{f, g, h\}$ and $E = \{(g, f), (h, f)\}$, as shown as the left hand figure of Figure 7. The procedures g and h are

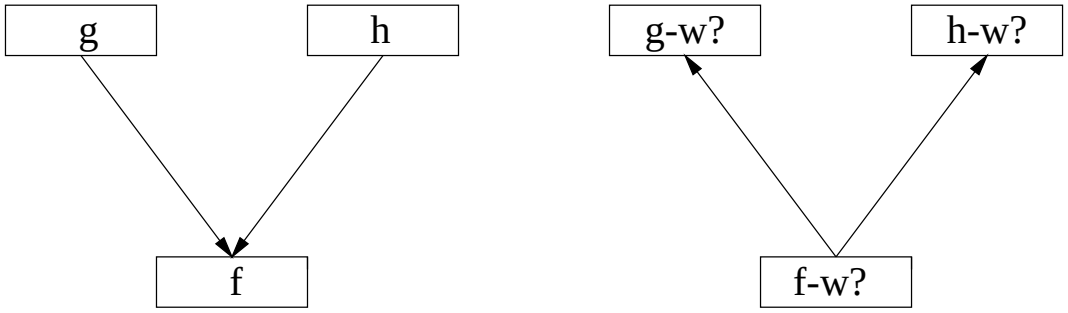

Figure 7: Left, a simple call graph with three nodes. The procedures g and h are dependent, but if we know f then the procedures are independent. The call graph with the arrows reversed, right, captures this.

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 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-working?*.

The right hand figures of Figures 5 - 7 illustrate the direct dependencies on the call graphs shown as the respective left hand figures. Definition 12 enables us to identify for each random quantity *f-working?* for each $f \in V$ on the call graph the parent sets; these are precisely the child sets of f on the call graph but considered as random quantities.

Definition 13 *The qualitative conditional independence network for the call graph* $\mathcal{G} = (V, E)$ *is the DAG* $\tilde{\mathcal{G}} = (V\text{-}working?, E_R)$ *where*

$$V\text{-}working? = \{f\text{-}working? : f \in V\}; \tag{18}$$

$$E_R = \{(g\text{-}working?, f\text{-}working?) : (f, g) \in E\}. \tag{19}$$

If G, H, F *are three sets of procedures on* $\mathcal{G}$, *then* G *is procedurally conditionally independent of* H *given* F, *written* $G \perp\!\!\!\perp H|F$ *if* F*-working? separates* G*-working? from* H*-working? on the moral graph* $\tilde{\mathcal{G}}_M(G\text{-}working?, H\text{-}working?, F\text{-}working?)$.

The qualitative conditional independence network for the call graph $\mathcal{G} = (V, E)$ thus maps the potential physical propagation of errors and the moral graph pinpoints the conditional independencies. As we have a one-to-one mapping between the collection of procedures V and the collection of random quantities V*-working?* with $f \mapsto f$*-working?*, for simplicity, we relabel the random quantities on the qualitative independence network as f rather than *f-working?*. Whilst also saving space, this allows us to visualise the qualitative conditional independence network as the call graph with the arrows reversed.

9 Conditional independencies around a root node

Suppose we have a DAG $\mathcal{G} = (V, E)$ and the node $f \in V$ is a root node. That is there is no $g \in V$ such that $(g, f) \in E$. Form the collections

$$A_f = f \cup ch(f), \tag{20}$$

$$B_f = \bigcup_{g \in V \setminus A_f} \{g \cup ch(g) \cup pa(g) \cup ch(pa(g))\}. \tag{21}$$

Notice that when considered on the associated moral graph $\tilde{\mathcal{G}}_M$, the collection of nodes B_f may be considered as

$$B_f = \bigcup_{g \in V \setminus A_f} \{g \cup bl(g)\}, \tag{22}$$

where $bl(g)$ denotes the Markov blanket of g. The Markov blanket is the set of neighbours of g on the moral graph $\tilde{\mathcal{G}}_M$, as can be easily checked. For further details on the Markov blanket see Cowell *et al.* (1999; p71). Notice that $f \notin B_f$.

Lemma 2 *Consider the call graph* $\mathcal{G} = (V, E)$ *and let* $f \in V$ *be a root node. Form the collection*

$$A_f = f \cup ch(f). \tag{23}$$

If $A_f \neq V$ *form the collection*

$$B_f = \bigcup_{g \in V \setminus A_f} \{g \cup bl(g)\}. \tag{24}$$

Then $\{A_f \cap B_f^c\} \perp\!\!\!\perp \{A_f^c \cap B_f\} | \{A_f \cap B_f\}$.

Proof - Since the moral graph, $\tilde{\mathcal{G}}_M$ is second-order global Markov then to show $\{A_f \cap B_f^c\} \perp\!\!\!\perp \{A_f^c \cap B_f\} | \{A_f \cap B_f\}$, we merely need that $\{A_f \cap B_f\}$ separates $\{A_f \cap B_f^c\}$ and $\{A_f^c \cap B_f\}$ on $\tilde{\mathcal{G}}_M$. Since $\{A_f^c \cap B_f\} = V \setminus A_f$, we have for each $g \in \{A_f^c \cap B_f\}$, $bl(g) \subset B_f$. Since $A_f \neq V$ and $\{A_f \cap B_f^c\} \neq \emptyset$, consider an element $f_A \in \{A_f \cap B_f^c\}$ and an element $g_B \in \{A_f^c \cap B_f\}$ and suppose there is a path $f_A \rightarrow g_B$. If there is no path, then the sets $\{A_f \cap B_f^c\}$, $\{A_f^c \cap B_f\}$ are separate on the moral graph and $\{A_f \cap B_f\} = \emptyset$: the two sets are independent. If there is a path, then for every path of this type, there is a node $f_A^* \in A_f$ and a node $g_B^* \in V \setminus A_f$ such that the edge (f_A^*, g_B^*) is present on the moral graph, $\tilde{\mathcal{G}}_M$. Hence, $f_A^* \in bl(g_B^*)$ and hence $f_A^* \in \{A_f \cap B_f\}$.

Having determined $\{A_f \cap B_f^c\}$, we may divide it into three sets, $\{f\}$, R_f and NR_f, where:

$$R_f = \{f_R \in A_f : \{ch(f_R) = \emptyset\} \cap \{pa(f_R) \setminus \{f\} = \emptyset\}\}, \tag{25}$$

$$NR_f = \{A_f \cap B_f^c\} \setminus \{f \cup R_f\}. \tag{26}$$

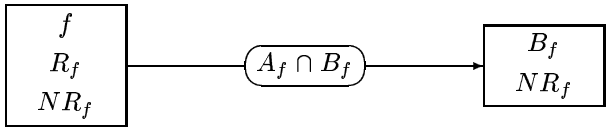

Figure 8: Dividing the nodes in V into two collections of nodes, $A_f \cap B_f^c$ and $B_f \cup NR_f$. The oval box represents a separator and contains the nodes $A_f \cap B_f$. Notice that $A_f = \{A_f \cap B_f^c\} \cup \{A_f \cap B_f\}$ and that $\{A_f \cap B_f^c\} \perp\!\!\!\perp \{A_f^c \cap B_f\} | \{A_f \cap B_f\}$

Suppose that $A_f \neq V$ and $\{A_f \cap B_f\} \neq \emptyset$. Then Figure 8 shows a possible graphical representation of these sets, and the derived conditional independence. We choose a suitable notation so that the sets $\{f\}$, R_f and NR_f are identifiable on the graphic. For example, we may always write f first and asterisk the nodes in NR_f. Thus, the graph is represented as two nodes and a separator.

If $A_f = V$, then the associated moral graph, $\tilde{\mathcal{G}}_M$, is complete; that is all the vertices are joined by an (undirected) arc. We may still represent this graphically by regarding B_f as being empty. Figure 9 shows the representation.

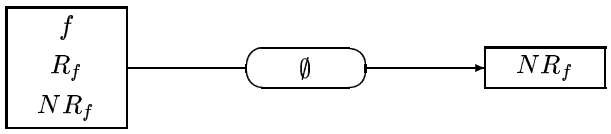

Figure 9: The representation when $A_f = V$. The associated moral graph is complete and there are no nodes that we can separate f from.

The third case to consider is when $A_f \neq V$ but $\{A_f \cap B_f\} = \emptyset$. In this case there are unconnected subgraphs within the graph, and at least two root nodes. If we considered these subgraphs separately, then the subgraph with root node f would have its associated moral graph as being complete, yielding the identical situation as shown in Figure 9. The remaining nodes, $A_f^c \cap B_f$ are completely separate. A graphical summary of this circumstance is given in Figure 10. Since the nodes in the set B_f form at least one graph themselves, we could form conditional independencies around the root nodes here. However, it is not just in this case

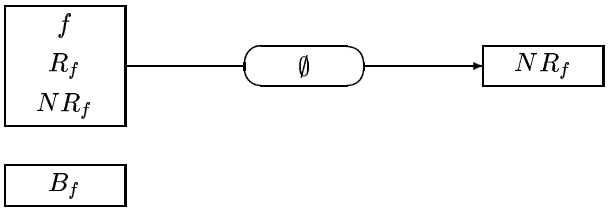

Figure 10: The representation when $A_f = V$. The associated moral graph is complete and there are no nodes that we can separate f from.

that we may have obvious further conditional independencies to investigate in an identical manner.

Returning to the graph $\mathcal{G}$, we may construct the associated moral graph $\tilde{\mathcal{G}}_M(V \setminus \{f \cup R_f\})$. Notice that it does not include the nodes $\{f \cup R_f\}$ within its set of nodes, indeed the moral graph $\tilde{\mathcal{G}}_M(V \setminus \{f \cup R_f\})$ is identical to the full moral graph formed from the call graph $\mathcal{G} \setminus \{f \cup R_f\} = (V \setminus \{f \cup R_f\}, E \setminus E_f)$, where E_f is the set of edges from f on $\mathcal{G}$, namely:

$$E_f \quad = \quad \{(f, h) \; \forall h \in ch(f)\}. \tag{27}$$

This observation thus shows the usefulness of splitting $A_f \cap B_f^c$ into the three collections $\{f\}$, R_f, NR_f. We could perform a similar independence check with a root node of the call graph $\mathcal{G} \setminus \{f \cup R_f\}$. Since this call graph has fewer nodes, but shares a moral graph with the original call graph $\mathcal{G}$ this suggests a possible algorithm for producing conditional independencies from the call graph $\mathcal{G}$. We derive such an algorithm in the next section.

10 An algorithm for deriving conditional independencies from a call graph

Suppose that $\mathcal{G} = (V, E)$ is a call graph and that $|V| = n$. Then it is always possible to well-order the nodes. That is there is a numbering such that if two nodes are connected, the arc between the node runs from the lower to the higher of the two nodes. Cowell *et al* (1999; p47) give an algorithm to construct a well-ordering of the nodes. We repeat the algorithm here.

Algorithm 1 *For the graph $\mathcal{G} = (V, E)$, we may well-order the nodes as follows.*

- *Begin with all nodes unnumbered.*
- *Set counter $i = 1$.*
- *While any nodes remain:*
  - *Select any root node.*
  - *Number the selected node as i.*
  - *Delete the node and all edges from the node to its' children.*
  - *Set $i = i + 1$.*

Thus, we can construct a bijective mapping $a : V \to \{1, \ldots, n\}$ so that if $f \in V$, $a(f) = k$ for some $k \in \{1, \ldots, n\}$. We may derive from the call graph $\mathcal{G} = (V, E)$ a series of conditional independence statements, similar to those derived in the previous section. The following algorithm explains how we do so.

Algorithm 2 *For the call graph $\mathcal{G} = (V, E)$, with $|V| = n$ and mapping $a : V \to \{1, \ldots, n\}$, form the $1 \times n$ vector nodesplit $= (0\ 0\ \ldots\ 0)$.*

- *Set $\mathcal{G}_1 = (V_1, E_1)$, where $V_1 = V$ and $E_1 = E$.*
- *Set $i = 1$.*
- *While $|V_i| \neq 0$ do:*
  - *We work on the graph $\mathcal{G}_i = (V_i, E_i)$.*
  - *If $f \in V_i$ is a root node, with $a(f) = k$, set*

$$nodesplit[k] \quad = \quad nodesplit[k] + 1, \tag{28}$$

and form the set

$$RN_i \quad = \quad \{f \in V_i : pa(f) = \emptyset\}. \tag{29}$$

  - *Choose any node $f_i \in RN_i$ such that*

$$nodesplit[a(f_i)] \quad = \quad \min_{f \in RN_i} nodesplit[a(f)], \tag{30}$$

and if $nodesplit[a(f_i)] > 1$, and $A_{f_{i-1}} \cap B_{f_{i-1}} \neq \emptyset$ then there exists $\gamma \in A_{f_{i-1}} \cap B_{f_{i-1}}$ such that $\gamma \in de(f_i)$.

  - *Construct the sets*

$$A_{f_i} \quad = \quad f_i \cup ch(f_i); \tag{31}$$

$$B_{f_i} \quad = \quad \bigcup_{g \in V_i \setminus A_{f_i}} \{g \cup ch(g) \cup pa(g) \cup ch(pa(g))\}. \tag{32}$$

  - *If $A_{f_i} \neq V_i$, we have the conditional independence statement that*

$$\{A_{f_i} \cap B^c_{f_i}\} \perp\!\!\!\perp \{A^c_{f_i} \cap B_{f_i}\} | \{A_{f_i} \cap B_{f_i}\}. \tag{33}$$

  - *Construct the sets*

$$R_{f_i} \quad = \quad \{f_R \in A_{f_i} : \{ch(f_R = \emptyset\} \cap \{pa(f_R \setminus \{f_i\} = \emptyset\}\}; \tag{34}$$

$$NR_{f_i} \quad = \quad \{A_{f_i} \cap B^c_{f_i}\} \setminus \{f_i \cup R_{f_i}\}. \tag{35}$$

  - *Construct the graph $\mathcal{G}_{i+1} = (V_{i+1}, E_{i+1})$ where*

$$V_{i+1} \quad = \quad V_i \setminus \{f_i \cup R_{f_i}\}; \tag{36}$$

$$E_{i+1} \quad = \quad E_i \setminus E_{f_i}, \tag{37}$$

where E_{f_i} is the set of edges from f_i on $\mathcal{G}_i$, namely

$$E_{f_i} \quad = \quad \{(f_i, h)\ \forall h \in ch(f_i)\}. \tag{38}$$

  - *Set $i = i + 1$.*

Having run this algorithm, we may use a graphical representation to illustrate the sets we calculate during the operation of the algorithm. The graphical display builds upon those given in Figures 8 - 10. We'll investigate in detail the first two stages of the graphical construction, before illustrating, by means of an example, the full representation and its' features.

10.1 Building a graphical representation of the results of the algorithm: first stage

We have our call graph $\mathcal{G} = (V, E)$. Suppose that at the first stage of the algorithm, we discover root nodes $f_{11}, \ldots, f_{m1}$ and decide to use f_{11} as our root node of interest in the algorithm. Suppose that $A_{f_{11}} \neq V$ and $\{A_{f_{11}} \cap B_{f_{11}}\} \neq \emptyset$. Thus, we have a situation similar to that illustrated in Figure 8. We could however have chosen any of the f_{j1} as the root node we split upon in the algorithm and so we choose to make there availability explicit. Figure 11 shows how this would be constructed.

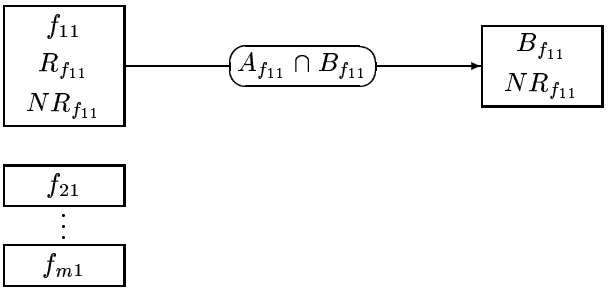

Figure 11: The available root nodes are $f_{11}, \ldots, f_{m1}$ and we show these on the left hand side to illustrate with which other nodes they were first available for splitting upon. The graph should be read as $\{A_{f_{11}} \cap B^c_{f_{11}}\} \perp\!\!\!\perp \{A^c_{f_{11}} \cap B_{f_{11}}\} | \{A_{f_{11}} \cap B_{f_{11}}\}$. Notice that for each $j = 2, \ldots, m$, $f_{j1} \in \{A^c_{f_{11}} \cap B_{f_{11}}\}$. In later stages, the nodes f_{j1} will be replaced by the relevant $A_{f_{j1}}$, but their positioning directly beneath $A_{f_{11}}$ will remain in place.

We then follow the second stage of the algorithm. We shall illustrate two cases in order to make the layout of our graphical representation more explicit.

10.2 Building a graphical representation of the results of the algorithm: second stage

Having performed the first split upon $\mathcal{G} = (V, E)$ with the root node f_{11}, for the second split we work with the graph $\mathcal{G} \setminus \{f_{11} \cup R_{f_{11}}\} = (V \setminus \{f_{11} \cup R_{f_{11}}\}, E \setminus E_{f_{11}})$. Notice that f_{21}, $\ldots$, f_{m1} are also root nodes on this graph. There are two cases to consider. Firstly that $\mathcal{G} \setminus \{f_{11} \cup R_{f_{11}}\}$ has additional root nodes $f_{12}, \ldots, f_{p2}$ as well as the remaining $f_{21}, \ldots, f_{m1}$. The second scenario is that $\mathcal{G} \setminus \{f_{11} \cup R_{f_{11}}\}$ has no other root nodes other than $f_{21}, \ldots, f_{m1}$.

In the first case, the algorithm determines that we will choose to split upon one of the f_{j2} nodes since these are newly available so that $nodesplit[a(f_{j2})] = 1$ for each $j = 1, \ldots, p$, whilst $nodesplit[a(f_{j1})] = 1$ for each $j = 2, \ldots, m$. Notice that, for each $j = 1, \ldots, p$, $f_{j2} \in A_{f_{11}}$. The algorithm thus states that we have a free choice of the f_{j2} to split upon. Suppose that we choose f_{12} and that $A_{f_{12}} \neq \{V \setminus \{f_{11} \cup R_{f_{11}}\}$, and $\{A_{f_{12}} \cap B_{f_{12}}\} \neq \emptyset$. We build upon Figure 11 by including the information from the second run of the algorithm. This extension is shown in Figure 12.

$A_{f_{11}} \cap B^c_{f_{11}} = \{f_{11} \cup R_{f_{11}} \cup NR_{f_{11}}\}$ and $A_{f_{12}} \cap B^c_{f_{12}} = \{f_{12} \cup R_{f_{12}} \cup NR_{f_{12}}\}$ are readily read off the graphic, alongside their corresponding separators, $A_{f_{11}} \cap B_{f_{11}}$ and $A_{f_{12}} \cap B_{f_{12}}$ respectively. $B_{f_{12}}$ may be read off and

$$B_{f_{11}} \quad = \quad \{f_{12} \cup R_{f_{12}} \cup NR_{f_{12}} \cup B_{f_{12}}\} \setminus NR_{f_{11}}, \tag{39}$$

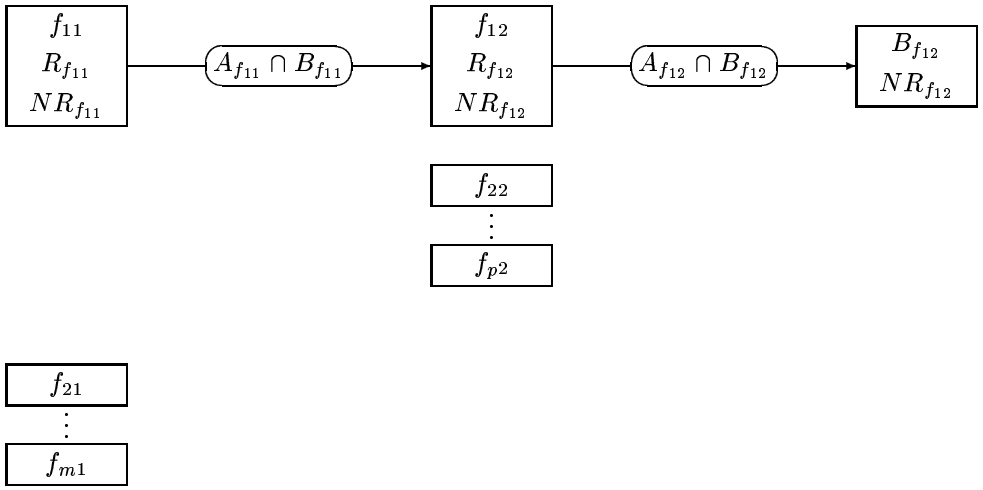

Figure 12: For the second step, the root nodes $f_{12}, \ldots, f_{p2}$ are additionally available and we illustrate where these became available. They are to the right of the f_{j1} for $j = 2, \ldots, m$ as they became available for splitting later. After splitting on f_{12}, the displayed root nodes, waiting for splitting, are all members of $\{A^c_{f_{12}} \cap B_{f_{12}}\}$.

or, all the nodes on the directed path from the node $A_{f_{11}}$ $B^c_{f_{11}}$ which are not in this node. The two basic conditional independence statements we may read off are:

$$\{A_{f_{11}} \cap B^c_{f_{11}}\} \perp\!\!\!\perp \{A^c_{f_{11}} \cap B_{f_{11}}\} | \{A_{f_{11}} \cap B_{f_{11}}\}; \tag{40}$$

$$\{A_{f_{12}} \cap B^c_{f_{12}}\} \perp\!\!\!\perp \{A^c_{f_{12}} \cap B_{f_{12}}\} | \{A_{f_{12}} \cap B_{f_{12}}\}. \tag{41}$$

Using the properties of conditional independence, we may combine these. For example, we have that

$$\{A_{f_{11}} \cap B^c_{f_{11}}\} \perp\!\!\!\perp \{A^c_{f_{12}} \cap B_{f_{12}}\} | \{\{A_{f_{11}} \cap B_{f_{11}}\} \cup \{A_{f_{12}} \cap B_{f_{12}}\}\}. \tag{42}$$

A close inspection of the algorithm shows that in this case, we will only now select one of the f_{j1} root nodes, for some $j = 2, \ldots, m$, for splitting after we have split upon all of the f_{k2} nodes.

The second scenario is that on the second stage of the algorithm, no new root nodes become available. We will then split upon one of $f_{21}, \ldots, f_{m1}$. We choose f_{21}. Suppose that $A_{f_{21}} \neq \{V \setminus \{f_{11} \cup R_{f_{11}}\}\}$, and $\{A_{f_{21}} \cap B_{f_{21}}\} \neq \emptyset$. This we add to Figure 11 as shown in Figure 13. We may then proceed through the stages of the algorithm in a similar way. The resulting construction is termed a moral tree. We now show its construction for a simple example; for spacial reasons the graph is drawn from north to south rather than the west to east direction of Figures 11 - 13.

11 Moral trees: an example

Recall Figure 1. We constructed the resulting dominance tree for the call graph; this was displayed in Figure 2. As an alternative approach, we construct the corresponding moral graph for Figure 1 using the algorithm. The moral tree is displayed in Figure 14.

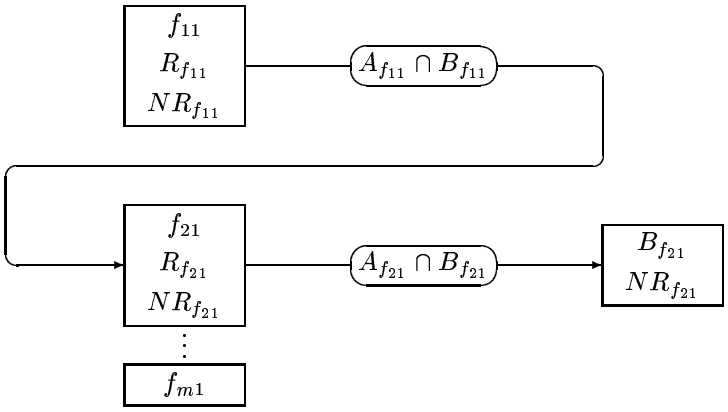

Figure 13: Following the split on f_{11}, no additional root nodes become available. The second split is on an f_{j1} for some $j = 2, \ldots, m$. We choose f_{21}. Notice how the node $A_{f_{21}}$ assumes the same position as the node f_{21} on Figure 11.

From Figure 14, we can see how the algorithm developed. At the first stage, we operate with the call graph as given in Figure 1. It has a solitary root node, $A000$ and we split upon this node. $A000$ and the arcs $(A000, B000)$, $(A000, C000)$ and $(A000, D000)$ are then removed from the call graph and we proceed with the remaining nodes and arcs as our graph in the algorithm. This has three root nodes: $B000$, $C000$ and $D000$ available for splitting upon. These three nodes are all placed on the second step of the graphic. We are free to choose to split on any of these. Since these nodes are all root nodes of unconnected graphs, the choice makes no difference to the graphical representation. Observe the node containing the set $\{B000,\ B100,\ B200\}$. The root node is $B000$ and $\{B100, B200\}$ $= R_{B000}$: the nodes have a single parent in $B000$ and no children. However, if we observe the node $\{D100, D200, D110\}$ we see that an asterisk appears alongside $D200$ and $D110$. This identifies that $\{D200,\ D110\} = NR_{D100}$: at this stage in the algorithm, $D200$ and $D110$ still have additional parents or children, which are also children of $D100$. As the node $\{D200,\ D110\}$ confirms, $D110$ is a child of $D200$ on the call graph, $\mathcal{G}$. This relationship is identifiable on the moral tree and needs no recourse to the call graph. Indeed, we may observe that there is a node that contains every node and its children and that these are identifiable. For example, the children of $A000$ are $B000$, $C000$ and $D000$. Since $A000$ is the only root node of Figure 1 and $A000$ appears in isolation in a node, then its children (which thus all appear in the adjoining separator) all have children that are not also children of $A000$. We don't split upon either $B100$, $B200$, $C110$ or $D110$: we know that these are childless nodes. Notice that after splitting upon $A000$, we observe that the graph splits into three unconnected subgraphs. Thus, we may immediately read the following (interesting) conditional independencies from the graph:

1. $\{A000\} \perp\!\!\!\perp \{B100, B200, C100, C200, C110, D100, D200, D110\} | \{B000, C000, D000\}$ (43)
2. $\{B000, B100, B200\} \perp\!\!\!\perp \{C000, C100, C200, C110\} | \emptyset;$ (44)
3. $\{B000, B100, B200\} \perp\!\!\!\perp \{D000, D100, D200, D110\} | \emptyset;$ (45)
4. $\{C000, C100, C200, C110\} \perp\!\!\!\perp \{D000, D100, D200, D110\} | \emptyset.$ (46)

Notice that the sets in equations (44) - (46) were the sets suggested by Burd & Munro (1999) as reuse candidates from inspection of the dominance tree (see Figure 2). Indeed, in

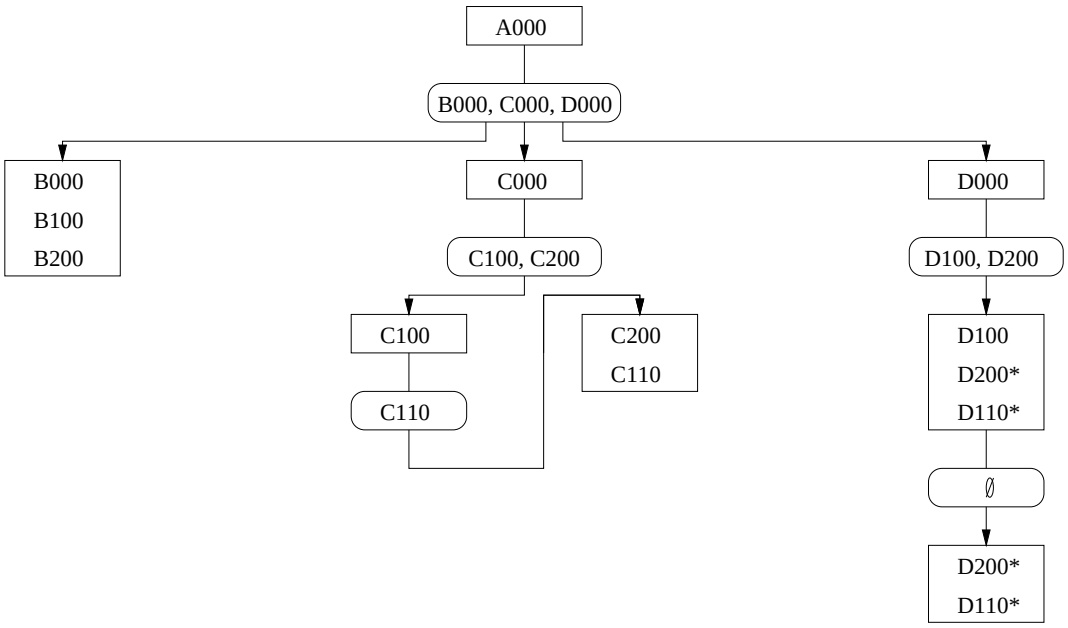

Figure 14: The moral tree resulting from the call graph of Figure 1.

shape, Figure 14 is very similar to Figure 2. It is straightforward to read off the dominance tree from Figure 14. Strongly directly dominated nodes will appear in one node on the moral tree, and at most one separator. If they appear in a separator then they must be the root node of the node they appear in on the moral tree to be strongly directly dominated. They are strongly directly dominated by the split upon node that the separator corresponds to. Hence, we immediately see that $B000$, $C000$, $D000$, $C100$, $C200$ and $D100$ are strongly directly dominated as they appear in exactly one node (as root nodes) on the moral graph and one separator. The dominators are, respectively, $A000$, $A000$, $A000$, $C000$, $C000$ and $D000$. $A000$, $B100$, $B200$ only appear in a solitary node and so are strongly directly dominated by the root node of that node; namely $A000$, $B000$ and $B000$ respectively.

$C110$ appears in exactly one node and one separator on the moral tree, but it is not the root node. Thus, it is not strongly directly dominated: it has more than one parent. Its parents are $C100$ and $C200$, which are both strongly directly dominated by $C000$ which thus directly dominates $C110$. $D200$ appears in two nodes and one separator on the moral tree. Its parents are $D000$ and $D100$ and $D000$ strongly directly dominates $D100$ and thus directly dominates $D200$. $D110$ appears in two nodes, having parents $D100$ and $D200$. These are both directly dominated by $D000$ ($D100$ strongly so) and so $D000$ directly dominates $D110$. Notice how we may visually see this directly dominators on the graph. Recall that a forward step (across a non-empty separator) on the moral tree constitutes a path on the call graph and the backward step a return to the head of a new path, then the arrow from the $C110$ separator to the node $\{C200, C110\}$ may be interpreted as an arrow from the $\{C100, C200\}$ separator to the $\{C200, C110\}$ node. The direct dominator of $C110$ is then the root-node in the node on the moral graph which is the root of these two branches on the moral tree, namely $C000$.

Suppose that we consider adding a call between $B000$ and $C110$ on the call graph of Figure 1. This results in a new moral tree as shown in Figure 15. The corresponding

dominance tree to this scenario is that shown in Figure 3.

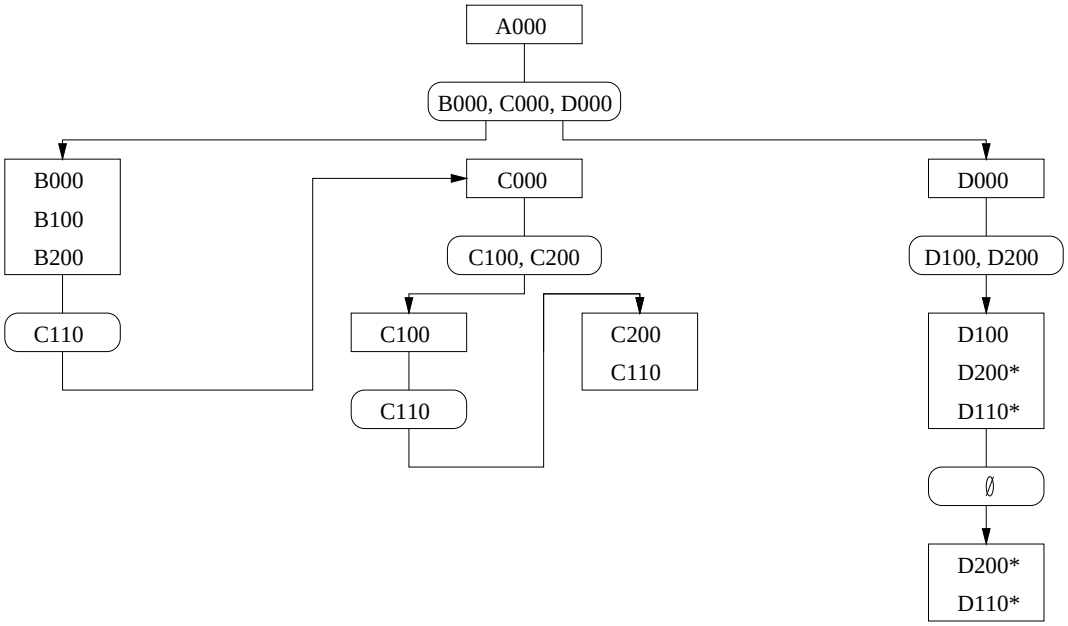

Figure 15: The moral tree resulting from the call graph of Figure 1 with an additional call between $B000$ and $C110$.

Observe the differences between Figure 14 and Figure 15. In Figure 14, we highlighted relations (43) - (46). The equivalent relations may be read from Figure 15.

1. $\{A000\} \perp\!\!\!\perp \{B100, B200, C100, C200, C110, D100, D200, D110\} | \{B000, C000, D000\}$ (47)
2. $\{B000, B100, B200\} \perp\!\!\!\perp \{C000, C100, C200\} | C110;$ (48)
3. $\{B000, B100, B200\} \perp\!\!\!\perp \{D000, D100, D200, D110\} | \emptyset;$ (49)
4. $\{C000, C100, C200, C110\} \perp\!\!\!\perp \{D000, D100, D200, D110\} | \emptyset.$ (50)

The only change to these four conditional independencies is to equation (48) which explicitly shows that the sets $\{B000, B100, B200\}$ and $\{C000, C100, C200\}$ cease to be independent as a resultant of the added call, but that they are conditionally independent given $C110$. Thus, the moral tree explicitly shows that the sets $\{B000, B100, B200\}$ and $\{C000, C100, C200\}$ are related to each other, but remain independent of $\{D000, D100, D200, D110\}$. There is no need to refer back to the call graph to deduce this, but this relationship cannot be found in Figure 3. We should emphasise a point about the algorithm here. Having split upon $A000$, $B000$, $C000$, $D000$ are all available for splitting upon and we have free choice. We choose $B000$ and after splitting, the only available root nodes are $C000$ and $D000$, so that we perform a backwards step in the algorithm. However, $A_{B000} \cap B_{B000} = C110$, so that we know that the three sets $de(B000)$, $de(C000)$, $de(D000)$ on $\mathcal{G}$ are not all unconnected. For the choice between $C000$ and $D000$ for splitting upon, we have to choose one that has $C110$ in its descendents. Thus, we choose $C000$. This enables us to identify $D000$ as being the root node of a separate subgraph since $A_{C200} \cap B_{C110} = \emptyset$, but $B_{C110} = \{D000, D100, D200, D110\}$.

Suppose that, as an alternative, we consider adding a call between $A000$ and $C110$ of

Figure 1. This results in a new moral tree as shown in Figure 16. As we explained when discussing Figure 3, the dominance tree corresponding to this is also given by Figure 3.

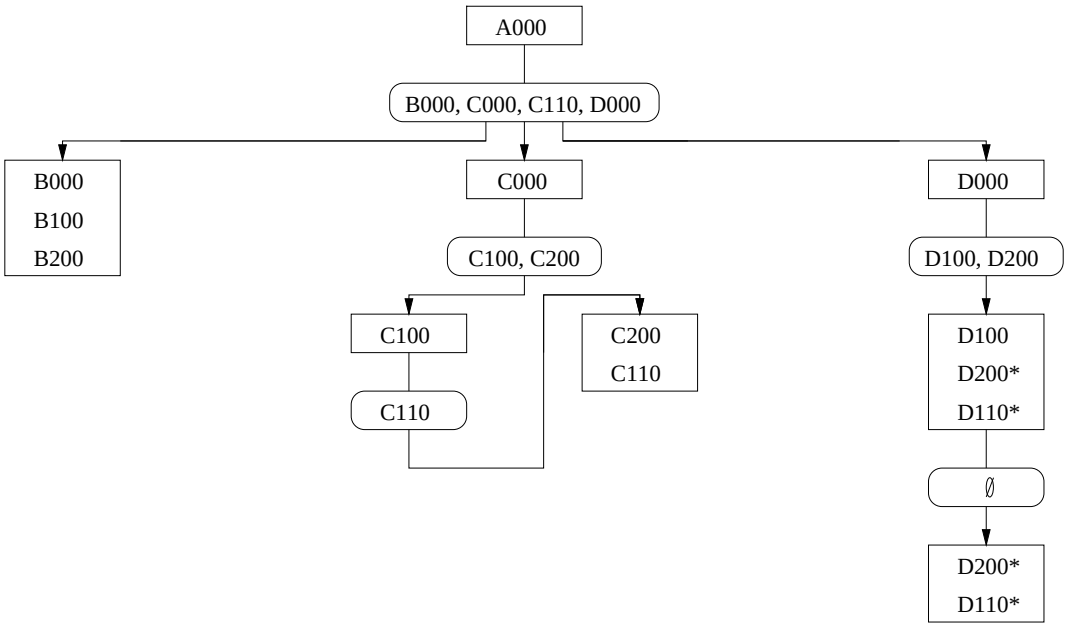

Figure 16: The moral tree resulting from the call graph of Figure 1 with an additional call between $A000$ and $C110$.

Observe the differences between Figure 14 and Figure 16. We state the equivalent relations on Figure 16 to those given by relations (43) - (46).

1. $\{A000\} \perp\!\!\!\perp \{B100, B200, C100, C200, D100, D200, D110\} | \{B000, C000, C110, D000\}$ (51)
2. $\{B000, B100, B200\} \perp\!\!\!\perp \{C000, C100, C200, C110\} | \emptyset;$ (52)
3. $\{B000, B100, B200\} \perp\!\!\!\perp \{D000, D100, D200, D110\} | \emptyset;$ (53)
4. $\{C000, C100, C200, C110\} \perp\!\!\!\perp \{D000, D100, D200, D110\} | \emptyset.$ (54)

Notice that the only change is in relation (51), which results from $C110$ also being a child of $A000$ and thus marrying $B000$, $C000$ and $D000$ on the corresponding moral graph. However, relations (52) - (54) remain unchanged. In particular observe that $\{B000, B100, B200\}$ is independent of $\{C000, C100, C200\}$. The moral trees shown in Figure 15 and Figure 16 are different to reflect the different independencies within the call structure; the corresponding dominance trees do not pick up upon this difference. The addition of a call between $B000$ and $C110$ and the conditional independencies shown by relations (48) - (50) supports the argument of the three reuse candidates:

1. {B000, B100, B200}
2. {C000, C100, C200}
3. {D000, D100, D200, D110}

as argued by Burd & Munro (1999), whilst additionally highlighting the role of the service candidate $C110$, which make ripple effects easier to understand. The addition of a call

between $A000$ and $C110$ argues that whilst we may take the above three sets as reuse candidates, we may extend $\{C000, C100, C200\}$ to $\{C000, C100, C200, C110\}$.

Suppose that to the original call graph of Figure 1, we add a call between $C000$ and $D000$. The resultant dominance tree for this change was given by Figure 4; the moral tree is given in Figure 17.

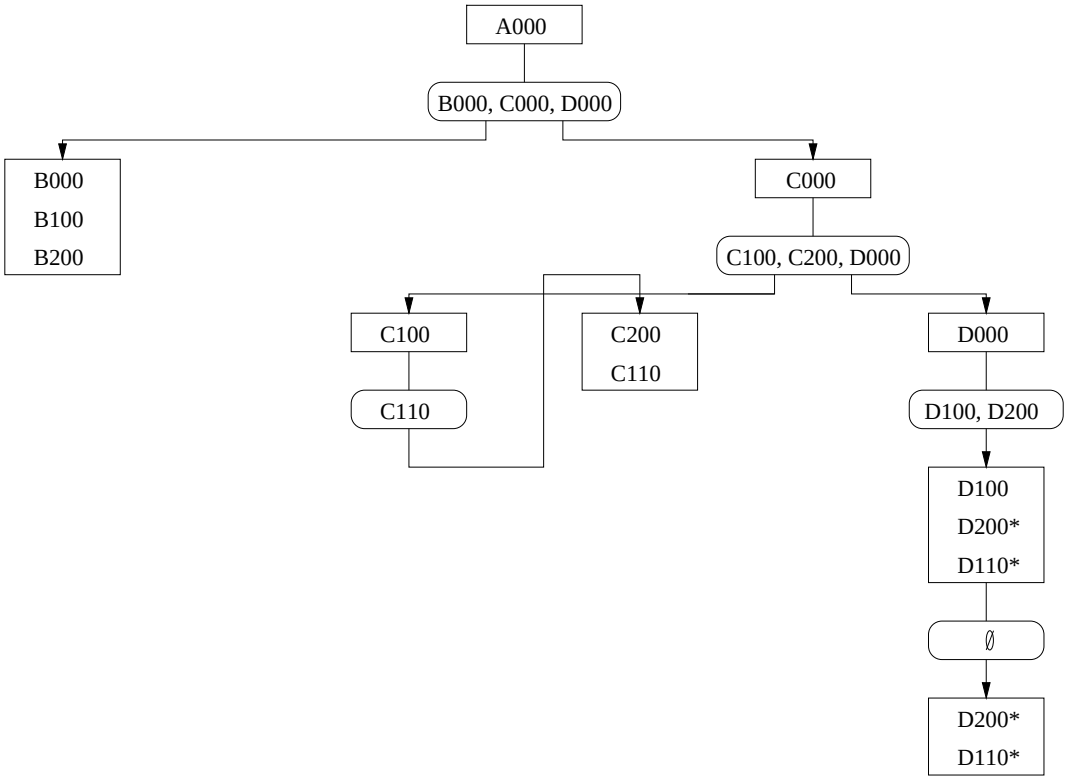

Figure 17: The moral tree resulting from the call graph of Figure 1 with an additional call between $C000$ and $D000$.

Observe how Figure 17 differs from Figure 16. These are perhaps best seen by examining the analogous (interesting) conditional independencies exhibited on Figure 17 to those on Figure 14. We now have:

1. $\{A000\} \perp\!\!\!\perp \{B100, B200, C100, C200, C110, D100, D200, D110\} | \{B000, C000, D000\}$ (55)
2. $\{B000, B100, B200\} \perp\!\!\!\perp \{C000, C100, C200, C110\} | \emptyset$; (56)
3. $\{B000, B100, B200\} \perp\!\!\!\perp \{D000, D100, D200, D110\} | \emptyset$; (57)
4. $\{C100, C200, C110\} \perp\!\!\!\perp \{D000, D100, D200, D110\} | \emptyset$. (58)

Thus, the consequence of the additional call is seen in relation (58). The call between $C000$ and $D000$ produces a dependence between the sets $\{C000, C100, C200, C110\}$ and $\{D000, D100, D200, D110\}$, but each of these sets remain independent from the set $\{B000, B100, B200\}$. As this dependence is caused by $C000$ calling $D000$, then relation (58) holds. This dependence is suggested in Figure 4 by the change of $D000$ from being strongly directly dominated to only being directly dominated, but from the dominance tree alone, it is impossible to determine what causes this dependence. The moral tree, however, shows this

clearly. Moreover, the relation (58) is only available on the moral tree and suggests that we may consider, in this case, using the following sets as reuse candidates:

1. $\{B000, B100, B200\}$
2. $\{C100, C200, C110\}$
3. $\{D110, D100, D200, D110\}$

This information is not supported on the dominance tree in this example. Suppose that there was also calls to $D000$ from $C100$ and $C200$. The dominance tree in this case does not change and remains the same as that of Figure 4, but in this case relation (58) no longer holds and this is picked up by the moral tree. The point was made in the discussion of Figure 4, that the addition of a call $C110$ to $D000$ also results in the dominance tree of Figure 4, but that we may consider using the sets

1. $\{B000, B100, B200\}$
2. $\{C000, C100, C200\}$
3. $\{D110, D100, D200, D110\}$

as reuse candidates. This intuition is supported by the moral tree in this case as Figure 18 shows.

Notice how Figure 17 differs from Figure 18, although the corresponding dominance trees are identical. It is clear on the moral tree that the alteration of direct dominance from strongly directly dominated for the node $D000$ is caused by the call $C110$ to $D000$, which appears as a node-separator pair and that the following relations hold:

$$\{C000, C100, C200\} \perp\!\!\!\perp \{D000, D100, D200, D110\} | C110; \tag{59}$$

$$\{C000, C100, C200, C110\} \perp\!\!\!\perp \{D100, D200, D110\} | D000. \tag{60}$$

The two examples given by Figures 17 and 18 show how it is able to detect deeper structure in the code than the dominance tree, helping to resolve potential difficulties of the moral tree.

References

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

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

Cowell, R. G., A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter (1999). *Probabilistic Networks and Expert Systems*. New York: Springer.

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

Dawid, A. P. (1980). Conditional independence for statistical operations. *Ann. Statist. 8*, 598–617.

Goldstein, M. (1981). Revising previsions: a geometric interpretation (with discussion). *J. R. Statist. Soc. B 43*, 105–130.

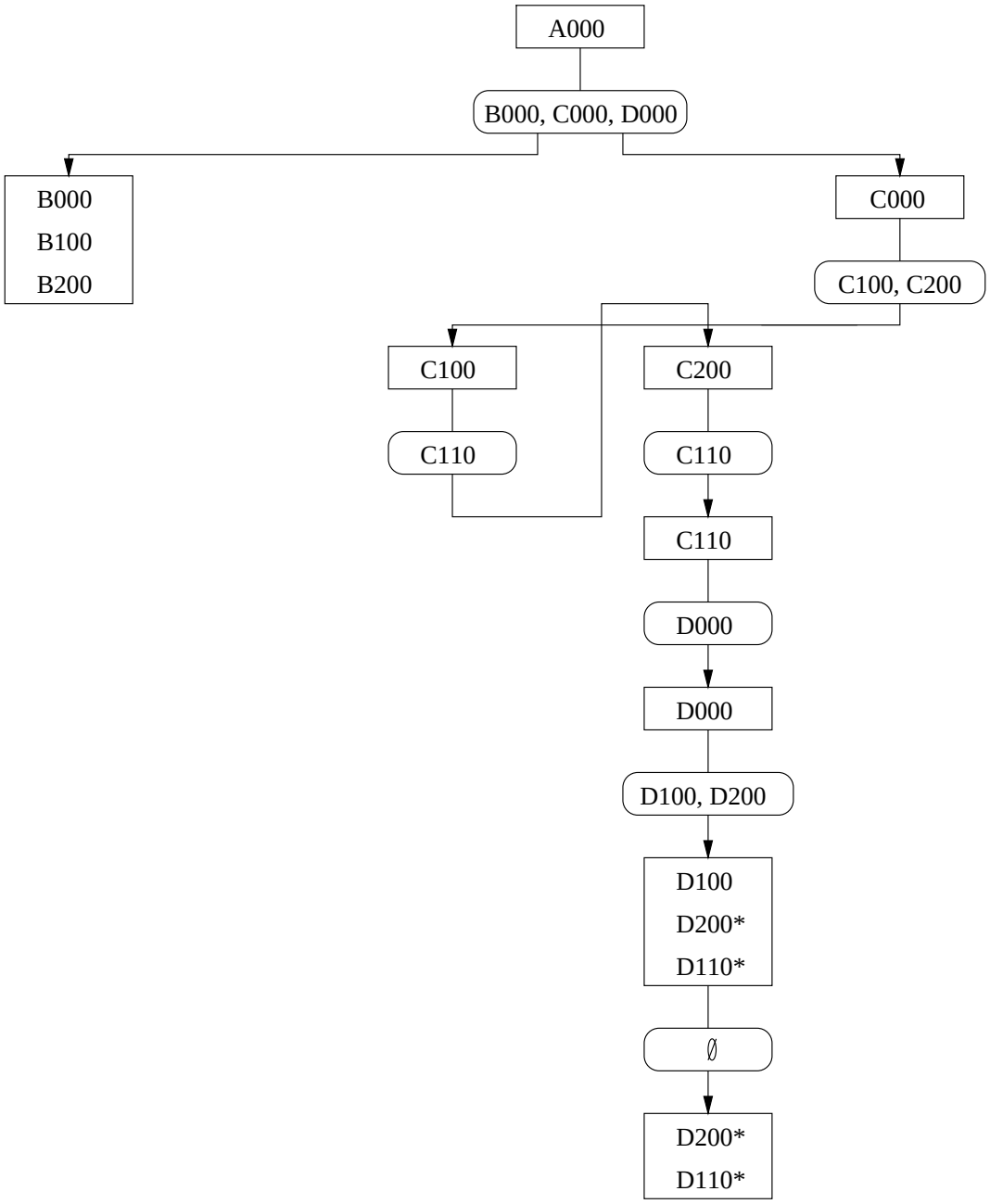

Figure 18: The moral tree resulting from the call graph of Figure 1 with an additional call between $C110$ and $D000$.

Goldstein, M. (1986). Exchangeable belief structures. *J. Amer. Statist. Ass. 81*, 971–976.

Goldstein, M. (1988). Adjusting belief structures. *J. R. Statist. Soc B 50*(1), 133–154.

Goldstein, M. (1990). Influence and Belief Adjustment. In R. M. Oliver and J. Q. Smith (Eds.), *Influence Diagrams, Belief Nets and Decision Analysis*, pp. 143–174. Wiley.

Goldstein, M. (1994). Revising exchangeable beliefs: subjectivist foundations for the inductive argument. In P. Freeman and A. Smith (Eds.), *Aspects of Uncertainty: A Tribute to D.V.Lindley*, pp. 201–222. Wiley.

Goldstein, M. (1999). Bayes linear analysis. In S. Kotz, C. B. Read, and D. L. Banks (Eds.), *Encyclopedia of Statistical Sciences Update Volume 3*, pp. 29–34. Chichester: Wiley.

Goldstein, M. and D. J. Wilkinson (2000). Bayes linear analysis for graphical models: The geometric approach to local computation and interpretive graphics. *Statistics and Computing 10*, 311–324.

Hetch, M. S. (1977). *Flow Analysis of Computer Programs*. Amsterdam: North-Holland.

Jensen, F. V. (1996). *An introduction to Bayesian networks*. London: University College London Press.

Lauritzen, S. L. (1996). *Graphical Models*. Oxford: Oxford Science Publications.

Lauritzen, S. L., A. P. Dawid, B. N. Larsen, and H.-G. Leimer (1990). Independence properties of directed Markov fields. *Networks 20*, 491–505.

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

Pearl, J. (1988). *Probabilistic Inference in Intelligent Systems*. San Mateo, California: Morgan Kaufman.

Smith, J. Q. (1989). Influence diagrams foe statistical modelling. *Annals of Statistics 17*, 654–672.

Smith, J. Q. (1990). Statistical Principles on Graphs. In R. M. Oliver and J. Q. Smith (Eds.), *Influence Diagrams, Belief Nets and Decision Analysis*, pp. 89–120. Wiley.

Whittaker, J. (1990). *Graphical Models in Applied Multivariate Statistics*. Chicester: Wiley.