Node Coarsening Calculi for Program Slicing

Mark Harman,
Rob Hierons
Brunel University,
Uxbridge, Middlesex,
UB8 3PH, UK.
Tel: +44 (0)1895 274 000
Fax: +44 (0)1895 251 686
Mark.Harman@brunel.ac.uk
Rob.Hierons@brunel.ac.uk

Sebastian Danicic,
John Howroyd,
Mike Laurence
Goldsmiths College,
University of London,
New Cross,
London SE14 6NW, UK.
Tel: +44 (0)20 7919 7856
Fax: +44 (0)20 7919 7853
Sebastian@mcs.gold.ac.uk

Chris Fox
Kings College,
University of London,
Strand,
London, WC2R 2LS, UK.
Tel: +44 (0)20 7848 2694
Fax: +44 (0)20 7848 2851
foxcj@dcs.kcl.ac.uk

Keywords: slicing, slice precision, node merging

Abstract

Several approaches to reverse and re-engineering are based upon program slicing. Unfortunately, for large systems, such as those which typically form the subject of reverse engineering activities, the space and time requirements of slicing can be a barrier to successful application.

Faced with this problem, several authors have found it helpful to merge Control Flow Graph (CFG) nodes, thereby improving the space and time requirements of standard slicing algorithms. The node-merging process essentially creates a 'coarser' version of the original CFG.

This paper introduces a theory for defining Control Flow Graph node coarsening calculi. The theory formalizes properties of interest, when coarsening is used as a precursor to program slicing. The theory is illustrated with a case study of a coarsening calculus, which is proved to have the desired properties of sharpness and consistency.

1 Introduction

Program slicing [26, 17] is a source code analysis technique which extracts parts of the source code associated with certain computations defined by a 'slicing criterion'.

Slicing has been used to assist several reverse and re-engineering activities. For example, Beck and Eichmann use slicing at both statement and module level to extract functionality from Ada programs [2]; Canfora at al. [6] used slicing as a part of the RE^2 reverse and re-engineering project; Cimitile et al. [7] showed how slicing (together with other techniques) can be used to identify reuse candidates.

Slicing has also been applied to problem areas, closely related to reverse engineering, such as program comprehension [8], software maintenance [3, 11] and testing [4, 13, 16]. Tip [24] and Binkley and Gallagher [5] provide detailed surveys of slicing.

Other dependence-based analyses have been applied to reverse engineering, for example chopping [18], tucking [20] and the RECAST method [10]. The theoretical framework presented in the present paper is defined for slicing, but could easily be extended to apply to these related dependence-based analysis techniques.

When constructing program slices, it is sometimes helpful to increase slice-construction speed at the expense of precision, by merging program nodes. This may be particularly important where slicing is applied to reverse engineering and re-engineering, where the subject program under analysis may be subject to several 'change phases' or where the systems to be analyzed are prohibitively large.

Recently, several authors have found that the space and time requirements of slicing large systems make the application of standard algorithms problematic [9, 22, 1]. This difficultly can be ameliorated by applying these standard techniques to an abstracted program. The abstraction is achieved by merging together nodes of the program's control flow graph (CFG), upon which the standard techniques for slice construction [26, 17] are based. This node merging introduces imprecision but reduces space and time complexity.

This paper provides a theory to underpin the process of node coarsening. The theory allows precise statements to be made concerning the correctness and efficacy of node coarsening. The theory allows for formal analysis of the trade off between precision and space and time complexity.

Two important coarsening calculus properties are formally defined:

- **Consistency**, which captures the safety concern that slices of coarsened programs are guaranteed to be correct;
- **Sharpness**, which captures the property that a set of coarsening inference rules is as precise as the abstraction will allow.

The theory is illustrated with a case study which introduces a coarsening calculus, R-coarsening, for a simple intraprocedural language. The theory also allows coarsening at the interprocedural level, where a procedure makes a natural choice for converting to a single node. The intraprocedural example of R-coarsening presented in Section 3 has been chosen because it illustrates the issues of consistency, sharpness and minimality in a comparatively straightforward manner.

For ease of exposition, only end slicing [19] will be considered, but the results presented can be generalized to cover other forms of slicing. Hereinafter, the terms 'slice' and 'end slice' will be used interchangeably. In end-slicing, the slicing criterion is simply a set of variables, V.

Definition 1.1 (End Slicing)
An end slice of a program p, constructed with respect to a slicing criterion V, is any program which can be constructed from p by deleting nodes of the Control Flow Graph (CFG) of p in such a way that the effect of p upon all variables in V is preserved.

An end slice s, of a program p, constructed with respect to a slicing criterion V, has two properties; one syntactic and the other semantic. The syntactic property is that s is constructed from p by removing nodes from the Control Flow Graph of p. The semantic property is that s has the same effect as p on all variables in V. A slice thus preserves a projection of the original program's syntax and semantics [14]. An example program and one of its end slices are given in the Figures 1a and 1b.

Slicing algorithms [26, 17] are traditionally insensitive to changes in a program's arithmetic and boolean expressions which have no effect upon sets of referenced variables. Therefore, it will be convenient to abstract away from the precise details of concrete program syntax, to an abstract syntax in which expressions are denoted by the sets of variables they reference.

In this paper, this is achieved using program schemas [27, 21]. A program *schema* has the same syntactic structure as a program up to expressions. In the schema, expressions are replaced by the application of some uninterpreted function or predicate symbol to a set of actual parameters. The actual parameters are the referenced variables of the expression which this abstraction replaces. Thus, in a schema, all that is 'known' about an expression is its set of referenced variables.

A single schema denotes an equivalence class of many programs, each of which is equivalent up to referenced variables of expressions. An *interpretation* of a schema is a program obtained from the schema by replacing each function and predicate symbol with a real (fully interpreted) function or predicate. For example, the schema corresponding to the program in Figure 1a is given in Figure 1c and its slice constructed for $\{\mathtt{x}\}$ (at the schema level) is given in Figure 1d.

In order to coalesce nodes together, a new syntactic construct, the *blob*, will be added to the language. Syntactically, a blob will be a form of statement, denoting a subgraph which has been collapsed onto a single node, the blob, which contains sufficient information to approximate the information denoted by the original subgraph.

The rest of this paper is organised as follows. Section 2 introduces a theory of coarsening, which is illustrated in section 3 with an instance of a coarsening calculus called R-coarsening. Section 4 contains a proof that R-coarsening is both consistent and sharp with respect to the definitions in Section 2. This illustrates the way in which the theory introduced in section 2 allows for formal and rigorous certification of approaches to coarsening. Section 5 describes the relationship between coarsening and other work on coalescing nodes of program graphs. Section 6 concludes.

2 A Theory of Coarsening

A coarsening calculus, $\mathcal{BC}$, is a quintuple, $(\Gamma, \mathcal{W}, \mathcal{B}, \mathcal{N}, \mathfrak{B})$. Each of the elements of a coarsening calculus are described in more detail below. First, a notation is introduced to describe the process of coarsening.

Definition 2.1 (Coarsens to)
Let $\mathcal{R}$ be an inference rule, axiom or set of rules and axioms of a coarsening calculus. $s \leadsto_{\mathcal{R}} b$ denotes the fact that the blob b can be inferred from the statement s using $\mathcal{R}$.

When the inference rule set is clear from the context, the reflexive, transitive closure of a relation $\leadsto_{\mathcal{R}}$, will be denoted by $\leadsto$.

It will be helpful to label each node with a unique variable, which will be called a 'node label'. The variable is a new variable introduced solely to signify the inclusion of the corresponding node in the slice. Since there is a one-to-one correspondence between the nodes of the CFG on the one hand and the union of the set of predicates and arithmetic expressions on the other, it will be possible to achieve this node labelling by including an additional referenced variable v_n in the expression or predicate associated with each node n. Provided that each v_n is unique and not

`begin`
`z:=y;`
`if z=3`
`then`
`begin`
`  c:=y;`
`  x:=25`
`end;`
`i:=i+1`
`end`
	`begin`
	`z:=y;`
	`if z=3`
	`then`
	`begin`
	
	`  x:=25`
	`end;`
	
	`end`
		`begin`
		`z:=`f_1`(y);`
		`if `b_1`(z)`
		`then`
		`begin`
		`  c:=`f_2`(y);`
		`  x:=`f_3`()`
		`end;`
		`i:=`f_4`(i)`
		`end`
			`begin`
			`z:=`f_1`(y);`
			`if `b_1`(z)`
			`then`
			`begin`
			
			`  x:=`f_3`()`
			`end;`
			
			`end`

1a: Original program	1b: Slice on x	1c: Schematic Version	1d: Schema slice on x

Figure 1: End Slicing on variable x

previously used, this variable will become needed in the slicing criterion precisely when the node it labels becomes needed.

In a coarsening calculus, $\mathcal{BC} = (\Gamma, \mathcal{W}, \mathcal{B}, \mathcal{N}, \mathfrak{B})$, Γ is the programming language to be coarsened and $\mathcal{W}$ is a slicing function which maps statements of Γ to functions on sets of variables and labels.

The slice of a program can easily be extracted from the slice function, $\mathcal{W}$, as defined by Definition 2.2 of Slice Computation below.

Definition 2.2 (Slice Computation)
Let $\mathcal{W}$ be a slicing function from programs to mappings on the set of labels and variables. Let L be the set of node labels. Given a slicing criterion c, the slice of a program p computed by $\mathcal{W}$ is $\mathcal{W}[\![p]\!]c \cap L$. The slice of p constructed with respect to the criterion c shall be denoted $\mathcal{S}[\![p]\!]c$.

In order to facilitate formal investigation of the properties of a coarsening calculus the slicing function is defined in a denotational style [23]. For a set of variables c, and statement s, $\mathcal{W}[\![s]\!]c$ returns a set of variables and node labels c', such that c' contains the node labels in the slice of s for the slicing criterion c. In addition c' also contains the set of variables upon whose initial value the slice depends.

For example

$$\mathcal{W}[\![\mathtt{z} := \mathtt{g}(\mathtt{y},\mathtt{z})]\!] = \lambda X. \begin{cases} \{\mathtt{g},\mathtt{y},\mathtt{z}\} \cup X & \text{if } \mathtt{z} \in X \\ X & \text{otherwise} \end{cases}$$

That is, when $\mathcal{W}$ is applied to `z := g(y,z)`, it produces a function which is capable of computing slices for the program fragment `z := g(y,z)`. Let F denote the function $\mathcal{W}[\![\mathtt{z} := \mathtt{g}(\mathtt{y},\mathtt{z})]\!]$. F can be used to compute end slices by applying it to a set of variables. In this case there are only two possibilities. That is, for any set B which contains z, $F(B) = \{g, y, z\} \cup B$ because the final value of the variables in B depend upon the line labelled by the node label g and the initial values of the variables y, z and the other (unaffected) variables in B. For any variable name other than z, F behaves like the identity function, because the final values of all other variables simply depend upon their initial value.

In a coarsening calculus, $\mathcal{BC} = (\Gamma, \mathcal{W}, \mathcal{B}, \mathcal{N}, \mathfrak{B})$, $\mathcal{B}$ is a set of inference rules, written $\frac{a}{b \rightsquigarrow c}$, denoting inferences of the form "if a holds, then the statement b of program p can be coarsened to give the statement c".

A program p might thus be transformed to p', using the inference rule to replace b in p by c in p'. By representing approaches to coarsening as inference systems, a separation is created between two concerns, namely: what constitutes a valid coarsening and how much coarsening will be required to achieve an acceptable improvement in speed of slice construction. This separation of concerns is necessary because the former concern has a more general logical flavor, while the later concern will be dependent upon the particular slicing application and system under consideration.

In a coarsening calculus, $\mathcal{BC} = (\Gamma, \mathcal{W}, \mathcal{B}, \mathcal{N}, \mathfrak{B})$, the syntactic class $\mathfrak{B}$ is the class of blob statements added to the language to facilitate coarsening.

Finally, in a coarsening calculus, $\mathcal{BC} = (\Gamma, \mathcal{W}, \mathcal{B}, \mathcal{N}, \mathfrak{B})$, $\mathcal{N}$ is a node function, which takes a statement and returns the set of node labels contained in the statement. For blobs, this will be the set of statements which have been coalesced to form the blob. For un-coarsened primitive statements, $\mathcal{N}$ will return the node label of the statement.

It is a requirement of a coarsening calculus that

$$\forall s \in \Gamma, b \in \mathfrak{B}. s \rightsquigarrow b \Rightarrow \mathcal{N}[\![s]\!] = \mathcal{N}[\![b]\!]$$

This requirement means that the exploitation of redundancy is *prohibited* from the coarsening process. It is required because coarsening is about coalescing nodes, and not about removing redundancy. Redundant code removal is a separate issue which should be addressed in a different (preferably earlier) analysis phase. Without this separation of concerns, the theory becomes far more intricate than necessary.

It will be helpful to define a semantic ordering, $\sqsubseteq$ on slicing functions. $f_1 \sqsubseteq f_2$ if slices produced by f_1 are always subsets of slices produced by f_2.

Definition 2.3 (Semantic Ordering)
Let f_1 and f_2 be two functions on sets of variables and node labels. f_1 *precedes* f_2, written $f_1 \sqsubseteq f_2$, iff $\forall c. f_1(c) \subseteq f_2(c)$.

It will also be useful to speak of strict semantic ordering, $\sqsubset$, which is defined in the obvious way:

$$f_1 \sqsubset f_2 \text{ iff } f_1 \sqsubseteq f_2 \wedge \neg(f_2 \sqsubseteq f_1)$$

Definition 2.4 (Consistency)
Let $\mathcal{BC} = (\Gamma, \mathcal{W}, \mathcal{B}, \mathcal{N}, \mathfrak{B})$ be a coarsening calculus. $\mathcal{BC}$ is *consistent* iff $\forall p \in \Gamma, p' \in \mathfrak{B}. p \rightsquigarrow p' \Rightarrow \mathcal{W}[\![p]\!] \sqsubseteq \mathcal{W}[\![p']\!]$.

In a consistent coarsening calculus it is impossible for a slice of a coarsened program to be smaller than the slice of the original program. This is important, because coarsening clearly cannot cause slices to get smaller. Were it to do so, the coarsening rules may not be safe and could not be considered consistent with the definition of slicing captured by $\mathcal{W}$.

Consistency merely guarantees that all slices constructed from a coarsened version of a program will be valid. However, a coarsening calculus could satisfy this requirement by being excessively conservative. In the most conservative case, the algorithm would simply return the entire program as the slice, regardless of the slicing criterion. Such overly conservative approaches will render the calculus safe but useless.

To address this issue of the 'level of conservativeness' of approximation in coarsening, two other properties of a coarsening calculus are defined: minimality and sharpness.

Definition 2.5 (Minimality)
Let $\mathcal{P}S$ be the powerset of a set S. Let $\mathcal{BC} = (\Gamma, \mathcal{W}, \mathcal{B}, \mathcal{N}, \mathfrak{B})$ be a coarsening calculus. $\mathcal{BC}$ is *minimal* iff $\forall s \in \Gamma, b \in \mathfrak{B}. s \rightsquigarrow b \Rightarrow \forall c \in \mathcal{P}(V \cup L). \mathcal{W}[\![s]\!]c - \mathcal{N}[\![s]\!] = \mathcal{W}[\![b]\!]c - \mathcal{N}[\![s]\!]$.

If a coarsening calculus is minimal then the slices constructed from blobs will be equivalent to those which would have been constructed by computing the slice, s of the original program, p and then adding to s all nodes which are in the blobs that contain nodes from s.

Minimality is a highly desirable property of a coarsening calculus, because it ensures that the *only* imprecision introduced by coarsening is embodied by the blobs themselves, and that the act of combining several nodes has no impact upon the data and control flow passing through the coarsened region of the CFG.

Using a non-minimal coarsening calculus the act of coarsening several nodes into a single blob may cause the data and control information of the coarsened region to be less precise than that of the original un-coarsened version. This may impact upon the slice computation of the entire program, as this imprecise information flows from the blob to the surrounding context. For such a non-minimal coarsening calculus, the effect of 'zooming in' to analyze a blob in more detail may result in a refinement of the slice of the surrounding code. By contrast a minimal calculus will provide the guarantee that zooming in on some chosen blob will only affect the slice of the blob, leaving the context in which the blob is located unchanged.

Desirable though minimality is, it remains a rather strong requirement upon a coarsening calculus, because it seems inevitable that, for many coarsening calculi, *some* of the available precision will be lost when nodes are combined to form a blob.

A related concept, *sharpness*, is therefore introduced. Intuitively, sharpness captures the property that a coarsening rule is 'as minimal as it can be' at the level of abstraction imposed by the syntax and semantics of the coarsening construct. If a coarsening calculus is not sharp then it is, in effect, *needlessly* imprecise, throwing away data and control information that could have been represented.

To illustrate sharpness, suppose two statements s_1 and s_2 appear in a sequence $s_1; s_2$, and that the sequence is to be merged to a single node n with defined and referenced variables which attempt to capture the information in the un-coarsened sequence. At first sight it may seem reasonable that the referenced variables of the blob should be the union of the referenced variables of its two constituents. This would yield a safe blob, but not a sharp one. For example, variables which are defined but not referenced in s_1 need not be referenced in the blob of $s_1; s_2$. To include them is safe, but it leads to *unnecessary* loss of precision.

Definition 2.6 (Sharpness)
Let $\mathcal{BC} = (\Gamma, \mathcal{W}, \mathcal{B}, \mathcal{N}, \mathfrak{B})$ be a coarsening calculus. $\mathcal{B}$ is *sharp* iff $\forall s \in \Gamma, b \in \mathfrak{B}. s \rightsquigarrow b \Rightarrow \neg\exists b' \in \mathfrak{B}. \mathcal{W}[\![s]\!] \sqsubseteq \mathcal{W}[\![b']\!] \sqsubset \mathcal{W}[\![b]\!]$.

Sharpness demands that no better blob can be found which would allow the coarsening calculus to be consistent. Thus the coarsening rule produces the best blob available within the constraints imposed by the structure of $\mathfrak{B}$ and the slicing function $\mathcal{W}$.

For a coarsening calculus to be useful it should be consistent and sharp. Minimality is also a desirable property. However, minimal coarsening calculi may require a large computational overhead to compute the blob and its associated data and control information. As the whole idea of coarsening is to compute blobs quickly, allowing precision to be traded for speed, it is unlikely that minimal coarsening calculi, though of theoretical interest, will be of practical use unless their blobs can be computed speedily.

3 Case Study: R-Coarsening

In this section a particular example of coarsening, called R-coarsening, is introduced in order to illustrate the application of the coarsening theory introduced in the previous section. Although introduced primarily for exposition purposes, the R-coarsening calculus is consistent and sharp and thus may form the basis for a practical and useful approach to coarsening.

R-coarsening will be defined for a simple intraprocedural language Γ, for which the syntax and slicing function, $\mathcal{W}$ are defined in Figure 2. The definition of the slicing function $\mathcal{W}$ for primitive statements (skip and assignment) is straightforward. For sequencing, the slicing functions of the two component statements are composed. For conditionals the slicing function is the union of the functions for the two branches composed with the function for the controlling predicate. For loops, the slicing function is the least solution to a recursion equation in f, which composes f with the function for the body and predicate of the loop. The slicing function, $\mathcal{W}$, uses two auxiliary functions $\mathbf{pred}$ and $\mathcal{C}$. The $\mathbf{pred}$ function captures the control dependence information imposed by predicates, while $\mathcal{C}$ computes the top level node labels of a statement (those which label the outermost expressions).

R-coarsening augments the language Γ with a single construct, the R-blobs, written $Rb(I, D, d, R)$. An R-blob is a quadruple, containing sufficient information to compute reasonably precise slices, while keeping the cost of computing blobs to a minimum.

In an R-blob, a distinction is introduced between the set of 'definitely defined' variables, denoted D, and the set of 'possibly defined' variables, denoted d. This distinction arises because several paths may be coalesced to form R-blob. Variables defined in all such paths are definitely defined by the R-blob. Those defined on at least one path (but not all paths) are 'possibly defined'.

Informally, the four components of an R-blob are as follows:

- I is the set of labels of the nodes which have been combined.
- D is the set of variables which are definitely assigned a value when control passes through the blob.
- d is the set of variables which may be assigned a value when control passes through the blob. (Clearly, in all R-blobs, $D \subseteq d$.)
- R is the set of referenced variables of the blob.

Syntactic Extensions

$$\Gamma ::= Rb(\mathcal{P}(L \cup V), \mathcal{P}(V), \mathcal{P}(V), \mathcal{P}(L \cup V))$$

Semantic Extensions

$$\mathcal{C}[\![Rb(I, D, d, R)]\!] = I$$

$$\mathcal{W}[\![Rb(I, D, d, R)]\!] =$$

$$\lambda in. \begin{cases} (in - D) \cup R \cup I & \text{if } in \cap (d \cup I) \neq \emptyset \\ in & \text{otherwise} \end{cases}$$

Figure 3: Augmentation of Figure 2 for R-coarsening

$$\begin{array}{rcl} \mathcal{N}[\![f : \mathit{skip}]\!] & = & \{f\} \\ \mathcal{N}[\![x := f(V)]\!] & = & \{f\} \\ \mathcal{N}[\![\mathit{begin}\ s_1; \cdots; s_n\ \mathit{end}]\!] & = & \bigcup_i \mathcal{N}[\![s_i]\!] \\ \mathcal{N}[\![\mathit{if}\ p(V)\ \mathit{then}\ s_1\ \mathit{else}\ s_2]\!] & = & \{p\} \cup \bigcup_i \mathcal{N}[\![s_i]\!] \\ \mathcal{N}[\![\mathit{while}\ p(V)\ \mathit{do}\ s]\!] & = & \{p\} \cup \mathcal{N}[\![s]\!] \\ \mathcal{N}[\![Rb(I, D, d, R)]\!] & = & I \end{array}$$

Figure 5: The node function, $\mathcal{N}$ for R-coarsening

3.1 Defining R-Coarsening using the Coarsening Theory

The theoretical framework from Section 2 is now used to define R-coarsening more formally.

In order to formally define a coarsening calculus according to the coarsening theory presented in section 2, each of the five elements $\Gamma, \mathcal{W}, \mathcal{B}, \mathcal{N}$ and $\mathfrak{B}$ must be defined. For R-coarsening, $\mathcal{W}$, is constructed by augmenting the rules for $\mathcal{W}$ given in Figure 2. The additional syntactic and semantic clauses are given in Figure 3. The inference rules and axioms of the R-coarsening calculus are given in Figure 4. The syntactic set $\mathfrak{B}$ consists of all syntactic constructs in the class $Rb(I, D, d, R)$ which can be reached from a program schema by the application of one or more of the inference rules, and the node labelling function $\mathcal{N}$ is defined in Figure 5.

In the the next section, the theory developed in section 2 is used to prove that R-coarsening is sound and sharp but that it is not minimal. This illustrates the way in which the calculus can be used to formalize properties of interest in a for graphs which contain coalesced nodes.

V : Variables $\qquad L$: Labels $\qquad \mathcal{W} : \Gamma \to \mathcal{P}(V \cup L) \to \mathcal{P}(V \cup L)$

$\mathbf{pred} : L \times \mathcal{P}(V) \times \mathcal{P}(L) \to (L \cup V) \to (L \cup V) \qquad \mathcal{C} : \Gamma \to \mathcal{P}(L)$

$\Gamma ::= L : \mathit{skip} \mid x := L(\mathcal{P}(V)) \mid \mathit{begin}\ \Gamma_1; \cdots; \Gamma_n\ \mathit{end} \mid \mathit{if}\ L(\mathcal{P}(V))\ \mathit{then}\ \Gamma_1\ \mathit{else}\ \Gamma_2 \mid \mathit{while}\ L(\mathcal{P}(V))\ \mathit{do}\ \Gamma$

$$
\begin{array}{rcl}
\mathcal{W}[\![f : \mathit{skip}]\!] & = & \lambda x.x \\
\mathcal{W}[\![s_1; s_2]\!] & = & \mathcal{W}[\![s_1]\!] \circ \mathcal{W}[\![s_2]\!] \\
\mathcal{W}[\![x := f(R)]\!] & = & \lambda in. \begin{cases} (in - \{x\}) \cup R \cup \{f\} & \text{if } x \in in \\ in & \text{otherwise} \end{cases} \\
\mathcal{W}[\![\mathit{if}\ p(R)\ \mathit{then}\ s_1\ \mathit{else}\ s_2]\!] & = & B \circ (\mathcal{W}[\![s_1]\!] \cup \mathcal{W}[\![s_2]\!]) \\
 & & \qquad \text{where } B = \mathbf{pred}(p, R, \mathcal{C}(s_1) \cup \mathcal{C}(s_2)) \\
\mathcal{W}[\![\mathit{while}\ p(R)\ \mathit{do}\ s]\!] & = & \mathit{fix} \lambda f. B \circ ((\mathcal{W}[\![s]\!] \circ f) \cup id) \\
 & & \qquad \text{where } B = \mathbf{pred}(p, R, \mathcal{C}(s)) \\
\mathbf{pred}(p, R, C) & = & \lambda in. \begin{cases} in \cup R \cup \{p\} & \text{if } in \cap C \neq \emptyset \\ in & \text{otherwise} \end{cases} \\
\mathcal{C}[\![f : \mathit{skip}]\!] & = & \{f\} \\
\mathcal{C}[\![x := f(R)]\!] & = & \{f\} \\
\mathcal{C}[\![s_1; s_2]\!] & = & \mathcal{C}[\![s_1]\!] \cup \mathcal{C}[\![s_2]\!] \\
\mathcal{C}[\![\mathit{if}\ p(R)\ \mathit{then}\ s_1\ \mathit{else}\ s_2]\!] & = & \{p\} \\
\mathcal{C}[\![\mathit{while}\ p(R)\ \mathit{do}\ s]\!] & = & \{p\}
\end{array}
$$

Figure 2: Syntax, Γ and Semantics, $\mathcal{W}$ of a Simple Illustrative Language

Axiom 3.1 (Skip) $f : \mathit{skip} \rightsquigarrow \mathit{Rb}(\{f\}, \emptyset, \emptyset, \emptyset)$

Axiom 3.2 (Assignment) $v := f(R) \rightsquigarrow \mathit{Rb}(\{f\}, \{v\}, \{v\}, R)$

Rule 3.1 (Sequence)

$$\frac{b_1 = \mathit{Rb}(I_1, D_1, d_1, R_1) \quad b_2 = \mathit{Rb}(I_2, D_2, d_2, R_2) \quad \{b_1, b_2\} \subseteq \mathfrak{B}}{b_1; b_2 \ \rightsquigarrow\ \mathit{Rb}(I_1 \cup I_2, D_1 \cup D_2, d_1 \cup d_2, R_1 \cup (R_2 - D_1))}$$

Rule 3.2 (Conditional)

$$\frac{b_1 = \mathit{Rb}(I_1, D_1, d_1, R_1) \quad b_2 = \mathit{Rb}(I_2, D_2, d_2, R_2) \quad \{b_1, b_2\} \subseteq \mathfrak{B}}{\mathit{if}\ p(R)\ \mathit{then}\ b_1 \mathit{else}\ b_2 \rightsquigarrow \mathit{Rb}(\{p\} \cup I_1 \cup I_2, D_1 \cap D_2, d_1 \cup d_2, R \cup R_1 \cup R_2)}$$

Rule 3.3 (Loop)

$$\frac{b = \mathit{Rb}(I, D, d, R) \quad b \in \mathfrak{B}}{\mathit{while}\ p(R')\ \mathit{do}\ b \rightsquigarrow \mathit{Rb}(\{p\} \cup I, \emptyset, d, R \cup R')}$$

Figure 4: A Calculus for R-Coarsening

4 Theoretical Properties of R-coarsening

This section uses the theory of coarsening to investigate and establish the theoretical properties of R-coarsening. It will be shown that R-coarsening is both consistent and sharp, but that it is not minimal.

Two lemmata will be useful in what follows.

Lemma 4.1 $\forall s \in \Gamma . \mathcal{W}[\![s]\!](A \cup B) = \mathcal{W}[\![s]\!]A \cup \mathcal{W}[\![s]\!]B$

Proof: By simple induction on the structure of Γ.

Lemma 4.2 (Inclusion)

$$\forall c, I, D, d, R . \mathcal{W}[\![Rb(I, D, d, R)]\!]c \subseteq (c - D) \cup R \cup I$$

Proof: Trivial.

4.1 R-Coarsening is Consistent

In this section the consistency of R-coarsening is established by proving that each inference rule is consistent.

The cases of *skip* and assignment are both trivial. There are three other cases, corresponding to the Γ constructs for sequences, conditionals and loops. Once each of the rules has been proved to be consistent then the combination of these rules must also, by transitivity, be consistent.

Case 1: Sequences
Let $r_i = Rb(I_i, D_i, d_i, R_i)$ (for i $\in \{1, 2\}$)
Let $r = Rb(I_1 \cup I_2, D_1 \cup D_2, d_1 \cup d_2, R_1 \cup (R_2 - D_1))$

Case 1.1 if $x \notin I_1 \cup I_2 \cup d_1 \cup d_2$, then
$\mathcal{W}[\![r_1]\!](\mathcal{W}[\![r_2]\!]\{x\}) = \{x\} = \mathcal{W}[\![r]\!]\{x\}$.

Case 2.2 if $x \in I_2 \cup d_2$ then
$\mathcal{W}[\![r_1]\!](\mathcal{W}[\![r_2]\!]\{x\}) - \mathcal{W}[\![r]\!]\{x\}$
$=\mathcal{W}[\![r_1]\!]((\{x\} - D_2) \cup R_2 \cup I_2) - \mathcal{W}[\![r]\!]\{x\}$
$=\mathcal{W}[\![r_1]\!](\{x\} - D_2) \cup \mathcal{W}[\![r_1]\!](R_2 \cup I_2) - \mathcal{W}[\![r]\!]\{x\}$
$\subseteq$ (*if* $x \in D_2$ *then* $\emptyset$ *else* $(\{x\} - D_1) \cup R_1 \cup I_1)$
$\cup R_1 \cup I_1 \cup ((R_2 \cup I_2) - D_1) - ((\{x\} - D_1 \cup D_2)$
$\cup R_1 \cup (R_2 - D_1) \cup I_1 \cup I_2)$ (since $I_2 \cap D_1 = \emptyset$)
$=\emptyset$.

Case 3.3 if $x \in (d_1 \cup I_1) - (d_2 \cup I_2)$ then
$\mathcal{W}[\![r_1]\!](\mathcal{W}[\![r_2]\!]\{x\})$
$=\mathcal{W}[\![r_1]\!]\{x\}$
$=(\{x\} - D_1) \cup R_1 \cup I_1$
$\subseteq (\{x\} - (D_1 \cup D_2)) \cup R_1 \cup (R_2 - D_1) \cup I_1 \cup I_2$
$=\mathcal{W}[\![r]\!]\{x\}$.

Thus $\mathcal{W}[\![r_1; r_2]\!] \sqsubseteq \mathcal{W}[\![r]\!]$.

Case 2: Conditionals
Let $r_i = Rb(I_i, D_i, d_i, R_i)$ (for i $\in \{1, 2\}$)
Let $q = Rb(I_1 \cup I_2 \cup \{p\}, D_1 \cap D_2, d_1 \cup d_2, R \cup R_1 \cup R_2)$.
Let $r =$ *if* $p(R)$ *then* r_1 *else* r_2.

Case 2.1 if $x \in d_1 \cup d_2 \cup I_1 \cup I_2$ then $\mathcal{W}[\![r]\!]\{x\} =$
$B(p, R, C(r_1) \cup C(r_2))(\mathcal{W}[\![r_1]\!]\{x\} \cup \mathcal{W}[\![r_1]\!]\{x\})$
$\subseteq \{p\} \cup R \cup \mathcal{W}[\![r_1]\!]\{x\} \cup \mathcal{W}[\![r_1]\!]\{x\}$
$\subseteq \{p\} \cup R \cup (\{x\} - D_1) \cup R_1 \cup I_1 \cup (\{x\} - D_2) \cup I_2 \cup R_2$
$\subseteq \{p\} \cup R \cup (\{x\} - (D_1 \cap D_2)) \cup \bigcup_{i=1,2} (R_i \cup I_i)$
$= \mathcal{W}[\![q]\!]\{x\}$.

Case 2.2 if $x \notin d_1 \cup d_2 \cup I_1 \cup I_2$ then $\mathcal{W}[\![r]\!]\{x\} =$
$B(p, R, C(r_1) \cup C(r_2))(\mathcal{W}[\![r_1]\!]\{x\} \cup \mathcal{W}[\![r_2]\!]\{x\})$
$= B(p, R, C(r_1) \cup C(r_2))\{x\}$
$= B(p, R, I_1 \cup I_2)\{x\} = \{x\}$
$\subseteq \mathcal{W}[\![q]\!]\{x\}$.

Case 3: Loops
Let $r = Rb(I, D, d, R)$.
Let $q = Rb(\{p\} \cup I, \emptyset, d, R \cup R')$.
Let $X_0 =$ *if* $p(R')$ *then* r *else* g : *skip*.
Let $X_{i+1} =$ *if* $p(R')$ *then* $r; X_i$ *else* g : *skip*.
Let $t = Rb(\{p\} \cup I, D, d, R \cup R')$.
Let $e = Rb(\{g\}, \emptyset, \emptyset, \emptyset)$.
Observe that $\forall i . \mathcal{W}[\![X_{i+1}]\!] \sqsupseteq \mathcal{W}[\![X_i]\!]$.
Also notice $\exists n \geq 0 . \mathcal{W}[\![$*while* $p(R')$ *do* $r]\!] = \mathcal{W}[\![X_n]\!]$.
Clearly $\mathcal{W}[\![X_0]\!] \sqsubseteq \mathcal{W}[\![q]\!]$.
Now assume $\mathcal{W}[\![X_i]\!] \sqsubseteq \mathcal{W}[\![q]\!]$;
Then $\mathcal{W}[\![X_{i+1}]\!] = \mathcal{W}[\![$*if* $p(R')$ *then* $r; X_i$ *else* g : *skip*$]\!]$
$\sqsubseteq \mathcal{W}[\![$*if* $p(R')$ *then* $Rb(I, D, d, R); q$ *else* g : *skip*$]\!]$
$\sqsubseteq \mathcal{W}[\![$*if* $p(R')$ *then* t *else* g : *skip*$]\!]$
$\sqsubseteq \mathcal{W}[\![$*if* $p(R')$ *then* t *else* $e]\!]$
$\sqsubseteq \mathcal{W}[\![q]\!]$ (Using if rule)
So $\mathcal{W}[\![$*while* $p(R')$ *do* $r]\!]$
$\sqsubseteq \mathcal{W}[\![q]\!]$.

4.2 R-Coarsening is Sharp

In this section it is proved that the R-coarsening calculus is sharp. The base cases of skip and assignment are trivial because R-coarsening rules for these two constructs are clearly minimal and are therefore sharp. The rest of the section considers the three constructs: sequencing, conditionals and loops. The following Lemma will be useful in the proofs that follow.

Lemma 4.1 (Sharpness Sufficiency Criterion)
Let $b_i = Rb(I_i, D_i, d_i, R_i)$ for $i \in \{1, 2\}$).

$$D_1 - R_1 \subseteq D_2 - R_2 \;\wedge\; d_1 \supseteq d_2 \;\wedge\; R_1 \supseteq R_2$$
$$\Rightarrow$$
$$\mathcal{W}[\![b_1]\!] \sqsupseteq \mathcal{W}[\![b_2]\!]$$

Proof: By straightforward case analysis.

Lemma 4.1 provides a sufficiency criterion for demonstrating that one blob is no worse than another and thus serves to provide a means of proving the sharpness of each rule in the R-coarsening calculus by contradiction. The proofs for the inductive cases, sequence, conditionals and loops are all proofs by contradiction. That is, for each of the three cases in which a rule allows the inference $s \rightsquigarrow b$, it will be assumed that there does exists some 'better blob', b' which has the property $\mathcal{W}[\![s]\!] \sqsubseteq \mathcal{W}[\![b']\!] \sqsubset \mathcal{W}[\![b]\!]$, leading to a contradiction.

Case 1: Sequences
Let $s_i = Rb(I_i, D_i, d_i, R_i)$ for $i \in \{1, 2\}$.
Let $s = s_1; s_2$.
Let $b' = Rb(I_1 \cup I_2, D_{b'}, d_{b'}, R_{b'})$.
Let $b = Rb(I_1 \cup I_2, D_1 \cap D_2, d_1 \cup d_2, R_1 \cup (R_2 - D_1))$.
Assume $\mathcal{W}[\![s]\!] \sqsubseteq \mathcal{W}[\![b']\!] \sqsubset \mathcal{W}[\![b]\!]$.
If $x \in d_2$ then $\mathcal{W}[\![s]\!]\{x\} \supseteq I_2$, so $d_{b'} \supseteq d_2$.
If $x \in d_1 - d_2$ then $\mathcal{W}[\![s]\!]\{x\} \supseteq I_1$, so $d_{b'} \supseteq d_1 - d_2$.
Therefore $d_{b'} \supseteq d_1 \cup d_2$.
If $x \in I_2$ then $\mathcal{W}[\![s]\!]\{x\} \supseteq R_2 - D_1$.
If $x \in I_1$ then $\mathcal{W}[\![s]\!]\{x\} \supseteq R_1$.
Therefore $R_{b'} \supseteq R_1 \cup (R_2 - D_1)$.
If $x \in D_{b'} - R_{b'}$ then $x \notin \mathcal{W}[\![b']\!]\{x\} \supseteq \mathcal{W}[\![s]\!]\{x\}$
so $x \in D_1 \cap D_2 - (R \cup R_1 \cup R_2)$.
Therefore $D_{b'} - R_{b'} \subseteq D_1 \cap D_2 - (R \cup R_1 \cup R_2)$.
Thus $\mathcal{W}[\![b]\!] \sqsubseteq \mathcal{W}[\![b']\!]$ by Lemma 4.1.

Case 2: Conditionals
Let $s_i = Rb(I_i, D_i, d_i, R_i)$ for $i \in \{1, 2\}$.
Let $s = \textit{if}\ p(R)\ \textit{then}\ b_1\ \textit{else}\ b_2$.
Let $\overline{I} = I_1 \cup I_2 \cup \{p\}$.
Let $\overline{R} = R \cup R_1 \cup R_2$.
Let $b' = Rb(\overline{I}, D', d', R')$.
Let $b = Rb(\overline{I}, D_1 \cap D_2, d_1 \cup d_2, \overline{R})$.
Assume $\mathcal{W}[\![s]\!] \sqsubseteq \mathcal{W}[\![b']\!] \sqsubset \mathcal{W}[\![b]\!]$.
$\mathcal{W}[\![s]\!] \sqsubseteq \mathcal{W}[\![b']\!] \ \wedge\ \mathcal{W}[\![s]\!]\overline{I} \subseteq \mathcal{W}[\![b']\!]\overline{I} \Rightarrow R' \supseteq \overline{R}$.
If $x \in d_1$ then $p \in \mathcal{W}[\![s]\!]\{x\} \subseteq \mathcal{W}[\![b']\!]\{x\}$,
so $x \in d'$, and so $d' \supseteq d_1$.
By a similar argument $d' \supseteq d_2$ so $d \supseteq d_1 \cup d_2$.
If $x \in D' - R'$ then $\mathcal{W}[\![s]\!]\{x\} \subseteq \mathcal{W}[\![b']\!]\{x\} = \overline{I} \cup R'$.
However, $x \notin R' \cup \overline{I}$,
so $x \in D_1 \cap D_2 - \overline{R}$, so $D' - R' \subseteq D_1 \cap D_2 - \overline{R}$.
Thus $\mathcal{W}[\![b]\!] \sqsubseteq \mathcal{W}[\![b']\!]$ by Lemma 4.1.

Case 3: Loops
Let $b_0 = Rb(I, D, d, R)$.
Let $X = \textit{if}\ p(R')\ \textit{then}\ b_0\ \textit{else skip}$.
Let $w = \textit{while}\ p(R')\ \textit{do}\ b_0$; hence $\mathcal{W}[\![w]\!] \sqsupseteq \mathcal{W}[\![X]\!]$.
Let $b = Rb(\{p\} \cup I, \emptyset, d, R \cup R')$.
Let $b' = Rb(\{p\} \cup I, \overline{D}, \overline{d}, \overline{R})$.
Assume $\mathcal{W}[\![b]\!] \sqsupset \mathcal{W}[\![b']\!] \sqsupseteq \mathcal{W}[\![X]\!]$.
If $x \in d$ then $\{p\} \cup I \cup R \cup R' \subseteq \mathcal{W}[\![X]\!]\{x\} \subseteq \mathcal{W}[\![b']\!]\{x\}$
$\Rightarrow x \in \overline{d}$, so $d \subseteq \overline{d}$.
Clearly $\mathcal{W}[\![b]\!]I = \mathcal{W}[\![X]\!]I \Rightarrow \overline{R} = R \cup R'$
(because I must be non-empty).
If $x \in \overline{D} - \overline{R}$ then $x \notin \mathcal{W}[\![b']\!]\{x\} \supseteq \mathcal{W}[\![X]\!]\{x\}$.
However $\forall x. x \in \mathcal{W}[\![X]\!]\{x\}$, so $\overline{D} - \overline{R} = \emptyset$.
Thus $\mathcal{W}[\![b]\!] \sqsubseteq \mathcal{W}[\![b']\!]$ by Lemma 4.1.

4.3 R-Coarsening is not Minimal

R-coarsening is not minimal. This arises because the inference rule 3.2 from Figure 4 is not minimal. Consider, for example, the program schema:

$$s = \textit{if}\ p(z)\ \textit{then}\ x := f_2(a)\ \textit{else}\ y := f_3(c)$$

for which it is possible, using rule 3.2 from Figure 4, to infer the R-blob:

$$b = Rb(\{p, f_2, f_3\}, \{\}, \{x, y\}, \{z, a, c\})$$

R-coarsening is not minimal because $\mathcal{W}[\![s]\!]x = \{p, z, f_2, a\}$, whereas $\mathcal{W}[\![b]\!]x = \{p, f_2, f_3, z, a, c\}$.

It is sufficient to demonstrate that a single rule is non-minimal in order to establish the non-minimality of the entire calculus. However, in this case, it turns out that the sequence rule is also non-minimal (consider $s = x := f_1(); y := f_2()$ for instance).

5 Related Work

The work presented here is similar to the graph reduction transformations T1 and T2, studied by Ullman, Hecht, Graham and Wegman [15, 25, 12], in which the graph transformations T1 and T2 are repeatedly applied to a (possibly unstructured) CFG to produce either a single node (in the case of well-behaved, but possibly unstructured CFGs) or an irreducible CFG (for highly unstructured programs).

The general form of the rules is depicted in Figure 6. The T1 transformation allows for a self targeting edge to be removed from a flow graph, while the T2 transformation allows one node (p in the figure) to 'consume' another (n in the figure). In the case of T2, the effect of this consumption upon the data and control flow information must also be recorded so that the coalesced node, (p, n), can be tagged with appropriate information.

The goal of CFG graph transformation was to find efficient ways of computing gen/kill information used in live-variable and reaching definition analysis. The transformations are applied as rules of a term rewriting system, in which terms are subgraphs. Ultimately, if the graph to be transformed is well-behaved[1] then the sequence of rewriting transformations leads to the construction of a single node which contains all the gen/kill information of the program.

[1] All structured programs and many unstructured ones have well-behaved graphs.

By contrast, the goal of the application to slicing considered in the present paper is not to reduce a program to a single node, rather it is to find some suitable compromise in which the graph retains sufficient detail to return useful slices, while affording a reasonable increase in speed of slice construction. In the gen/kill case, no information is lost in the 'coarsening process' using T1 and T2. However, for the slicing application *some* information *will* typically be lost.

Another difference lies in the convenience of schemas (rather than graphs). For example, consider the case of coalescing a structured conditional statement using the T1 and T2 transformations. Clearly, the only way that this can be achieved is for the predicate to consume first one branch and then the other, using two successive applications of the T2 rule. However, in applying T1 and T2 to slicing, the T2 transformation will introduce *unnecessary* imprecision, because the choice of definitely defined variables is forced to be imprecise (resulting in an empty set for D). The schema rule for R-coarsening the same conditional may have a non-empty set for D.

This leads to imprecise slices, for instance, the R-coarsening of

$$\mathit{if}\ p(z)\ \mathit{then}\ x := f_2(a)\ \mathit{else}\ x := f_3(c)$$

using T2 transformations will be

$$G_1 = \mathit{Rb}(\{p, f_2, f_3\}, \{\}, \{x\}, \{z, a, c\})$$

whereas it could have been

$$G_2 = \mathit{Rb}(\{p, f_2, f_3\}, \{x\}, \{x\}, \{z, a, c\})$$

The R-blob G_2 correctly includes x as a definitely defined variable, whereas G_1 does not. Failure to recognize x as definitely defined does not lead to inconsistency. However, the definition of the slice function $\mathcal{W}$ ensures that adding members to the 'definitely defined' set reduces slice size in some cases (and never increases it). More formally, in terms of the theory, $\mathcal{W}[\![G_2]\!] \sqsubset \mathcal{W}[\![G_1]\!]$, making the latter blob clearly preferable.

6 Conclusion

This paper has introduced a theory of coarsening. Coarsening is expressed as a calculus. This has two advantages:

- It allows for many different implementations of coarsening, each of which respects the rules of the calculus, but differs in the way in which it trades precision for speed.
- It allows for the definition of generic semantic properties of all coarsening calculi.

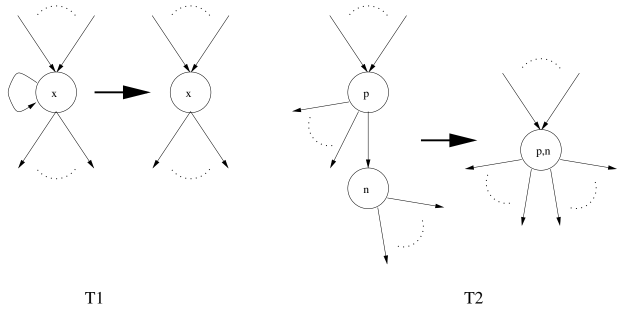

Figure 6: Graph Coalescing Transformations

The paper focussed on the second of these two advantages, formally defining the concepts of consistency, sharpness and minimality of a coarsening calculus. A consistent calculus produces correct slices. A sharp calculus is one which produces the most precise slices possible at the level of abstraction embodied in the coarsening construction. A minimal calculus is one which produces no additional imprecision over-and-above the grouping together of nodes produced by coarsening. Of these properties, a calculus must possess the first two to be of practical use.

The theory was illustrated by a case study which defined R-coarsening. R-coarsening is a sharp and consistent calculus, but it is not a minimal one.

References

[1] ATKINSON, D. C., AND GRISWOLD, W. G. Implementation techniques for efficient data-flow analysis of large programs. Tech. Rep. UCSD TR CS2001-0665, Feb. 2001.

[2] BECK, J., AND EICHMANN, D. Program and interface slicing for reverse engineering. In *IEEE/ACM 15^{th} Conference on Software Engineering (ICSE'93)* (1993), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 509–518.

[3] BENNETT, K., AND MORTIMER, R. Maintenance and abstraction of program data using formal transformations. In *IEEE International Conference on Software Maintenance* (1996), IEEE Computer Society Press, Los Alamitos, California, USA.

[4] BINKLEY, D. W. The application of program slicing to regression testing. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998, pp. 583–594.

[5] BINKLEY, D. W., AND GALLAGHER, K. B. Program slicing. In *Advances of Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1996, pp. 1–50.

[6] CANFORA, G., CIMITILE, A., AND MUNRO, M. RE^2: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance : Research and Practice 6*, 2 (1994), 53–72.

[7] CIMITILE, A., DE LUCIA, A., AND MUNRO, M. Qualifying reusable functions using symbolic execution. In *Proceedings of the 2^{nd} working conference on reverse engineering* (Toronto, Canada, 1995), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 178–187.

[8] DE LUCIA, A., FASOLINO, A. R., AND MUNRO, M. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension* (Berlin, Germany, Mar. 1996), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 9–18.

[9] DENG, Y., KOTHARI, S., AND NAMARA, Y. Program slice browser. In 9^{th} *IEEE International Workshop on Program Comprehesion (IWPC'01)* (Toronto, Canada, May 2001), IEEE Computer Society Press, Los Alamitos, California, USA. To appear.

[10] EDWARDS, H., AND M.MUNRO. RECAST : Reverse Engineering from COBOL to SSADM Specification. In *International Conference on Software Engineering* (1993), IEEE Computer Society Press, Los Alamitos, California, USA.

[11] GALLAGHER, K. B., AND LYLE, J. R. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (Aug. 1991), 751–761.

[12] GRAHAM, S. L., AND WEGMAN, M. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM 23*, 1 (Jan. 1976), 172–202.

[13] HARMAN, M., AND DANICIC, S. Using program slicing to simplify testing. *Software Testing, Verification and Reliability 5*, 3 (Sept. 1995), 143–162.

[14] HARMAN, M., SIMPSON, D., AND DANICIC, S. Slicing programs in the presence of errors. *Formal Aspects of Computing 8*, 4 (1996), 490–497.

[15] HECHT, M. S., AND ULLMAN, J. D. Flow graph reducibility. In *Conference Record, Fourth Annual ACM Symposium on Theory of Computing* (Denver, Colorado, 1–3 May 1972), pp. 238–250.

[16] HIERONS, R. M., HARMAN, M., AND DANICIC, S. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability 9*, 4 (1999), 233–262.

[17] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–61.

[18] JACKSON, D., AND ROLLINS, E. J. A new model of program dependences for reverse engineering. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering* (Dec. 1994), pp. 2–10.

[19] LAKHOTIA, A. Rule–based approach to computing module cohesion. In *Proceedings of the 15^{th} Conference on Software Engineering (ICSE-15)* (1993), pp. 34–44.

[20] LAKHOTIA, A., AND DEPREZ, J.-C. Restructuring programs by tucking statements into functions. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998, pp. 677–689.

[21] LUCKHAM, D. C., PARK, D. M. R., AND PATERSON, M. S. On formalised computer programs. *J. of Computer and System Sciences 4*, 3 (June 1970), 220–249.

[22] RILLING, J., A.SEFFAH, AND J.LUKAS. MOOSE – a software comprehension framework. In 5^{th} *World Multi-Conference on systemics, cybernetics and informatics (SCI 2001).* to appear.

[23] STOY, J. E. *Denotational semantics: The Scott–Strachey approach to programming language theory.* MIT Press, 1985. Third edition.

[24] TIP, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (Sept. 1995), 121–189.

[25] ULLMAN, J. D. Fast algorithms for the elimination of common subexpressions. *Acta Informatica 2*, 3 (1973), 191–213.

[26] WEISER, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

[27] WEYUKER, E. J. The applicability of program schema results to programs. *International Journal of Computer and Information Sciences 8*, 5 (Oct. 1979), 387–403.