A Formalisation of the Relationship between Forms of Program Slicing

Dave Binkley [a] Sebastian Danicic [b] Tibor Gyimóthy [c]
Mark Harman [d] Ákos Kiss [c] Bogdan Korel [e]

[a] *Loyola College in Maryland, Baltimore, Maryland, 21210-2699, USA*

[b] *Goldsmiths College, University of London, New Cross, London SE14 6NW, UK*

[c] *Institute of Informatics, University of Szeged, 6720 Szeged, Hungary*

[d] *King's College London, Strand, London WC2R 2LS, UK*

[e] *Illinois Institute of Technology, Chicago, IL 60616-3793, USA*

Abstract

The widespread interest in program slicing within the source code analysis and manipulation community has led to the introduction of a large number of different forms of slicing. Each preserves some aspect of a program's behaviour and simplifies the program to focus exclusively upon this behaviour. In order to understand the similarities and differences between forms of slicing, a formal mechanism is required. This paper further develops a formal framework for comparing forms of slicing using a theory of program projection. This framework is used to reveal the ordering relationship between various static, dynamic, simultaneous and conditioned forms of slicing.

Key words: program slicing, dynamic slicing, static slicing, conditioned slicing, simultaneous slicing, formal methods

1 Introduction

Program slicing is a technique for extracting parts of a program which affect a chosen set of variables of interest. By focusing on the computation of only

Email addresses: `binkley@cs.loyola.edu` (Dave Binkley), `s.danicic@gold.ac.uk` (Sebastian Danicic), `gyimi@inf.u-szeged.hu` (Tibor Gyimóthy), `mark@dcs.kcl.ac.uk` (Mark Harman), `akiss@inf.u-szeged.hu` (Ákos Kiss), `korel@iit.edu` (Bogdan Korel).

a few variables, slicing eliminates parts of the program which cannot affect these variables. The reduced program is called a *slice*.

Slicing has many applications because it allows a program to be simplified by focusing attention on a sub–computation of interest for a chosen application. It has been applied to reverse engineering [16,64], program comprehension [30,40], software maintenance [15,19,32,33], debugging [1,49,57,69], testing [10,35,37,44,45], component re–use [3,18], program integration [13,47], and software metrics [4,53,58]. There are several surveys of slicing, its applications and variations [11,12,29,39,65].

Slices can be static [48,68], or dynamic [2,51] or conditioned [14,22,31]. In static slicing, the input to the program is unknown and the slice must therefore preserve meaning for all possible initial states. By contrast, in dynamic slicing, the input to the program is known, and so the slice needs only preserve meaning for the input under consideration. Dynamic slicing is particularly useful in applications like debugging, where the input to the program has a crucial bearing on the problem in hand. Conditioned slicing lies in between static and dynamic. The slice must preserve the meaning of the original with respect to a set of initial states. The slicing algorithm uses knowledge about properties of the initial state in which the program will be executed in order to remove infeasible paths.

This paper is based on previous work by the authors [6–8] which is concerned with formal definitions and properties of slicing (rather than algorithms for computing them). The main contributions of [6–8] are to provide a mathematical framework which enables many different forms of slicing to be defined and compared. The concept that one form of slicing is *stronger than* another is made precise. The mathematical framework is based on the projection theory [36,38] of Harman, Danicic, and Binkley. *Unified equivalence* is introduced. Unified equivalence allows the semantics of different forms of slicing to be defined simply by different instantiations of its four parameters. The power of the approach is demonstrated by expressing eight forms of dynamic and static slicing, some traditional and some new, in terms of the unified equivalence and then proving that the eight forms give rise to a lattice with respect to strength.

In slicing, the smaller the slice the better. We formalise the definition of the set of minimal slices of a program with respect to a form of slicing and a slicing criterion [6]. An important relationship between relative size of minimal slices and relative strength of slicing is proved. Namely, the weaker the form of slicing, the smaller are its minimal slices.

The primary contributions of this paper can be summarised as follows:

(i) We further develop the previously introduced theoretical framework [6–

8]. The theory is not merely a mechanism for formally explaining what has gone before in program slicing; it also has a generative role. The theory makes possible to show that several new forms of slicing exist.

(ii) We use the formal framework to define both *simultaneous slicing* and *conditioned slicing*.

(iii) We formally define what it means for one form of slicing to be a generalisation of another and prove, using the theory, that both simultaneous slicing and conditioned slicing are generalisations of the standard forms of slicing considered in our previous work [6–8].

(iv) We use the theory to compare forms of simultaneous and conditioned slicing both with each other and with the other forms of slicing. We show that each form of simultaneous slicing is stronger than the corresponding non–simultaneous form and that each form of conditioned slicing is weaker than the corresponding static simultaneous form and stronger than the corresponding dynamic simultaneous form.

The rest of the paper is organized as follows:

Section 2 sets out a manifesto for a research programme in formalizing the theoretical foundations of program slicing. This paper then goes on to make a contribution to this manifesto.

In Section 3, the reader is introduced to the definitions and results of the theory [6–8] and the theory is further developed.

In Section 4 the theory is applied to simultaneous slicing [25]. In simultaneous slicing, the slicing criterion consists of many program points as opposed to a single program point in conventional, non–simultaneous slicing. Different forms of simultaneous slicing are formally defined (Definition 26) using unified equivalence. It is defined what it means for one form of slicing to be a generalisation of another (Definition 27). Using this definition, it is proved (Lemma 28) that each form of simultaneous slicing is a generalisation of the corresponding standard form given in [6–8]. It is proved (Lemma 29) that the different forms of simultaneous slice give rise to a lattice isomorphic to the lattice of standard forms given in [6–8] and that the lattice is strict (Lemma 30). It is proved that each form of simultaneous slicing is stonger than its corresponding standard form (Lemma 31). Finally (Lemma 32) it is proved that the lattice obtained from comparing minimal slices for simultaneous slicing is isomorphic to that of the standard case, the lattice is strict, and the minimal slices of each form of simultaneous slicing are larger than the corresponding standard forms given in [6–8].

In Section 5, the theory is applied to conditioned slicing [14,22,23,31]. The static and dynamic paradigms represent two extremes - either we say nothing about the input to the program (static slicing) or we say everything (dynamic slicing). Conditioned slicing allows us to bridge the gap. Different forms of conditioned slicing are formally defined (Definition 33) using unified equivalence. It is proved (Lemma 34) that each form of conditioned slicing is a generalisation of the corresponding standard form given in [6–8]. It is proved (Lemma 35) that the different forms of conditioned slice give rise to a lattice isomorphic to the lattice of simultaneous static forms given in Definition 26 and that the lattice is strict (Lemma 36). It is proved that each form of conditioned slicing is weaker than its corresponding static form (Lemma 37) and stronger than its corresponding dynamic form (Lemma 38).

Finally (Lemma 39) it is proved that the lattice obtained from comparing minimal slices for conditioned slicing is isomorphic to the lattice obtained by considering the different forms of static slicing. The lattice is strict, and the minimal slices of each form of conditioned slicing are larger than the corresponding simultaneous static forms and smaller than than the corresponding simultaneous dynamic forms. The relationship between all the forms of slicing considered in this paper is summarised in Figure 11.

Section 6 presents related work and, finally, in Section 7 the conclusions and directions for future work are given.

2 A Manifesto

The program projection theory provides a general framework, within which many current approaches to slicing could be formulated. This section sets out a seven point research agenda for a theoretical formalization of slicing with the aim of stimulating interest in the wider research community in tackling some of these problems.

(1) **Criteria**
This paper shows how dynamic, static, simultaneous and conditioned forms of slicing can be brought within the projection theory framework, allowing relationships between them to be explored. There are many other forms of slicing, which might benefit from a similar treatment. For example, quasi-static [66], forward [48] and amorphous [42] forms of slicing. There has been some work in this area, for example amorphous static slicing has been expressed within the projection framework [36], but more work is required to formulate all forms of slicing within the projection framework.

(2) **Minimality**
Much work needs to be done to characterise the forms of program that have minimal slices for each kind of slicing criterion. The theoretical results which would accrue from this research endeavor would have far–

reaching implications for the applications of slicing, since the size of the slice is important in all applications. Recent work [24] has shown that program schematology may be useful in the formulation and investigation of questions of slice minimality.

(3) **Complexity**
The algorithmic complexity of static, syntax preserving, slicing has been well-known for some time [48,62], but for other forms of slicing, the bounds on algorithmic complexity are less well understood.

(4) **Notation**
Although a comparatively trivial problem, the issue of a unified notation for expressing notions of slicing and their slicing criteria remains important. A common, widely used and unified notation would help to facilitate communication and would assist in expressing the relationships between forms of slicing.

(5) **Semantics**
Slicing does not preserve the traditional strict semantics of the programming language in which subject programs and their slices are expressed [17,27,34,43]. It is therefore necessary to define and capture the semantics preserved by slicing algorithms. This is a form of theory 'reverse engineering', since the algorithms are already in place, and the theory is dragging somewhat behind. Abstract Interpretation [20] would clearly be useful in this part of the research programme. An understanding of the semantics preserved by slicing is crucial to proving correctness. Intra-procedural static, syntax-preserving slicing has been proved correct [63]. This result was extended to inter-procedural static, syntax-preserving slicing by Binkley [5] and to a restricted set of programs with pointers by Horwitz et al. [46]. However, for other forms of slicing there are no correctness results.

Other important areas which would benefit from a formal semantic treatement include:

(a) definitions of data dependence in the presence of heap-allocated data that vary in precision following underlying points-to and alias analyses,
(b) definitions of projection and semantic correctness that take into account infinite traces and concurrency, and
(c) generalized notions of data/control dependence that into take account synchronization and event-based communication.

(6) **Relationships**
This paper shows how formalization of slicing criteria within the projection framework allows for slices to be compared using a lattice theoretic approach. The ability to explore the relationships between slicing criteria is one of the principal benefits of the theory we advocate. The results should help us to better understand slicing criteria.

(7) **Executability**
Some forms of slicing are executable. These are easier to fit into a theoreti-

cal framework, such as that proposed in the present paper, because there is an obvious equivalence between the slice and the original program. However, there are non-executable forms of program slice [9,48,66], and it remains an open challenge as to how these can be defined formally and compared to executable forms of slicing.

3 The Formal Framework

In this section, we introduce and further develop the formal framework [6–8] for defining and comparing forms of slicing. There are small differences between this exposition and previous ones [6–8] including the following:

(i) Here, we define the semantics preserved by slicing to be a mapping from slicing criteria to equivalence relations over programs rather than just an equivalence relation over programs. This technical change, percolates down thoughout the whole theory resulting in some simplifications to some of the definitions, results and proofs.
(ii) Now, in order to compare forms of slicing, we need a *slicing criterion comparison relation* that defines precisely which equivalence relations are comparable.
(iii) Finally, there is a change in nomenclature. Here, we do not talk about one form of slicing *subsuming* another, but rather one form of slicing being *weaker* than another. The reason for this change is that the notion of one definition being weaker than another is already well established. This captures exactly what is meant when comparing forms of slicing.

3.1 Program Projection Theory

Program Projection Theory [36,38] is a method for defining different forms of program slicing. Using projection theory, in order to define a form of slicing two relations on programs are required: a *pre-order*[1], $\lesssim$, over programs, and an *equivalence relation*, $\approx$, over programs. The relation, $\lesssim$, called the *syntacitc ordering* defines the syntactic property that the particular form of slicing seeks to optimize. Programs that are lower according to the ordering are considered to be *better*. The relation, $\approx$, called *semantic equivalence*, captures the semantic property that slicing intends to preserve.

Definition 1 (($\lesssim, \approx$)–projection) *Let $\lesssim$ be a pre-order over programs and let $\approx$ be an equivalence relation over programs. Program q is a $(\lesssim, \approx)$–projection*

[1] A pre-order is a transitive, reflexive relation.

of p with respect to ($\lesssim$,$\approx$) if and only if $q \lesssim p$ and $q \approx p$.

Projection theory, thus, elegantly separates the syntactic and semantic constraints inherent in different forms of program slicing.

3.2 The Syntactic Ordering Induced by Statement Deletion

For traditional static backward slicing and all other forms of slicing considered in this paper, slices are produced by deleting zero or more statements from the original program. We write $q \sqsubseteq p$ to mean q is obtained from p by deleting zero or more statements. This can be formalised as follows:

Definition 2 (The Traditional Syntactic Ordering: $\sqsubseteq$) *Let F be a function that takes a program and returns a partial function from line-numbers to statements, such that the function $F(p)$ maps l to c if and only if program p contains the statement c at line number l. Traditional syntactic ordering, denoted by $\sqsubseteq$, is defined as follows:*

$$p \sqsubseteq q \Leftrightarrow F(p) \ \subseteq \ F(q).$$

Lemma 3 *The traditional syntactic ordering, $\sqsubseteq$, has the property that every set of programs has a minimal element with respect to $\sqsubseteq$.*

In general, programs are sliced with respect to a slicing criterion. In static program slicing, for example, the slicing criterion is a set of a variables of interest V and a program line number n. Line numbers are used purely to identify statements and are not strictly part of the programming language. For this reason, in order to define a form of slicing we need to define two entities: a 'syntactic' pre-order and a 'semantic' function from slicing criteria to equivalences over programs. Formally,

Definition 4 (($\lesssim, E$)–slices) *Let $\lesssim$ be a syntactic ordering and E a function from slicing criteria to equivalence relations on programs, and c a slicing criterion. We say q is a $(\lesssim, E)$–slice of p with respect to slicing criterion c if and only if $q \lesssim p$ and q $E(c)$ p. If q is a $(\lesssim, E)$–slice of p with respect to slicing criterion c we write $q(\lesssim, E(c))p$.*

In other words, q is an $(\lesssim, E)$–Slice of p with respect to slicing criterion c if and only if q is a $(\lesssim, E(c))$ projection of p.

Static backward slicing was introduced by Mark Weiser [67]. Program q is a static backward slice of p with respect to (V, n) if p and q *agree semantically* (this is defined formally, later) with respect to variables V at line n. There is, of course, also the syntactic requirement that q must be obtained from p by statement deletion, *i.e.* $q \sqsubseteq p$. The semantic equivalence that static slicing respects is based upon the *state trajectory* [68].

Definition 5 (A State Trajectory) *A* state trajectory *is a finite sequence of (line-number, state) pairs:*

$$(n_1, \sigma_1)(n_2, \sigma_2) \ldots (n_k, \sigma_k)$$

where a state is a partial function mapping variables to values. Entry i is (n_i, σ_i) means that after i statement executions the program will be in state, σ_i, and the next statement to be executed is at line number n_i.

Notation: We write T_p^σ for the state trajectory arising from executing program, p, in initial state, σ.

For q to be a static slice of p with respect to slicing criterion (V, n), it must be *static backward equivalent* with respect to (V, n). Static backward equivalence is formally defined in Definition 8, but informally it means that for every initial state σ, the trajectories, T_p^σ and T_q^σ must semantically agree at (V, n). Again, informally, two trajectories agree semantically at (V, n) means that if we remove all elements apart from those which mention n and then, in what remains, just consider the subset of the states which are concerned with the set of variables V, then the two trajectories should appear to be identical. Formally, this is defined using two operations: *state restriction*, written '$\upharpoonright$', and a projection function called '*proj*'.

Definition 6 (State Restriction, $\upharpoonright$) *Given a state, σ and a set of variables V, $\sigma \upharpoonright V$ restricts σ so that it is defined only for variables in V:*

$$(\sigma \upharpoonright V)x = \begin{cases} \sigma\, x & \textit{if } x \in V, \textit{ and} \\ \perp & \textit{otherwise.} \end{cases}$$

For slicing criterion (V, n), and state trajectory $T = (l_1, \sigma_1)(l_2, \sigma_2) \ldots (l_k, \sigma_k)$, the function $Proj$ is defined as follows:

Definition 7 (*Proj*)

$$Proj_{(V,n)}(T) = Proj'_{(V,n)}(l_1, \sigma_1) \ldots Proj'_{(V,n)}(l_k, \sigma_k).$$

where

$$Proj'_{(V,n)}(l,\sigma) = \begin{cases} (l, \sigma \upharpoonright V) & \textit{if } l = n, \textit{ and} \\ \lambda & \textit{otherwise} \end{cases}$$

where λ denotes the empty string.

Having defined the necessary auxiliary functions, we are now in a position to define static backward equivalence, the semantic relationship preserved by backward static slicing as originally defined by Weiser [67].

Definition 8 (Static Backward Equivalence) *Given two programs p and q, and slicing criterion (V,n), p is* static backward equivalent *to q, written $p\,\mathcal{S}(V,n)\,q$, if and only if for all initial states σ, when the execution of p in σ gives rise to a state trajectory T_p^σ and the execution of q in σ gives rise to a state trajectory T_q^σ, then $Proj_{(V,n)}(T_p^\sigma) = Proj_{(V,n)}(T_q^\sigma)$.*

The static slicing semantic equivalence relation is parameterised by V and n, and hence it really defines a function from slicing criterion (V,n), to equivalence relations over programs. This reflects the fact that each slicing criterion yields slices that respect a different projection of the semantics of the program from which they are constructed.

In order to express Weiser's static backward slicing formally as a *form of slicing* as detailed in Definition 4, it is defined to be the pair, $(\sqsubseteq, \mathcal{S})$. This represents the fact that the syntactic ordering used by static backward slicing is traditional statement deletion, $\sqsubseteq$, and that the semantic equivalence function used by static backward slicing is the function, $\mathcal{S}$, from slicing criteria to equivalences over programs given in Definition 8.

EXAMPLE.
Consider programs p and q in Figure 1. Program p is static backward slice of q, because p is static backward equivalent to q, written $p\,\mathcal{S}(\{\texttt{z}\},9)\,q$ because for every initial state σ,

$$Proj_{(\{z\},9)}(T_p^\sigma) = Proj_{(\{z\},9)}(T_q^\sigma).$$

and p is obtained from q by statement deletion.

3.4 Comparing forms of slicing

Central to this work is the problem of defining whether one form of slicing is comparable with another. Is form of slicing A *weaker* than form of slicing B? Informally, form of slicing A is weaker than form of slicing B if and only if, for all programs p and q, whenever q is a B–slice of p with respect to

1	`n=input();`	1	`n=input();`
2	`s=0;`		
3	`z=1;`	3	`z=1;`
4	`while (n>1)`	4	`while (n>1)`
5	`{`	5	`{`
6	`s=s+n;`		
7	`z=z*n;`	7	`z=z*n;`
8	`n=n-1;`	8	`n=n-1;`
9	`}`	9	`}`
p: Original Program		q: Slice for $(\{\texttt{z}\}, 9)$	

Fig. 1. A Program and one of its slices

slicing criterion c it is also an A–slice of p with respect to slicing criterion c. Problematic in such a definition is that different forms of slicing do not always have the same type of slicing criteria. For example, in static backward slicing, the slicing criterion is a pair consisting of a set of variables and a line number, whereas in dynamic slicing it is a triple consisting of a set of variables, a line number, and an initial state. Before two such forms of slicing can be compared, it is necessary, therefore, to define which slicing criteria of each form of slice can be compared. Having done that, we define form of slicing, A, to be weaker than form of slicing, B, if and only if whenever slicing criteria c_A and c_B can be compared, q is a B–slice of p with respect to slicing criterion c_B it is also an A–slice of p with respect to slicing criterion c_A. In order to compare any two forms of slicing, a *slicing criterion comparison relation* for these two forms of slicing must first be defined.

Let $(\lesssim_A, E_A)$ and $(\lesssim_B, E_B)$ be forms of slicing, and let $\rightarrow$ be a slicing criterion comparison relation. We now formally define what it means for $(\lesssim_A, E_A)$ to be weaker than $(\lesssim_B, E_B)$, with respect to $\rightarrow$.

Definition 9 (Weaker than)
Slicing form $(\lesssim_A, E_A)$ is weaker than slicing form $(\lesssim_B, E_B)$ with respect to slicing criterion comparison relation, $\rightarrow$, if and only if

$\forall p, q \in P, c_A \rightarrow c_B$

$$q \ (\lesssim_B, E_B(c_B)) \ p \implies q \ (\lesssim_A, E_A(c_A)) \ p.$$

We write $(\lesssim_A, E_A) \overset{\rightarrow}{\subseteq} (\lesssim_B, E_B)$ to mean slicing form $(\lesssim_A, E_A)$ is weaker than slicing form $(\lesssim_B, E_B)$ with respect to slicing criterion comparison relation $\rightarrow$.

Dual to the definition of *weaker than* is *stronger than*:

Definition 10 (Stronger than)
Slicing form $(\lesssim_A, E_A)$ is stronger than slicing form $(\lesssim_B, E_B)$ with respect to slicing criterion comparison relation, $\rightarrow$ if and only if $(\lesssim_B, E_B)$ is weaker than

$(\lesssim_A, E_A)$ *with respect to* $\rightarrow$.

3.4.1 Example

As an example, we will now define a form of dynamic slicing, $(\sqsubseteq, \mathcal{D})$, and prove that it is weaker than static backward slicing, $(\sqsubseteq, \mathcal{S})$.

In the form of slicing represented by $(\sqsubseteq, \mathcal{D})$, the slicing criterion is of the form (σ, V, n) where σ is a single program state. The only difference between static and this form of dynamic slicing is that for dynamic slicing, only the trajectories corresponding to the single state, σ, need agree whereas for static slicing the trajectories corresponding to all states must agree.

Definition 11 (Dynamic Backward Equivalence, $\mathcal{D}$)
Given two programs p *and* q*, and slicing criterion* (σ, V, n)*,* p *is dynamic backward equivalent to* q*, written* $p\ \mathcal{D}(\sigma, V, n)\ q$*, if and only if when the execution of* p *in* σ *gives rise to a state trajectory* T_p^σ *and the execution of* q *in* σ *gives rise to a state trajectory* T_q^σ*, then* $Proj_{(V,n)}(T_p^\sigma) = Proj_{(V,n)}(T_q^\sigma)$.

In order to prove that $(\sqsubseteq, \mathcal{D})$, is weaker than $(\sqsubseteq, \mathcal{S})$, first we need to define the slicing criterion comparison relation, $\stackrel{1}{\rightarrow}$, that relates the different forms of slicing criteria between $\mathcal{D}$ and $\mathcal{S}$ as follows:

Definition 12 ($\stackrel{1}{\rightarrow}$)

$$(\sigma, V, n) \stackrel{1}{\rightarrow} (V, n) \quad \textit{for all } V, n, \sigma.$$

Now we state and prove the result that the form of dynamic slicing denoted by $(\sqsubseteq, \mathcal{D})$ is weaker than static backward slicing, $(\sqsubseteq, \mathcal{S})$.

Lemma 13 $(\sqsubseteq, \mathcal{D}) \stackrel{\stackrel{1}{\rightarrow}}{\subseteq} (\sqsubseteq, \mathcal{S})$.

PROOF. All that is required to prove is that for all σ, p, q, V, n,

$$p\ \mathcal{S}(V, n)\ q \implies p\ \mathcal{D}\ (\sigma, V, n)\ q.$$

This follows immediately from the definition of static backward equivalence, $\mathcal{S}$ (Definition 8), and the definition of $\mathcal{D}$ (Definition 11).

3.5 Minimal Slices

Statements like:

"Dynamic slices are smaller than static slices."

are occasionally heard amongst slicing researchers. We know intuitively what is meant by such statements but clearly, not all dynamic slices *are* smaller than all static slices. Even for a given choice of program point and variable, the statement may not be true, because of differences in slicing algorithms. Furthermore, there is the complication of which particular dynamic slicing definition one is to adopt; some are incomparable with static slicing. One interpretation of what is meant by such statements is that the *minimal* slices inherent in dynamic slicing are smaller than the *minimal* slices inherent in static slicing. One achievement of [6] is to make such statements precise. The main result of [6] is to formalise the fact that the weaker a form of slicing, the smaller are its minimal slices.

Informally, for any form of slicing $(\lesssim, E)$ and for any program p we define a *minimal slice* of p to be a slice of p with respect to c which has no syntactically smaller slices with respect to c.

Definition 14 (Minimal Slices) *Let $(\lesssim, E)$ be a form of slicing, c a slicing criterion, and p a program. Then program q is a minimal slice of p with respect to form of slicing, $(\lesssim, E)$ and c if and only if q is a $(\lesssim, E)$–slice of p with respect to c and no q' that is strictly syntactically smaller than q (with respect to $\lesssim$) is a $(\lesssim, E)$–slice of p with respect to c. Formally, for all programs p, q, and q':*

$$\begin{aligned} q\ \mathbb{M}^c(\lesssim, E)\ p \iff\ & q\ (\lesssim, E(c))\ p \\ & \textit{and} \\ & q' \lesssim (\neq)\ q \implies \neg(q'\ E(c)\ p). \end{aligned}$$

We can now define the set of minimal slices of p with respect to slicing criterion c as follows:

Definition 15 (The Set of Minimal Slices of p with respect to c) *Let p be a program, $(\lesssim, E)$ a form of slicing and c a slicing criterion. The set, $\mathbb{M}^c_p(\lesssim, E)$ of minimal slices of p with respect to c is defined as:*

$$\mathbb{M}^c_p(\lesssim, E) = \{q | q\ \mathbb{M}^c(\lesssim, E)\ p\}.$$

Definition 16 (Extending $<$ to sets) *Given a pre-order $<$ over a set S, we can define a pre-order over subsets of S as follows*

$$T_1 < T_2 \iff \textit{for all } t_2 \in T_2 \textit{ there exists } t_1 \in T_1 \textit{ with } t_1 < t_2.$$

Definition 17 (Smaller) *Let $(\lesssim, E_A)$ and $(\lesssim, E_B)$ be forms of slicing, and let $\rightarrow$ be a slicing criterion comparison relation between the slicing criteria of E_A and E_B.*

If $\mathbb{M}_p^{c_A}(\lesssim, E_A) \lesssim \mathbb{M}_p^{c_B}(\lesssim, E_B)$ *for all* $c_A \to c_B$ *for all programs,* p *we say the minimal slices of* $(\lesssim, E_A)$ *are smaller than the minimal slices of* $(\lesssim, E_B)$ *with respect to* $\to$ *and write*

$$(\lesssim, E_A) \stackrel{\rightarrow}{\lessapprox} (\lesssim, E_B).$$

Dual to the definition of *Smaller* is the definition of *Larger*:

Definition 18 (Larger) *Let* $(\lesssim, E_A)$ *and* $(\lesssim, E_B)$ *be forms of slicing, and let* $\to$ *be a slicing criterion comparison relation between the slicing criteria of* E_A *and* E_B*. If*

$$(\lesssim, E_A) \stackrel{\rightarrow}{\lessapprox} (\lesssim, E_B).$$

then we say the minimal slices of $(\lesssim, E_B)$ *are larger than the minimal slices of* $(\lesssim, E_A)$*.*

Lemma 19 (Duality Lemma) *Let* $(\lesssim, E_A)$ *and* $(\lesssim, E_B)$ *be forms of slicing where* $\lesssim$ *is such that every set of programs has a minimal element with respect to* $\lesssim$*, and let* $\to$ *be a slicing criterion comparison relation between the slicing criteria of* E_A *and* E_B*. Then*

$$(\lesssim, E_A) \stackrel{\rightarrow}{\subseteq} (\lesssim, E_B) \implies (\lesssim, E_A) \stackrel{\rightarrow}{\lessapprox} (\lesssim, E_B).$$

In other words, if form of slicing A is weaker then form of slicing B with respect to $\to$ and A and B both use the same syntactic ordering, then the minimal slices of A are smaller than the minimal slices of B with respect to $\to$.

PROOF. Let $(\lesssim, E_A) \stackrel{\rightarrow}{\subseteq} (\lesssim, E_B)$ and let $q \in \mathbb{M}_p^{c_B}(\lesssim, E_B)$.
Then, by Definition 14, $q\ E_B(c_B)\ p$ and so, by Definition 9, $q\ E_A(c_A)\ p$ for all $c_A \to c_B$. But $q \lesssim q$ (since $\lesssim$ is reflexive), so either there is no $q_1 \lesssim q$ such that $q_1\ E_B(c_B)\ p$ in which case $q \in \mathbb{M}_p^{c_A}(\lesssim, E_A)$ (by Definition 14) as required, or by Lemma 3, there exists a q_1 with $q_1 \in \mathbb{M}_p^{c_A}(\lesssim, E_A)$ and $q_1 \lesssim q$. In either case there is an element of $\mathbb{M}_p^{c_A}(\lesssim, E_A)$ which is $\lesssim q$. Hence $(\lesssim, E_A) \stackrel{\rightarrow}{\lessapprox} (\lesssim, E_B)$ by Definition 17.

An example of the use of Lemma 19 is as follows: We proved that dynamic form, $(\sqsubseteq, \mathcal{D})$, is weaker than Weiser's backward static form, $(\sqsubseteq, \mathcal{S})$. From Lemma 19, it follows that the minimal slices of the dynamic form, $(\sqsubseteq, \mathcal{D})$, are smaller than the minimal slices of Weiser Static slicing (Relative to the slicing criterion comparison relation $\stackrel{1}{\rightarrow}$ given in Definition 12).

3.6 Unified Equivalence

Another major contribution of [6–8] is to define *unified equivalence*, a framework for defining the semantics of a large class of forms of slicing. In [6–8], eight different forms of slicing were formally defined using unified equivalence. These include:

(1) Standard slicing at a point in a program with respect to a set of variables. ($\mathcal{D}$ and $\mathcal{S}$)
(2) Korel and Laski style slicing [51], a stronger form where the program and the slice must follow identical paths.($\mathcal{D}_{KL}$ and $\mathcal{S}_{KL}$)
(3) Iteration count slicing, where a program and its slice need only agree at a particular iteration of a program point. ($\mathcal{D}_i$ and $\mathcal{S}_i$)
(4) Korel and Laski iteration count slicing, where a program and its slice both must follow identical paths and need only agree at a particular iteration of a program point.($\mathcal{D}_{KLi}$ and $\mathcal{S}_{KLi}$)

Unified equivalence is based on the observation that slicing criteria differ in four orthogonal aspects:

- the set of initial states, S, of interest. (Normally a singleton for dynamic slicing and the set of all states for static),
- the set of variables, V, of interest,
- whether one or all occurrences of an instruction in the trajectory are considered (P) (This allows us to distinguish between iteration count and non–iteration count forms of slicing.),
- which elements, I, of the trajectory apart from the slice points in P are important. (This allows us to distinguish between KL (path aware) and non-path aware forms of slicing.)

Unified equivalence is defined in terms of an auxiliary trajectory function, $Proj^*$, which in turn is defined in terms of an auxiliary function, $Proj'^*$. The function, $Proj^*$, is a generalisation of the projection function, $Proj$, originally defined by Weiser [67] and given here in Definition 7.

First, we define the auxiliary function, $Proj'^*$. It is defined in terms of 5 parameters: a set of variables V, a set of (line-number, natural number) pairs P, a set of line numbers I, a (line-number, natural number) pair (n, k), and a state σ:

Definition 20 ($Proj'^*$)

$$Proj'^*(V, P, I, (n, k), \sigma) = \begin{cases} (n, \sigma \restriction V) & \textit{if } (n, k) \in P, \\ (n, \bot) & \textit{if } (n, k) \notin P \textit{ and } n \in I, \\ \lambda & \textit{otherwise.} \end{cases}$$

Note that $Proj'^*(V, P, I, (n, k), \sigma)$ evaluates either to a single pair or an empty sequence of pairs depending on its parameters. It, in effect, keeps a projected version of each pair if either $(n, k) \in P$ or $n \in I$ and otherwise it 'throws away' the pair completely.

Definition 21 ($Proj^*$) *$Proj^*$ is a function which takes a set of variables V, a set of (line-number, natural number) pairs P, a set of line numbers I and a state trajectory $T = (n_1, \sigma_1) \dots (n_m, \sigma_m)$. It is given by:*

$$Proj^*_{(V,P,I)}(T) = Proj'^*(V, P, I, (n_1, k_1), \sigma_1) \dots Proj'^*(V, P, I, (n_m, k_m), \sigma_m)$$

where for each $i \in \{1 \dots m\}$, k_i is the number of occurrences of n_i in $(n_1, \sigma_1) \dots (n_i, \sigma_i)$.

Here, a new trajectory is being constructed by concatenating together the $Proj'^*(V, P, I, (n_i, k_i), \sigma_i)$ each of which is either a single pair or an empty sequence. Having defined the auxiliary function, $Proj^*$, we are finally in a position to define unified equivalence.

Definition 22 (Unified Equivalence) *Unified equivalence is a function, $\mathcal{U}$, that given*

- *a set of program states, S,*
- *a set of variables V*
- *a set of set of (line-number, natural number) pairs P, and*
- *a function X from sets of line-numbers $\times$ sets of line numbers $\to$ sets of line-numbers*

returns the equivalence relation $\mathcal{U}(S, V, P, X)$ on programs defined as follows:

$$p \quad \mathcal{U}(S, V, P, X) \quad q$$
$$\textit{iff}$$
$$\forall \sigma \in S : Proj^*_{(V,P,X(\overline{p},\overline{q}))}(T^\sigma_p) = Proj^*_{(V,P,X(\overline{p},\overline{q}))}(T^\sigma_q).$$

where $Proj^*$ is the function given in Definition 21. and the notation $\overline{p}$ represents the set of all line-numbers of program p.

By instantiating this definition with appropriate parameters we express the semantic aspect of different forms of slicing. In [6–8], by instantiating unified equivalence with appropriate parameters, eight different forms of slicing are defined. Each of the following definitions correspond to the semantic equivalence function part in the definition of the eight different forms of slicing:

Definition 23 (Eight Forms of Slicing)

Static slicing:	$\mathcal{S}$	$= \lambda(V, n) \cdot \mathcal{U}(\mathbf{\Sigma}, V, \{n\} \times \mathbf{N}, \varepsilon)$
Dynamic slicing:	$\mathcal{D}$	$= \lambda(\sigma, V, n) \cdot \mathcal{U}(\{\sigma\}, V, \{n\} \times \mathbf{N}, \varepsilon)$
Static iteration count slicing:	$\mathcal{S}_i$	$= \lambda(V, n, k) \cdot \mathcal{U}(\mathbf{\Sigma}, V, \{(n, k)\}, \varepsilon)$
Dynamic iteration count slicing:	$\mathcal{D}_i$	$= \lambda(\sigma, V, n, k) \cdot \mathcal{U}(\{\sigma\}, V, \{(n, k)\}, \varepsilon)$
Static KL slicing:	$\mathcal{S}_{KL}$	$= \lambda(V, n) \cdot \mathcal{U}(\mathbf{\Sigma}, V, \{n\} \times \mathbf{N}, \cap)$
Dynamic KL slicing:	$\mathcal{D}_{KL}$	$= \lambda(\sigma, V, n) \cdot \mathcal{U}(\{\sigma\}, V, \{n\} \times \mathbf{N}, \cap)$
Static KL iteration count slicing:	$\mathcal{S}_{KLi}$	$= \lambda(V, n, k) \cdot \mathcal{U}(\mathbf{\Sigma}, V, \{(n, k)\}, \cap)$
Dynamic KL iteration count slicing:	$\mathcal{D}_{KLi}$	$= \lambda(\sigma, V, n, k) \cdot \mathcal{U}(\{\sigma\}, V, \{(n, k)\}, \cap)$

where $\mathbf{\Sigma}$ is the set of all possible initial states, σ is a single state, k is a natural number, $\mathbf{N}$ is the set of natural numbers, V is a set of variables, and for every set of line numbers, x and y, $\varepsilon(x, y) = \emptyset$, and $\cap$ denotes the set intersection operation.

3.7 A Description of the Eight Forms of Slicing

In the traditional static formulation for slicing, for example, the set of states of interest is the set of all possible states, $\mathbf{\Sigma}$. The set of variables, V and the point in the program n are those of the traditional static slicing criterion. For traditional static slicing, the slicing process must preserve the behaviour of the program at the point of interest n, and for each possible execution of n (hence $n \times \mathbf{N}$ above). However, the traditional definition of static slicing makes no requirement on the way in which the slice must be computed (hence ε above). On the other hand, KL–slicing does not require a slice to behave for all possible initial states the same way as the original program does, but only for a specific one, σ. Moreover, the point of interest is only one occurrence of a statement, (n, k). Contrary to the traditional static slicing, KL–slicing does care about the path of execution in the slice, thus parameter I of $Proj^*$ is $\overline{p} \cap \overline{q}$. In Figure 2, the program performs no input, so the relation $\mathcal{U}$ for this program will not be affected by different choices of the first parameter. The set of variables, V is $\{\mathtt{y}\}$, n is set to 7 and k to 1. So for all states σ, $p\ \mathcal{U}(\sigma, V, \{(n, k)\}, \varepsilon)\ q$ but $\neg(p\ \mathcal{U}(\sigma, V, \{(n, k)\}, \cap)\ q)$. That is, the fourth parameter of $\mathcal{U}$, which captures the presence or absence of the KL requirement, is sensitive to the difference in the two programs p and q in Figure 2. Observe that for both

1	`x=1;`	1	`x=1;`
2	`x=2;`		
3	`if (x>1)`	3	`if (x>1)`
4	`  y=1;`	4	`  y=1;`
5	`else`	5	`else`
6	`  y=1;`	6	`  y=1;`
7	`z=y;`	7	`z=y;`
Program p		Program q	

$V = \{\mathtt{y}\}, n = 7, k = 1$

Fig. 2. Example to capture the difference between KL and non-KL equivalence relations.

1	`x=1;`	1	`x=1;`
2	`while (x<=2) {`	2	`while (x<=2) {`
3	`  y=1;`	3	`  y=1;`
4	`  if (x==1)`		
5	`    y=2;`		
6	`  z=y;`	6	`  z=y;`
7	`  x++;`	7	`  x++;`
8	`}`	8	`}`
Program p		Program q	

$V = \{\mathtt{y}\}, n = 6, k = 2$

Fig. 3. Example to capture the difference between iteration count aware and iteration count unaware equivalence relations.

programs, the final value of y is 1, regardless of how the program is executed. However, the trajectory followed by the program q differs from that followed by p *even* when the two trajectories are restricted to those nodes which occur in both programs; it seems that q arrives at the same answer as p but in a different way.

The requirement that a slice observes this (stringent) requirement for equivalence is similar to the path equivalence studied in the context of program restructuring [50,60]. It is useful in the context of debugging however. When slicing is applied to debugging, it is important that the sliced program faithfully reproduces the behaviour that causes a fault to manifest itself as an error. For this reason, program q would not be a useful slice of program p in Figure 2. In this regard, the KL requirement is important for debugging applications of slicing [49,57]. It may also be important in applications to program comprehension [30,52], because, in these applications, the programmer typically tries to understand the behaviour of the original program in terms of the behaviour of the slice. However, for other applications, such as testing, reuse, and restructuring [3,15,41], the KL requirement is unimportant because program modification is inherent to these application areas.

To see how the iteration count can affect the meaning of the equivalence preserved by slicing, consider the program in the left-hand column of Figure 3. In this program, the conditional at line numbers 4 and 5 can only affect the value of `y` at Line 6 on the first time it is executed. Therefore, choosing the second iteration of this statement in the slicing criterion, will allow the conditional to be deleted. That is, in terms of equivalence, for all states σ, $p\ \mathcal{U}(\sigma, \{\mathtt{y}\}, \{(6,2)\}, \cap)\ q$ and $p\ \mathcal{U}(\sigma, \{\mathtt{y}\}, \{(6,2)\}, \varepsilon)\ q$.

When slicing is applied to debugging, the iteration count will be of interest, but in other applications it is unlikely to be of interest. This is because debugging typically starts when the program fails due to a fault. To locate the fault, a slice can be constructed. Of course, it would be sensible to take into account the iteration count for the statement which reveals the error when constructing the slice; this may reduce the size of the slice, thereby reducing debugging effort.

Although it was (implicitly) introduced as part of Korel and Laski's dynamic slicing criterion, the iteration count concept is independent of whether a slice is to be static or dynamic. The same is true of the KL requirement. This can be seen from the fact that no input was necessary in the two examples used to illustrate the difference in equivalence relations produced by including or excluding these two requirements.

Furthermore, it is possible to find static computations in which the iteration count is an interesting and useful concept. For example, in loop carried dependence, it may take several iterations of a loop in order to propagate a dependence from one point to another. An example of this is the program which computes values in the Fibonacci sequence in Figure 4. This program performs no input. In the example, the ability to focus upon different iteration counts allows the dependence structure to be examined in more detail; it becomes possible to see how dependence grows with each loop iteration. In this example, on the first iteration the value of the variable `prev` does not depend on the assignment to `curr` at Line 6, but it does on the second (and subsequent iterations). As this example shows, the concept of an iteration count may be a useful slicing criterion in its own right.

Finally, consider the example in Figure 5, this illustrates the traditional difference between static and dynamic slicing. That is, for dynamic slicing the input affects the outcome of slicing, while for static slicing, the slice must be correct for all possible initial states. This is the difference between static and dynamic slicing to which most authors [2,14] refer. However, as the preceding discussion shows, there are two other aspects to a dynamic slice: path equivalence (or otherwise) and iteration count sensitivity (or otherwise).

1 `prev=1;`		1 `prev=1;`
2 `curr=1;`	2 `curr=1;`	2 `curr=1;`
3 `i=1;`		3 `i=1;`
4 `while (i<n) {`		4 `while (i<n) {`
5 `oldc=curr;`	5 `oldc=curr;`	5 `oldc=curr;`
6 `curr=curr+prev;`		6 `curr=curr+prev;`
7 `prev=oldc;`	7 `prev=oldc;`	7 `prev=oldc;`
8 `i++;`		8 `i++;`
9 `}`		9 `}`
Program p	$V = \{\texttt{prev}\}, n = 7, k = 1$	$V = \{\texttt{prev}\}, n = 7, k > 1$

Fig. 4. Example where iteration count is interesting in a static computation

1 `y=1;`	1 `y=1;`
2 `x=input();`	
3 `if (x>1)`	
4 `y=2;`	
5 `z=y;`	5 `z=y;`
Program p	Program q

$\sigma = \langle 1 \rangle, V = \{\texttt{y}\}, n = 5, k = 1$

Fig. 5. Example to capture the difference between static and dynamic equivalence relations.

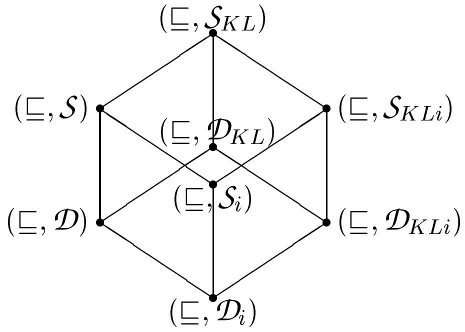

Fig. 6. The *Weaker than* relation between the eight forms of slicing. Form of slicing A is below form of slicing B in the picture implies A is weaker than B hence by, Lemma 19, the minimal slices of A are smaller than than the minimal slices of B.

3.8 Comparing the Eight Forms of Slicing

As discussed in Section 3.4, in order to compare the eight different forms of slicing, it is necessary to define a slicing criterion comparison relation. In this case, we call the slicing criterion comparison relation $\xrightarrow{1}$. It is defined by the following rules:

Definition 24 (The Slicing Criterion Comparison Relation , $\xrightarrow{1}$)

- $(\sigma, V, n) \xrightarrow{1} (V, n)$ *for all* σ, V, n. *(For comparing* $\mathcal{D}$ *to* $\mathcal{S}$ *and* $\mathcal{D}_{KL}$ *to* $\mathcal{S}_{KL}$*.)*
- $(V, n, k) \xrightarrow{1} (V, n)$ *for all* V, n, k *(For comparing* $\mathcal{S}_i$ *to* $\mathcal{S}$ *and* $\mathcal{S}_{KLi}$ *to* $\mathcal{S}_{KL}$*.)*
- $(\sigma, V, n, k) \xrightarrow{1} (V, n, k)$ *for all* σ, V, n, k *(For comparing* $\mathcal{D}_i$ *to* $\mathcal{S}_i$ *and* $\mathcal{D}_{KL}$ *to* $\mathcal{S}_{KLi}$*.)*
- $(\sigma, V, n, k) \xrightarrow{1} (\sigma, V, n)$ *for all* σ, V, n, k *(For comparing* $\mathcal{D}_i$ *to* $\mathcal{D}$ *and* $\mathcal{D}_{KLi}$ *to* $\mathcal{D}_{KL}$*.)*
- $(\sigma, V, n, k) \xrightarrow{1} (V, n)$ *for all* σ, V, n, k. *(For comparing* $\mathcal{D}_i$ *to* $\mathcal{S}$ *and* $\mathcal{D}_{KLi}$ *to* $\mathcal{S}_{KL}$*.)*

This is an extension of Definition 12. An important result of [7,8] is to show that the eight forms of slicing form the lattice shown in Figure 6. Form of slicing A is below form of slicing B implies A is weaker than B. This result follows from the unified equivalence lemma (Lemma 25).

Lemma 25 (Unified Equivalence Lemma) *Given sets of initial states* S_1 *and* S_2*, sets of variables* V_1 *and* V_2*, sets of points of interests* P_1 *and* P_2 *and functions of pairs of line number sets* X_1 *and* X_2 *such that*

$$S_1 \subseteq S_2,\ \ V_1 \subseteq V_2,\ \ P_1 \subseteq P_2, \quad \textit{and}\ \forall p, q : X_1(\overline{p}, \overline{q})\ \subseteq\ X_2(\overline{p}, \overline{q})$$

$$\textit{then}$$

$$p\ \mathcal{U}(S_2, V_2, P_2, X_2)\ q \implies p\ \mathcal{U}(S_1, V_1, P_1, X_1)\ q.$$

The Duality Lemma 19 and the Unified Equivalence Lemma 25 imply that in Figure 6, form of slicing A is below form of slicing B implies that the minimal slices of A are smaller than (using Definition 17) the minimal slices of B.

As an example, we show how static slicing and dynamic iteration count slicing are compared. The way they map slicing criteria to parameters of unified equivalence is given by:

$$\begin{array}{llll} \mathcal{S} & \text{maps} & (V, n) & \text{to} & (\mathbf{\Sigma},\ \ V,\ \{n\} \times \mathbf{N},\ \varepsilon) \\ \mathcal{D}_i & \text{maps} & (\sigma, V, n, k) & \text{to} & (\{\sigma\},\ V,\ \{(n, k)\},\ \ \varepsilon). \end{array}$$

Relevant slicing criteria comparison in this case is $(\sigma, V, n, k) \xrightarrow{1} (V, n)$ for all σ, V, n, k. To prove $\mathcal{D}_i$ is weaker than $\mathcal{S}$ all that is required to prove is that for all σ, V, n, k:

(i) $\{\sigma\} \subseteq \mathbf{\Sigma}$
(ii) $V \subseteq V$
(iii) $\{(n, k)\} \subseteq \{n\} \times \mathbf{N}$ and
(iv) $\forall p, q : \varepsilon(\overline{p}, \overline{q})\ \subseteq\ \varepsilon(\overline{p}, \overline{q})$

.

All these results are trivially true. If we use traditional syntactic ordering, or any other ordering where all sets have a minimal element, the duality lemma (Lemma 19) tells us that the minimal slices of dynamic iteration count slicing will never be larger than those of static slicing.

Lemma 19 expresses the general result that provided that we slice with respect to a syntactic ordering of sufficient strength (*i.e.* that all set have at least one minimal element with respect to the ordering) one form of slicing is weaker than another implies that the weaker one has smaller minimal slices.

Lemma 25, highlights the power of using unified equivalence for expressing forms of slicing. Having expressed a form of slicing in these terms makes it fairly straightforward to compare it for strength with other forms which have been similarly expressed.

In [7], by doing a case analysis, it is further proved that for the forms of slicing in Figure 6 the implications also go the other way: *i.e.* if A is not above B then the forms of slicing are not comparable by strength or by size.

These results reveal the intricate structure which underlies the definitions of different forms of slicing. They also highlight the presence of previously implicit criteria for slicing, for example, the iteration count of Korel and Laski's dynamic slicing, buried in the accepted definitions of a slice. This is both theoretically interesting and practically important. One practical value of the framework is that it allows new forms of slicing to be defined by combining properties of traditional definitions. Such forms of slicing although not yet implemented, almost certainly will have useful applications. Further investigation of these applications will be the subject of future work.

4 Simultaneous Slicing

In this section, various forms of simultaneous slicing are defined and compared using unified equivalence. We introduce the notion of one form of slicing being a *generalisation* of another. This is a further development of the theoretical framework. It is proved that the simultaneous forms are generalisations of the forms of slicing defined in [6–8]. The results of [6–8] relating strength and minimality are thus extended to a wider class of forms of slicing.

In simultaneous slicing, a slice is taken at many points in the program simultaneously. The slicing criterion for simultaneous slicing, rather than being a set of variables, V and the point in the program n, as in traditional slicing, is a set of variables, V and a non–empty *set*, $\{n_1, \ldots, n_m\}$ of program–points.

1	`a=input();`	1	`a=input();`
2	`b=input();`	2	`b=input();`
3	`t=a+b;`	3	`t=a+b;`
4	`z=a;`		
5	`printf("%d",t);`	5	`printf("%d",t);`
6	`t=a-b;`	6	`t=a-b;`
7	`printf("%d",t);`	7	`printf("%d",t);`
Program p		Program q	

Fig. 7. In decomposition slicing, all points of program p which output the value of variable t are required. To achieve this, we slice with respect to variable t at points 5 and 7 simultaneously.

An example of the use of simultaneous slicing[25] is in decomposition slicing [33]. Gallagher and Lyle [33] show how a decomposition slice can be built for each variable of a program. An example of simultaneous slicing is given in Figure 7. Here we want to slice at all points at which the value of the variable, t, is output, namely lines 5 and 7.

4.1 *Defining Simultaneous Slicing Using Unified Equivalence*

As simultaneous slicing was originally formulated [25,28], it is a static form of path unaware slicing and therefore the S component in the unified equivalence will be instantiated to $\mathbf{\Sigma}$, the set of all program states, and the X component to the function ε, which maps pairs of sets of line–numbers to the empty set. We use $\mathcal{SS}$ to stand for simultaneous slicing. Using unified equivalence, it can be defined as follows:

$$\mathcal{SS} = \lambda(V, \{n_1, \ldots, n_m\}) \cdot \mathcal{U}(\mathbf{\Sigma}, V, \{n_1, \ldots, n_m\} \times \mathbf{N}, \varepsilon)$$

Having defined static, non–path aware, simultaneous slicing, $\mathcal{SS}$, the universal equivalence $\mathcal{U}$, allows us to define a whole range of forms of simultaneous slicing including:

- $\mathcal{SD}$: Dynamic Simultaneous Slicing,
- $\mathcal{SD}_{KL}$ and $\mathcal{SS}_{KL}$: Dynamic and Static Path Aware (KL) Simultaneous Slicing,
- $\mathcal{SD}_i$ and $\mathcal{SS}_i$: Dynamic and Static Iteration Count Simultaneous Slicing, and finally,
- $\mathcal{SD}_{KLi}$ and $\mathcal{SS}_{KLi}$: Dynamic and Static Path Aware Iteration Count Simultaneous Slicing.

4.1.1 The Slicing Criteria for different forms of Simultaneous Slicing

As in the forms of slicing discussed in [6–8], for all forms of dynamic simultaneous slicing there will be an extra parameter in the slicing criterion namely a state σ.

For iteration count simultaneous slicing, there will be a set of line–number natural–number pairs, $\{(n_1, k_1), \ldots, (n_m, k_m)\}$ representing that the program and the slice must agree at the k_ith iteration of line–number n_i for all i in $\{1, \ldots, m\}$. Note that it is possible that the same line number n_i, say, occurs more than once in $\{(n_1, k_1), \ldots, (n_m, k_m)\}$. For example we may be interested in the 7th and the 23rd iteration of line-number 5.

We can now define each form of simultaneous slice in terms of the unified equivalence as follows:

Definition 26 (Forms of Simultaneous Slice)

$$
\begin{aligned}
\mathcal{SD} &= \lambda(\sigma, V, C) \cdot \mathcal{U}(\{\sigma\}, V, C \times \mathbf{N}, \varepsilon) \\
\mathcal{SS} &= \lambda(V, C) \cdot \mathcal{U}(\mathbf{\Sigma}, V, C \times \mathbf{N}, \varepsilon) \\
\mathcal{SD}_i &= \lambda(\sigma, V, C') \cdot \mathcal{U}(\sigma, V, C', \varepsilon) \\
\mathcal{SS}_i &= \lambda(V, C') \cdot \mathcal{U}(\mathbf{\Sigma}, V, C', \varepsilon) \\
\mathcal{SD}_{KL} &= \lambda(\sigma, V, C) \cdot \mathcal{U}(\sigma, V, C \times \mathbf{N}, \cap) \\
\mathcal{SS}_{KL} &= \lambda(V, C) \cdot \mathcal{U}(\mathbf{\Sigma}, V, C \times \mathbf{N}, \cap) \\
\mathcal{SD}_{KLi} &= \lambda(\sigma, V, C') \cdot \mathcal{U}(\{\sigma\}, V, C', \cap) \\
\mathcal{SS}_{KLi} &= \lambda(V, C') \cdot \mathcal{U}(\mathbf{\Sigma}, V, C', \cap)
\end{aligned}
$$

where $C = \{n_1, \ldots, n_m\}$ *and* $C' = \{(n_1, k_1), \ldots, (n_m, k_m)\}$.

4.2 Generalisation

Intuitively, simultaneous slicing is a generalisation of non–simultaneous slicing because it allows the slicing criterion to consist of one or more points as opposed to exactly one. Before we can make such claims, it is necessary to define formally, what is meant for one form of slicing to be a generalisation of another.

Definition 27 (Generalisation) *We define form of slicing* $(\lesssim, E_A)$ *to be a generalisation of form of slicing* $(\lesssim, E_B)$ *if and only if there is a function,* g*, say, from the slicing criteria of* $(\lesssim, E_B)$ *to the slicing criteria of* $(\lesssim, E_A)$ *such* *that $E_A(g(c)) = E_B(c)$ for all slicing criteria, c, of $(\lesssim, E_B)$. We call g the generalisation function between the two forms.*

We can now formalise our intuition above:

Lemma 28 *Each of the eight forms of simultaneous slicing is a generalisation of the corresponding form of static and dynamic slicing.*

PROOF. In this case the generalisation function, g, between the slicing criteria of non–simultaneous forms of slicing given in Definition 23 and the simultaneous forms given in Definition 26 is defined as follows:

- $g(\sigma, V, n) = (\sigma, V, \{n\})$
- $g(V, n) = (V, \{n\})$
- $g(\sigma, V, n, k) = (\sigma, V, \{(n, k)\})$
- $g(V, n, k) = (V, \{(n, k)\})$

Each of the eight cases must be proved separately. We prove just one as an illustration of the method. The other cases are proved similarly. We prove that $(\sqsubseteq, \mathcal{SS})$ is a generalisation of $(\sqsubseteq, \mathcal{S})$. The generalisation function for this case will map the slicing criteria (V, n) of $\mathcal{S}$ to the slicing criteria $(V, \{n\})$ of $\mathcal{SS}$. By Definition 27, we must prove that for all (V, n),

$$\mathcal{SS}(V, \{n\}) = \mathcal{S}(V, n)$$

LHS=$\mathcal{U}(\mathbf{\Sigma}, V, \{n\} \times \mathbf{N}, \varepsilon)$ by Definition 26.
RHS=$\mathcal{U}(\mathbf{\Sigma}, V, \{n\} \times \mathbf{N}, \varepsilon)$ by Definition 23.

LHS=RHS as required.

4.3 Comparing forms of Simultaneous Slicing

In this section, we compare forms of simultaneous slicing both with each other and with the different forms of slicing given in Definition 23.

Lemma 29 *The lattice obtained from comparing simultaneous slices is given by the right hand cube in Figure 8.*

PROOF. Following the approach introduced in Section 3 in order to compare different forms of simultaneous slicing above, we first need to define the slicing criterion comparison relation between the different forms. In this case, the slicing criterion comparison relation, $\xrightarrow{2}$, is defined as follows:

$$
\begin{array}{ll}
(\sigma, V, \{n_1, \ldots, n_m\}) & \stackrel{2}{\rightarrow} (V, \{n_1, \ldots, n_m\}) \\
(V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) & \stackrel{2}{\rightarrow} (V, \{n_1, \ldots, n_m\}) \\
(\sigma, V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) & \stackrel{2}{\rightarrow} (V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) \\
(\sigma, V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) & \stackrel{2}{\rightarrow} (V, \{n_1, \ldots, n_m\})
\end{array}
$$

Using the unified equivalence lemma (Lemma 25), by inspection, it can be seen that the only difference occurs in P, the third parameter in the unified equivalence. For the original forms of slicing we had

$$\{(n, k)\} \subseteq \{n\} \times \mathbf{N}.$$

and for simultaneous slicing the corresponding result follows because

$$\{(n_1, k_1), \ldots, (n_m, k_m)\} \subseteq \{n_1, \ldots, n_m\} \times \mathbf{N}.$$

Lemma 30 *The lattice obtained from comparing simultaneous slices given by the right hand cube in Figure 8 is strict. i.e. None of the forms of slicing in the right hand cube in Figure 8 are equal.*

PROOF. By Lemma 28, each form of simultaneous slicing is a generalisation of the corresponding form in Definition 23. It was proved in [7] that the corresponding lattice for the forms given in Definition 23 is strict. ¿From this it follows that the lattice obtained from comparing simultaneous slices is given by the right hand cube in Figure 8 is strict.

Lemma 31 *Each form of simultaneous slicing is stronger than its corresponding standard form. In other words, each form of slicing in right hand cube in Figure 8 is stronger than the corresponding form in the left hand cube connected to it by the dotted line.*

PROOF. In order to prove this relationship it is necessary, again, to define the slicing criterion comparison relation, $\stackrel{3}{\rightarrow}$, between the forms of slicing being compared. Informally, $\stackrel{3}{\rightarrow}$ will map the $\{n_1, \ldots, n_m\}$ in non–iteration count simultaneous slicing criteria to all the single elements n_i and it will map $\{(n_1, k_1), \ldots, (n_m, k_m)\}$ in iteration count simultaneous slicing criteria to all the single pairs (n_i, k_i). Formally, for all $i \in \{1, \ldots, m\}$:

$$
\begin{array}{ll}
(V, \{n_1, \ldots, n_m\}) & \stackrel{3}{\rightarrow} (V, n_i) \\
(V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) & \stackrel{3}{\rightarrow} (V, n_i, k_i) \\
(\sigma, V, \{n_1, \ldots, n_m\}) & \stackrel{3}{\rightarrow} (\sigma, V, n_i) \\
(\sigma, V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) & \stackrel{3}{\rightarrow} (\sigma, V, n_i, k_i)
\end{array}
$$

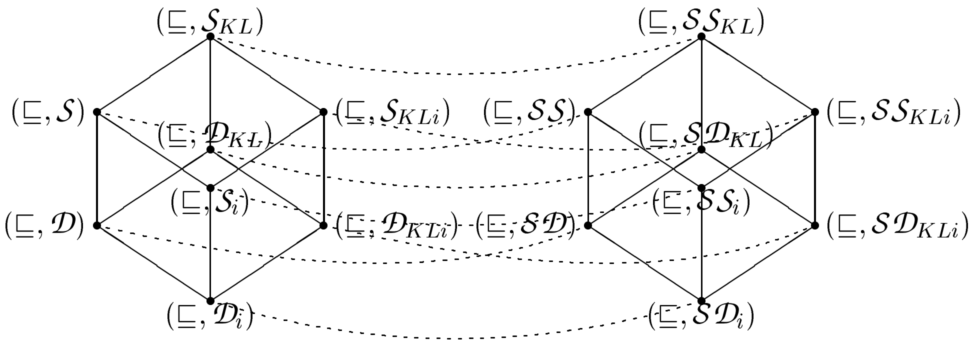

Fig. 8. A diagram representing the *weaker than* relation between the eight forms of simultaneous slicing in the right cube. It also shows the relationship between simultaneous slicing and the conventional forms. Form of slicing A is weaker than B if A is connected by a dotted line to B and is to the left of B or A is below B.

Having defined $\stackrel{3}{\rightarrow}$, we again appeal to the unified equivalence lemma (Lemma 25) to prove each case. As an example, we prove that

$$(\lesssim, \mathcal{D}) \stackrel{\stackrel{3}{\rightarrow}}{\subseteq} (\lesssim, \mathcal{S}\mathcal{D}).$$

By Definition 26, $\mathcal{S}\mathcal{D} = \lambda(\sigma, V, \{n_1, \ldots, n_m\}) \cdot \mathcal{U}(\{\sigma\}, V, \{n_1, \ldots, n_m\} \times \mathbf{N}, \varepsilon)$

and by Definition 23, $\mathcal{D} = \lambda(\sigma, V, n) \cdot \mathcal{U}(\{\sigma\}, V, \{n\} \times \mathbf{N}, \varepsilon)$

If we pick slicing criteria that are suitably related by $\stackrel{3}{\rightarrow}$, namely $(\sigma, V, \{n_1, \ldots, n_m)\} \stackrel{3}{\rightarrow} (\sigma, V, n_i)$ for some $i \in \{1, \cdots, m\}$, we must show that

$$\mathcal{U}(\{\sigma\}, V, \{n_i\} \times \mathbf{N}, \varepsilon) \subseteq \mathcal{U}(\{\sigma\}, V, \{n_1, \ldots, n_m\} \times \mathbf{N}, \varepsilon).$$

This follows immediately from the unified equivalence lemma (Lemma 25) because

$$n_i \subseteq \{n_1, \ldots, n_m\}.$$

All the other cases are proved similarly.

Lemma 32 (Minimal Slices for Simultaneous Slicing)

(i) The lattice obtained from comparing minimal slices for simultaneous slicing is given by the right hand cube in Figure 8.
(ii) The lattice is strict.
(iii) The minimal slices of each form of simultaneous slicing are not smaller than the corresponding standard form.

PROOF. These results (i) and (iii) follow immediately from the duality lemma (Lemma 19). Result (ii) follows immediately from Lemma 28.

In this section, different forms of simultaneous slicing have been formally defined (Definition 26) using unified equivalence. It has been proved (Lemma 28) that each form of simultaneous slicing is a generalisation of the corresponding standard form given in Definition 23. It has been proved (Lemma 29) that the different forms of simultaneous slice give rise to a lattice isomorphic to the lattice of standard forms given in Definition 23 and that the lattice is strict (Lemma 30). It is proved that each form of simultaneous slicing is stronger than it corresponding standard form (Lemma 31). Finally (Lemma 32) it is proved that the lattice obtained from comparing minimal slices for simultaneous slicing is isomorphic to that of the standard case, the lattice is strict, and the minimal slices of each form of simultaneous slicing are larger than the corresponding standard forms of Definition 23.

5 Conditioned Slicing

The static and dynamic paradigms represent two extremes - either we say nothing about the input to the program (static slicing) or we say everything (dynamic slicing). Conditioned slicing, introduced by De Lucia et al. [14,22,30,31] allows us to bridge the gap. It allows us to provide information to the slicing tool about the input without being so specific as to give the precise values. As an example of the way in which conditioning identifies sub-programs, consider the Taxation program in Figure 9. The figure contains a fragment of a program which encodes a previous year's UK tax regulations. Each person has a personal allowance which is an amount of un-taxed income. The size of this personal allowance depends upon the status of the person, which is encoded in the boolean variables `blind`, `married` and `widow`, and the integer variable `age`. For example, given the condition

```
age>=65  &&  age<75  &&  income==36000  &&  blind==0
                     &&  married==1
```

the conditioned sliced program, with respect to the variable `tax` at the last line of the program, gives rise to the statements which appear boxed in the figure. This is useful because it allows the software engineer to isolate a subcomputation concerned with the initial condition of interest. The sub-program extracted can be compiled and executed as a separate code unit. It will be guaranteed to mimic the behaviour of the original if the initial condition is met.

By choosing this condition to be simply the constant predicate 'true', the definition of conditioned slicing becomes that of static slicing and by making it a conjunction of equalities, it is possible to mimic the effect of a dynamic slicing. These observation have led several authors to observe that conditioned

```
if (age >= 75)
    personal = 5980;
else if (age >= 65)
    personal = 5720;
else
    personal = 4335;
if (age >= 65 && income > 16800)
    if (4335 > personal-((income-16800)/2))
        personal = 4335;
    else
        personal = personal-((income-16800)/2);
if (blind == 1)
    personal = personal + 1380;

if (married == 1 && age >= 75)
    pc10 = 6692;
else if (married == 1 && age >= 65)
    pc10 = 6625;
else if (married == 1 || widow == 1)
    pc10 = 3470;
else
    pc10 = 1500;
if (married == 1 && age >= 65 && income > 16800)
    if (3470 > pc10-(income-16800)/2)
        pc10 = 3470;
    else
        pc10 = pc10-((income-16800)/2);

if (income - personal <= 0)
    tax = 0;
else
    income = income - personal;
if (income <= pc10)
{
    tax = pc10 * rate10;
    income = income - pc10;
}
if (income <= 28000)
    tax = tax + income * rate23;
else
{
    tax = tax + 28000 * rate23;
    income = income - 28000;
    tax = tax + income * rate40;
}
```

Conditioned Sliced program: boxed lines of code

Condition: `age>=65 && age<75 && income==36000 && blind==0 && married==1`

Variable: `tax`   **Slice point**: Last Line of program.

Fig. 9. A Fragment of the Taxation Calculation Program

slicing “subsumes” static and dynamic slicing [14,22,31]. However, this use of the term “subsumes” differs from the one used in our previous work [6–8]. Using the terminology of this paper conditioned slicing is *a generalisation* of static and dynamic slicing. We prove this formally in this section.

5.1 Defining Conditioned Slicing using Unified Equivalence

In conditioned slicing the slicing criterion will contain an expression representing the set of initial states over which the program and its slice must agree. We take the opportunity to incorporate the generalisations obtained by considering simultaneous slicing into our definition of conditioned slicing. So, in fact, here we are defining simultaneous conditioned slicing, which is a generalisation of conditioned slicing.

For conditioned slicing, the slicing criterion has a parameter Σ', representing a non–empty set of initial states for which the program and its slice must agree. We will show that this is a generalisation of both dynamic slicing, where just a single initial state is considered and static slicing where all initial states are considered.

Definition 33 *Four Forms of Conditioned Slicing*

$$\begin{aligned}
\mathcal{C} &= \lambda(\Sigma', V, C) \cdot \mathcal{U}(\mathbf{\Sigma}, V, C \times \mathbf{N}, \varepsilon), \\
\mathcal{C}_i &= \lambda(\Sigma', V, C') \cdot \mathcal{U}(\mathbf{\Sigma}, V, C', \varepsilon), \\
\mathcal{C}_{KL} &= \lambda(\Sigma', V, C) \cdot \mathcal{U}(\mathbf{\Sigma}, V, C \times \mathbf{N}, \cap), \\
\mathcal{C}_{KLi} &= \lambda(\Sigma', V, C') \cdot \mathcal{U}(\mathbf{\Sigma}, V, C', \cap),
\end{aligned}$$

where $C = \{n_1, \ldots, n_m\}$ *and* $C' = \{(n_1, k_1), \ldots, (n_m, k_m)\}$ *and* $\Sigma' \subseteq \mathbf{\Sigma}$.

5.2 Comparing the Forms of Conditioned Slicing With Each Other

In order to compare the forms of conditioned slicing, it is necessary, again, to define the slicing criterion comparison relation between the four forms of conditioned slicing being compared. In this case, we call slicing criterion comparison relation $\stackrel{4}{\rightarrow}$. It is defined by the following rules:

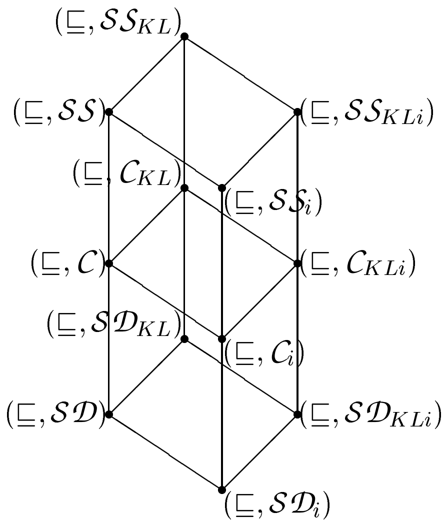

Fig. 10. A diagram representing the *weaker than* relation between the four forms of conditioned slicing in the middle diamond. It also shows the relationship between conditioned slicing and simultaneous forms. Form of slicing A is weaker than B if A is below B.

$$
\begin{array}{lll}
(\Sigma', V, \{n_1, \ldots, n_m\}) & \overset{4}{\to} (\Sigma'', V, \{n_1, \ldots, n_m\}) & \text{where } \Sigma' \subseteq \Sigma'' \\
(\Sigma', V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) & \overset{4}{\to} (\Sigma'', V, \{n_1, \ldots, n_m\}) & \text{where } \Sigma' \subseteq \Sigma'' \\
(\Sigma', V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) & \overset{4}{\to} (\Sigma'', V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) & \text{where } \Sigma' \subseteq \Sigma'' \\
(\Sigma', V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) & \overset{4}{\to} (\Sigma'', V, \{n_1, \ldots, n_m\}) & \text{where } \Sigma' \subseteq \Sigma''
\end{array}
$$

Using the transitive closure of $\overset{4}{\to}$, the unified equivalence Lemma shows us that, all forms of conditioned slicing are stronger than the corresponding form of dynamic slicing and weaker than the corresponding form of static slicing. The duality lemma (Lemma 19) shows us also that if we use the traditional syntactic ordering to construct the slices, the minimal slices of conditioned slicing will never be smaller than the corresponding dynamic slices and never bigger than the corresponding static slices.

Lemma 34 *Each of the four forms of conditioned slicing is a generalisation of the corresponding form of static and dynamic slicing.*

PROOF. There are four cases to prove. As in Lemma 28, we just prove one of them as the others can be proved similarly. We prove that $(\sqsubseteq, \mathcal{C}_{KLi})$ is a generalisation of $(\sqsubseteq, \mathcal{SS}_{KLi})$. In this case the generalisation function will map slicing criteria (V, C') of $(\sqsubseteq, \mathcal{SS}_{KLi})$ to $(\mathbf{\Sigma}, V, C')$ of $(\sqsubseteq, \mathcal{C}_{KLi})$, where C' is of the form $\{(n_1, k_1), \ldots, (n_m, k_m)\}$ and $\mathbf{\Sigma}$ is the set of all states. We must prove that:

$$\mathcal{SS}_{KLi}(V, C') = \mathcal{C}_{KLi}(\mathbf{\Sigma}, V, C')$$

LHS=$\mathcal{U}(\mathbf{\Sigma}, V, C', \cap)$ by Definition 26 and
RHS=$\mathcal{U}(\mathbf{\Sigma}, V, C', \cap)$ by Definition 33.
LHS=RHS as required.

Lemma 35 *The lattice obtained from comparing conditioned slices is given by the middle diamond in Figure 10.*

PROOF. This follows immediately from the unified equivalence lemma (Lemma 25).

Lemma 36 *The lattice obtained from comparing conditioned slices is given by the middle diamond in Figure 10 is strict. i.e. none of the forms of slicing in the middle diamond in Figure 10 are equal.*

PROOF. By Lemma 34, each form of conditioned slicing is a generalisation of the corresponding form in Definition 26. In Lemma 30, it was proved that the corresponding lattice for the forms given in Definition 23 is strict. ¿From this it follows that the lattice obtained from comparing forms of conditioned slicing given by the right hand cube in Figure 10 is strict.

Lemma 37 *Each form of conditioned slicing is weaker than its corresponding simultaneous static form. In other words, each form of slicing in middle diamond in Figure 10 is weaker than the corresponding form in the top diamond.*

PROOF. In order to prove this relationship, it is necessary, again, to define the slicing criterion comparison relation, $\stackrel{5}{\rightarrow}$, between the forms of slicing being compared. The $\stackrel{5}{\rightarrow}$ is defined as follows:

$$\begin{array}{ll} (\Sigma', V, \{n_1, \ldots, n_m\}) & \stackrel{5}{\rightarrow} (V, \{n_1, \ldots, n_m\}) \\ (\Sigma', V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) & \stackrel{5}{\rightarrow} (V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) \end{array}$$

Having defined $\stackrel{5}{\rightarrow}$, we again appeal to the unified equivalence lemma (Lemma 25) to prove each case.

Lemma 38 *Each form of conditioned slicing is stronger than its corresponding simultaneous dynamic form. In other words, each form of slicing in middle diamond in Figure 10 is stronger than the corresponding form in the bottom diamond.*

PROOF. In order to prove this relationship, it is necessary, again, to define the slicing criterion comparison relation, $\stackrel{6}{\rightarrow}$, between the forms of slicing being compared. The $\stackrel{6}{\rightarrow}$ is defined as follows:

$$
\begin{array}{ll}
(\sigma, V, \{n_1, \ldots, n_m\}) & \xrightarrow{6} (\{\sigma\}, V, \{n_1, \ldots, n_m\}) \\
(\sigma, V, \{(n_1, k_1), \ldots, (n_m, k_m)\}) & \xrightarrow{6} (\{\sigma\}, V, \{(n_1, k_1), \ldots, (n_m, k_m)\})
\end{array}
$$

Having defined $\xrightarrow{6}$, we again appeal to the unified equivalence lemma (Lemma 25) to prove each case.

Lemma 39 (Minimal Slices for Conditioned Slicing)

(i) The lattice obtained from comparing minimal slices for conditioned slicing is given by the middle diamond in Figure 10.
(ii) The lattice is strict.
(iii) The minimal slices of each form of conditioned slicing are smaller than the corresponding static form.
(iv) The minimal slices of each form of conditioned slicing are larger than the corresponding dynamic form.

PROOF. These results (i), (iii) and (iv) follow immediately from the duality lemma (Lemma 19). Result (ii) follows immediately from Lemma 34.

In this section, different forms of conditioned slicing have been formally defined (Definition 33) using unified equivalence. It has been proved (Lemma 34) that each form of conditioned slicing is a generalisation of the corresponding standard form given in Definition 26. It has been proved (Lemma 35) that the different forms of conditioned slice give rise to a lattice isomorphic to the lattice of standard static forms given in Definition 26 and that the lattice is strict (Lemma 36). It is proved that each form of conditioned slicing is weaker than its corresponding static form (Lemma 37) and stronger than its corresponding dynamic form (Lemma 38).

Finally (Lemma 39) it is proved that the lattice obtained from comparing minimal slices for conditioned slicing is isomorphic to the lattice obtained by considering the different forms of static slicing. The lattice is strict, and the minimal slices of each form of conditioned slicing are smaller than the corresponding static forms and larger than the corresponding dynamic forms of Definition 26.

In Figure 11, we illustrate pictorially how all forms of slicing considered in this paper are related from the weakest form: dynamic iteration count slicing, $(\sqsubseteq, \mathcal{D}_i)$, to the strongest: simultaneous, static path aware slicing, $(\sqsubseteq, \mathcal{SS}_{KL})$.

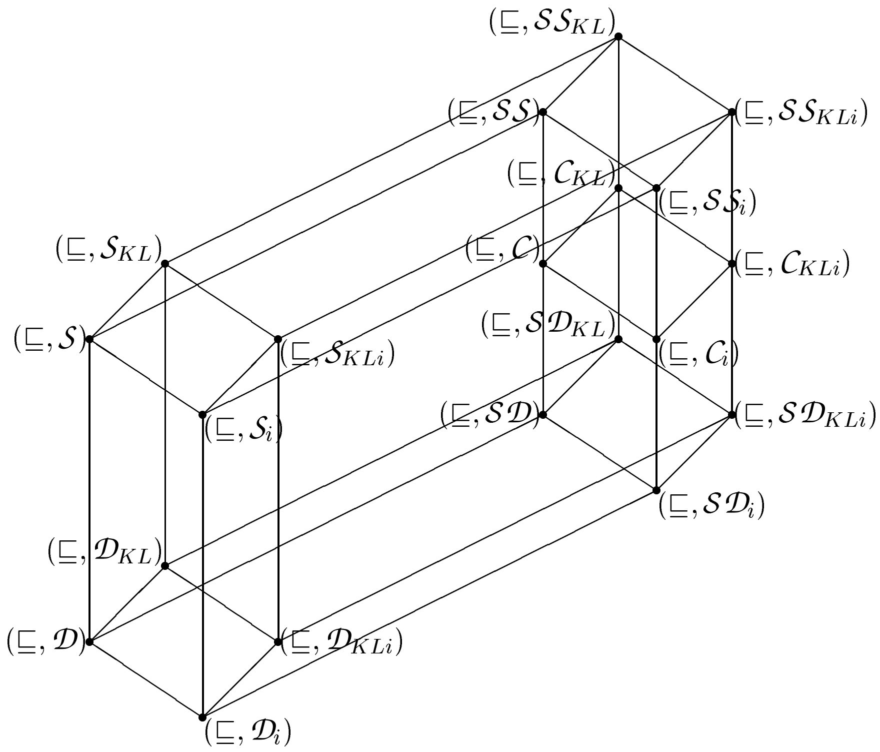

Fig. 11. The *weaker than* relation for all forms of slicing considered in this paper.

6 Related Work

The work presented in this paper is a development of work presented at the 5th SCAM workshop [6] and also builds on our previous work on the formalisation of slice properties and the relationships between slicing criteria [6,7]. This paper extends our previous work to consider slices and minimal slices for both conditioned and simultaneous slicing criteria and their relationship to static and dynamic slicing criteria.

Closest to our work is probably the work of Venkatesh [66] where the aim is to separate definitions of slices from the algorithms which compute them. Venkatesh [66] introduced and formally defines the semantics of a variety of already existing forms of slice as well as introducing some of his own.

He defines a simple procedural language L with assignments, conditionals and loops (all statements being uniquely labelled). He defines a slice to be a set of labels, and defines a function '$syn(s, L)$' which constructs a legal program from a program and a subset of its labels in the obvious way. A set of labels L is a *static* backward end slice of p with respect to variable v if and only if for all states σ

$$\mathcal{M}[\![p]\!]\sigma(v) = \mathcal{M}[\![syn(p, L)]\!]\sigma(v)$$

A *Dynamic* Slice is defined in terms of an initial state σ_0. He defines L to be a dynamic slice with respect to program p and variable v if and only if

$$\mathcal{M}[\![p]\!]\sigma_0(v) = \mathcal{M}[\![syn(p, L)]\!]\sigma_0(v)$$

A static slice is therefore a special form of dynamic slice. However, Venkatesh's version of dynamic slicing is a simplified version, in which the additional features of iteration count and the KL criterion are not present. This simplifies his treatment of the relationship between static and dynamic slicing. In our earlier paper [8], we showed that the relationship between static and dynamic slicing is more involved than had been thought by previous authors (including Venkatesh) because of the effect of the iteration count and KL criterion. Venkatesh's paper appeared before the advent of conditioned slicing and, therefore, Venkatesh does not consider conditioned slicing. However, Venkatesh does introduce an early and more limited form of conditioned slicing, called Quasi Static slicing, in which the set of states in conditioned slicing is replaced by a set of input sequences, all of which agree upon an input prefix. The main contribution of Venkatesh's work is that it introduces the idea that there are many different feasible semantic definitions of a slice. Another limitation of his work is that his semantics are not trajectory based. His semantics do not allow slicing to be at arbitrary program points and so, in effect, only *end slicing* is considered.

Other theoretical work has attempted to lay the foundations of slicing. However, this previous work has been primarily concerned with static slicing and with intra-technique relationships rather than inter-technique relationships. For example, Reps and Yang [63] show that the PDG is adequate as a representation of program semantics, allowing it to be used in static slicing. Reps [61] shows how interprocedural-slicing can be formulated as a graph reachability problem, once again focusing on static slicing. Cartwright and Felleisen [17] show that the PDG semantics is a lazy semantics, because of the demand driven nature of the representation, while Giacobazzi and Mastroeni [34] present a transfinite semantics to attempt to capture the behaviour of static slicing. Harman et al. show the slicing is lazy in the presence of errors [43]. Weiser [67] observed that his slicing algorithm was not dataflow minimal and speculated on the question of whether dataflow minimal slices were computable. Danicic showed how this problem could be reformulated as a theorem about unfolding [21] while Laurence et al. [56] show how the problem can be expressed in terms of program schematology.

There are several surveys of slicing: Tip [65], and Binkley and Gallagher [11] provide surveys of program slicing techniques and applications. De Lucia [29] presents a shorter, but more up-to-date survey of slicing paradigms. Binkley and Harman [12] present a survey of empirical results on program slicing. These papers provide a broad picture of slicing technology, tools, applications, definitions, and theory. Harman et al. [38,36] introduced the projection theory used in this paper to analyse inter-technique slicing relationships. In this previous work the projection theory was used to explain the difference between syntax-preserving and amorphous slicing, whereas the present paper is concerned solely with syntax–preserving slicing.

7 Conclusion and Future Work

The primary contributions of this paper can be summarised as follows:

(i) We set out a seven point research agenda for a theoretical formalisation of slicing with the aim of stimulating interest in the wider research community in tackling some of these problems.
(ii) We clarify and improve some of the notation and presentation of the results of the relevant previous work in the theoretical formalisation of slicing[6–8].
(iii) We use the formal framework introduced in [6–8] to define both *simultaneous slicing* and *conditioned slicing.*
(iv) We formally define what it means for one form of slicing to be a generalisation of another and prove, using the theory, that both simultaneous slicing and conditioned slicing are generalisations of the standard forms of slicing considered in our previous work [6–8].
(v) We use the theory to compare forms of simultaneous and conditioned slicing both with each other and with the other forms of slicing. We show that each form of simultaneous slicing is weaker than the corresponding non–simultaneous form and that each form of conditioned slicing is weaker than the corresponding static simultaneous form and stronger than the corresponding dynamic simultaneous form.

Future work will consider the relationships among minimal formulations of slice and operations on slicing criteria and will attempt to encompass additional forms of slicing within the theoretical framework established by this paper.

There are non-executable forms of program slice [9,48,66]. Since the semantics of non-executable program fragments is not well defined it is not, at present, clear how such forms fit into our semantically–based framework. This is a subject for future work.

Using program schemas [24], the semantics of slicing has been considered at a level of abstraction more closely related to that at which most slicing algorithms operate. Future work will investigate combining the theory of program schemas [26,54–56] with the work presented here.

Previous work [17,27,34,43,59] has strongly indicated that slicing is more closely related to forms of lazy semantics than to standard semantics. Future work will investigate the incorporation of such lazy semantic forms into the framework.

8 Acknowledgements

Mark Harman is supported, in part, by EPSRC Grants GR/R98938, GR/M58719, GR/M78083 and GR/R43150 and by two development grants from Daimler-Chrysler. Dave Binkley is supported by National Science Foundation grant CCR0305330. Tibor Gyimóthy and Ákos Kiss are supported by The Péter Pázmány Program of the Hungarian National Office of Research and Technology (no. RET-07/2005).

References

[1] H. Agrawal, R. A. DeMillo, E. H. Spafford, Debugging with dynamic slicing and backtracking, Software Practice and Experience 23 (6) (1993) 589–616.

[2] H. Agrawal, J. R. Horgan, Dynamic program slicing, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, 1990, pp. 246–256.

[3] J. Beck, D. Eichmann, Program and interface slicing for reverse engineering, in: IEEE/ACM 15^{th} Conference on Software Engineering (ICSE'93), IEEE Computer Society Press, Los Alamitos, California, USA, 1993, pp. 509–518.

[4] J. M. Bieman, L. M. Ott, Measuring functional cohesion, IEEE Transactions on Software Engineering 20 (8) (1994) 644–657.

[5] D. Binkley, Multi-procedure program integration (August 1991).

[6] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Ákos Kiss, B. Korel, Minimal slicing and the relationships between forms of slicing, in: 5^{th} IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 05), IEEE Computer Society Press, Los Alamitos, California, USA, Budapest, Hungary, 2005, pp. 45–54, best paper award winner.

[7] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Ákos Kiss, B. Korel, Theoretical foundations of dynamic program slicing, Theoretical Computer ScienceAccepted Jan 2006.

[8] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, L. Ouarbya, Formalizing executable dynamic and forward slicing, in: 4^{th} International Workshop on Source Code Analysis and Manipulation (SCAM 04), IEEE

Computer Society Press, Los Alamitos, California, USA, Chicago, Illinois, USA, 2004, pp. 43–52.

[9] D. W. Binkley, Precise executable interprocedural slices, ACM Letters on Programming Languages and Systems 3 (1-4) (1993) 31–45.

[10] D. W. Binkley, The application of program slicing to regression testing, in: M. Harman, K. Gallagher (Eds.), Information and Software Technology Special Issue on Program Slicing, Vol. 40, Elsevier, 1998, pp. 583–594.

[11] D. W. Binkley, K. B. Gallagher, Program slicing, in: M. Zelkowitz (Ed.), Advances in Computing, Volume 43, Academic Press, 1996, pp. 1–50.

[12] D. W. Binkley, M. Harman, A survey of empirical results on program slicing, Advances in Computers 62 (2004) 105–178.

[13] D. W. Binkley, S. Horwitz, T. Reps, Program integration for languages with procedure calls, ACM Transactions on Software Engineering and Methodology 4 (1) (1995) 3–35.

[14] G. Canfora, A. Cimitile, A. De Lucia, Conditioned program slicing, in: M. Harman, K. Gallagher (Eds.), Information and Software Technology Special Issue on Program Slicing, Vol. 40, Elsevier Science B. V., 1998, pp. 595–607.

[15] G. Canfora, A. Cimitile, A. De Lucia, G. A. D. Lucca, Software salvaging based on conditions, in: International Conference on Software Maintenance (ICSM'96), IEEE Computer Society Press, Los Alamitos, California, USA, Victoria, Canada, 1994, pp. 424–433.

[16] G. Canfora, A. Cimitile, M. Munro, RE^2: Reverse engineering and reuse re-engineering, Journal of Software Maintenance : Research and Practice 6 (2) (1994) 53–72.

[17] R. Cartwright, M. Felleisen, The semantics of program dependence, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, 1989, pp. 13–27.

[18] A. Cimitile, A. De Lucia, M. Munro, Identifying reusable functions using specification driven program slicing: a case study, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'95), IEEE Computer Society Press, Los Alamitos, California, USA, Nice, France, 1995, pp. 124–133.

[19] A. Cimitile, A. De Lucia, M. Munro, A specification driven slicing process for identifying reusable functions, Software maintenance: Research and Practice 8 (1996) 145–178.

[20] P. Cousot, R. Cousot, Abstract interpretation frameworks, Journal of Logic and Computation 2 (4) (1992) 511–547.

[21] S. Danicic, Dataflow minimal slicing, PhD thesis, University of North London, UK, School of Informatics (Apr. 1999).

[22] S. Danicic, M. Daoudi, C. Fox, M. Harman, R. M. Hierons, J. Howroyd, L. Ouarbya, M. Ward, Consus: A lightweight program conditioner, Journal of Systems and Software (2005) 241–262Special Issue: Software Reverse Engineering.

[23] S. Danicic, A. De Lucia, M. Harman, Building executable union slices using conditioned slicing, in: 12^{th} International Workshop on Program Comprehension (IWPC 2004), IEEE Computer Society Press, Los Alamitos, California, USA, Bari, Italy, 2004, pp. 89–97.

[24] S. Danicic, C. Fox, M. Harman, R. M. Hierons, J. Howroyd, M. Laurence, Slicing algorithms are minimal for programs which can be expressed as linear, free, liberal schemas, The computer Journal 48 (2005) 737–748.

[25] S. Danicic, M. Harman, A simultaneous slicing theory and derived program slicer, in: 4^{th} RIMS Workshop in Computing, Kyoto University, Kyoto, Japan, 1996.

[26] S. Danicic, M. Harman, R. Hierons, J. Howroyd, M. Laurence, Applications of linear program schematology in dependence analysis, in: $1^{st.}$ International Workshop on Programming Language Interference and Dependence, Verona, Italy, 2004.
URL `http://profs.sci.univr.it/ mastroen/noninterference.html`

[27] S. Danicic, M. Harman, J. Howroyd, L. Ouarbya, A lazy semantics for program slicing, in: $1^{st.}$ International Workshop on Programming Language Interference and Dependence, Verona, Italy, 2004.
URL `http://profs.sci.univr.it/ mastroen/noninterference.html`

[28] S. Danicic, M. Harman, Y. Sivagurunathan, A parallel algorithm for static program slicing, Information Processing Letters 56 (6) (1995) 307–313.

[29] A. De Lucia, Program slicing: Methods and applications, in: 1^{st} IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, Los Alamitos, California, USA, Florence, Italy, 2001, pp. 142–149.

[30] A. De Lucia, A. R. Fasolino, M. Munro, Understanding function behaviours through program slicing, in: 4^{th} IEEE Workshop on Program Comprehension, IEEE Computer Society Press, Los Alamitos, California, USA, Berlin, Germany, 1996, pp. 9–18.

[31] C. Fox, S. Danicic, M. Harman, R. M. Hierons, ConSIT: a fully automated conditioned program slicer, Software—Practice and Experience 34 (2004) 15–46, published online 26th November 2003.

[32] K. B. Gallagher, Evaluating the surgeon's assistant: Results of a pilot study, in: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, California, USA, 1992, pp. 236–244.

[33] K. B. Gallagher, J. R. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering 17 (8) (1991) 751–761.

[34] R. Giacobazzi, I. Mastroeni, Non–standard semantics for program slicing, Higher-Order and Symbolic Computation 16 (4) (2003) 297–339, special issue on Partial Evalution and Semantics-Based Program Manipulation.

[35] R. Gupta, M. J. Harrold, M. L. Soffa, An approach to regression testing using slicing, in: Proceedings of the IEEE Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, California, USA, Orlando, Florida, USA, 1992, pp. 299–308.

[36] M. Harman, D. W. Binkley, S. Danicic, Amorphous program slicing, Journal of Systems and Software 68 (1) (2003) 45–64.

[37] M. Harman, S. Danicic, Using program slicing to simplify testing, Software Testing, Verification and Reliability 5 (3) (1995) 143–162.

[38] M. Harman, S. Danicic, Amorphous program slicing, in: 5^{th} IEEE International Workshop on Program Comprenhesion (IWPC'97), IEEE Computer Society Press, Los Alamitos, California, USA, Dearborn, Michigan, USA, 1997, pp. 70–79.

[39] M. Harman, R. M. Hierons, An overview of program slicing, Software Focus 2 (3) (2001) 85–92.

[40] M. Harman, R. M. Hierons, S. Danicic, J. Howroyd, C. Fox, Pre/post conditioned slicing, in: IEEE International Conference on Software Maintenance (ICSM'01), IEEE Computer Society Press, Los Alamitos, California, USA, Florence, Italy, 2001, pp. 138–147.

[41] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper, Testability transformation, IEEE Transactions on Software Engineering 30 (1) (2004) 3–16.

[42] M. Harman, L. Hu, M. Munro, X. Zhang, D. W. Binkley, S. Danicic, M. Daoudi, L. Ouarbya, Syntax-directed amorphous slicing, Journal of Automated Software Engineering 11 (1) (2004) 27–61.

[43] M. Harman, D. Simpson, S. Danicic, Slicing programs in the presence of errors, Formal Aspects of Computing 8 (4) (1996) 490–497.

[44] R. M. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Software Testing, Verification and Reliability 9 (4) (1999) 233–262.

[45] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, M. Daoudi, Conditioned slicing supports partition testing, Software Testing, Verification and Reliability 12 (2002) 23–28.

[46] S. Horwitz, P. Pfeiffer, T. Reps, Dependence analysis for pointer variables, in: Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation, ACM SIGPLan Notices, Portland, OR, USA, 1989, pp. 28–40.

[47] S. Horwitz, J. Prins, T. Reps, Integrating non–interfering versions of programs, ACM Transactions on Programming Languages and Systems 11 (3) (1989) 345–387.

[48] S. Horwitz, T. Reps, D. W. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12 (1) (1990) 26–61.

[49] M. Kamkar, Interprocedural dynamic slicing with applications to debugging and testing, PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, available as Linköping Studies in Science and Technology, Dissertations, Number 297 (1993).

[50] D. E. Knuth, R. W. Floyd, Notes on avoiding "go to" statements, Information Processing Letters 1 (1) (1971) 23–31.

[51] B. Korel, J. Laski, Dynamic program slicing, Information Processing Letters 29 (3) (1988) 155–163.

[52] B. Korel, J. Rilling, Dynamic program slicing in understanding of program execution, in: 5^{th} IEEE International Workshop on Program Comprenhesion (IWPC'97), IEEE Computer Society Press, Los Alamitos, California, USA, Dearborn, Michigan, USA, 1997, pp. 80–89.

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

[54] M. R. Laurence, Equivalence of linear, free, liberal program schemas is decidable in polynomial time, Ph.D. thesis, Goldsmiths College, University of London (July 2004).

[55] M. R. Laurence, Deciding equivalence of liberal, free, function-linear schemas to their slices, submitted to Journal of Logic and Algebraic Programming.

[56] M. R. Laurence, S. Danicic, M. Harman, R. Hierons, J. Howroyd, Equivalence of conservative, free, linear program schemas is decidable, Theoretical Computer Science 290 (2003) 831–862.

[57] J. R. Lyle, M. Weiser, Automatic program bug location by program slicing, in: 2^{nd} International Conference on Computers and Applications, IEEE Computer Society Press, Los Alamitos, California, USA, Peking, 1987, pp. 877–882.

[58] L. M. Ott, J. J. Thuss, Slice based metrics for estimating cohesion, in: Proceedings of the IEEE-CS International Metrics Symposium, IEEE Computer Society Press, Los Alamitos, California, USA, Baltimore, Maryland, USA, 1993, pp. 71–81.

[59] L. Ouarbya, A lazy semantics for program slicing, PhD Thesis, Department of Computing, Goldsmiths College, University of London (2005).

[60] L. Ramshaw, Eliminating goto's while preserving program structure, Journal of the ACM 35 (4) (1988) 893–920.

[61] T. Reps, Program analysis via graph reachability, in: M. Harman, K. Gallagher (Eds.), Information and Software Technology Special Issue on Program Slicing, Vol. 40, Elsevier Science B. V., 1998, pp. 701–726.

[62] T. Reps, S. Horwitz, M. Sagiv, G. Rosay, Speeding up slicing, in: ACM Foundations of Software Engineering (FSE'94), ACM SIGSOFT Software Engineering Notes 19, 5 (December 1994), New Orleans, LA, 1994, pp. 11–20.

[63] T. Reps, W. Yang, The semantics of program slicing, Tech. Rep. Technical Report 777, University of Wisconsin (1988).

[64] D. Simpson, S. H. Valentine, R. Mitchell, L. Liu, R. Ellis, Recoup – Maintaining Fortran, ACM Fortran forum 12 (3) (1993) 26–32.

[65] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (3) (1995) 121–189.

[66] G. A. Venkatesh, The semantic approach to program slicing, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Canada, 1991, pp. 26–28, proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[67] M. Weiser, Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method, PhD thesis, University of Michigan, Ann Arbor, MI (1979).

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

[69] M. Weiser, J. R. Lyle, Experiments on slicing–based debugging aids, Empirical studies of programmers, Soloway and Iyengar (eds.), Molex, 1985, Ch. 12, pp. 187–197.