Math. Struct. in Comp. Science (2011), *vol.* 21, *pp.* 1339–1362. © Cambridge University Press 2011
doi:10.1017/S0960129511000223

On the computational complexity of dynamic slicing problems for program schemas

SEBASTIAN DANICIC[†][§], ROBERT. M. HIERONS[‡] and
MICHAEL R. LAURENCE[†]

[†]*Department of Computing, Goldsmiths College, University of London, London SE14 6NW, United Kingdom*
Email: s.danicic@gold.ac.uk
[‡]*Department of Information Systems and Computing, Brunel University, Middlesex, UB8 3PH, United Kingdom*
Email: rob.hierons@brunel.ac.uk

Received 2 June 2008; revised 23 March 2010

Given a program, a quotient can be obtained from it by deleting zero or more statements. The field of program slicing is concerned with computing a quotient of a program that preserves part of the behaviour of the original program. All program slicing algorithms take account of the structural properties of a program, such as control dependence and data dependence, rather than the semantics of its functions and predicates, and thus work, in effect, with program schemas. The dynamic slicing criterion of Korel and Laski requires only that program behaviour is preserved in cases where the original program follows a particular path, and that the slice/quotient follows this path. In this paper we formalise Korel and Laski's definition of a dynamic slice as applied to linear schemas, and also formulate a less restrictive definition in which the path through the original program need not be preserved by the slice. The less restrictive definition has the benefit of leading to smaller slices. For both definitions, we compute complexity bounds for the problems of establishing whether a given slice of a linear schema is a dynamic slice and whether a linear schema has a non-trivial dynamic slice, and prove that the latter problem is NP-hard in both cases. We also give an example to prove that minimal dynamic slices (whether or not they preserve the original path) need not be unique.

1. Introduction

A schema represents the statement structure of a program by replacing real functions and predicates by symbols representing them. A schema S thus defines a whole class of programs which all have the same structure. A schema is *linear* if it does not contain more than one occurrence of the same function or predicate symbol. As an example, Figure 1 gives a schema S, and Figure 2 shows one of the programs obtainable from the schema of Figure 1 by interpreting its function and predicate symbols.

[§] This work was partly supported by the Engineering and Physical Sciences Research Council, UK, under grant EP/E002919/1.

$$\begin{array}{lll} u := h(); & & \\ \textit{if } p(w) & \textit{then} & v := f(u); \\ & \textit{else} & v := g(); \end{array}$$

Fig. 1. A schema

$$\begin{array}{lll} u := 1; & & \\ \textit{if } w > 1 & \textit{then} & v := u + 1; \\ & \textit{else} & v := 2; \end{array}$$

Fig. 2. A program defined from the schema of Figure 1

The subject of schema theory is connected with that of program transformation, and was originally motivated by a wish to compile programs effectively (Greibach 1975). Thus an important problem in schema theory is that of establishing whether two schemas are equivalent: that is, whether they always have the same termination behaviour and give the same final value for every variable, given any initial state and any interpretation of function and predicate symbols. The history of this problem is discussed further in Section 1.2.

Schema theory is also relevant to program slicing, and this is the motivation for the main results of this paper. We define a *quotient* of a schema S to be any schema obtained by deleting zero or more statements from S. A quotient of S is non-trivial if it is distinct from S. Thus, a quotient of a schema is not required to satisfy any semantic condition; it is defined purely syntactically. The field of program slicing is concerned with computing a quotient of a program that preserves part of the behaviour of the original program. Program slicing is used in program comprehension (De Lucia *et al.* 1996; Harman *et al.* 2001), software maintenance (Canfora *et al.* 1994; Cimitile *et al.* 1996; Gallagher 1992; Gallagher and Lyle 1991) and debugging (Agrawal *et al.* 1993; Kamkar 1993; Lyle and Weiser 1987; Weiser and Lyle 1985).

All program slicing algorithms take account of the structural properties of a program, such as control dependence and data dependence, rather than the semantics of its functions and predicates, and thus work, in effect, with linear program schemas. There are two main forms of program slicing: static and dynamic.

— In static program slicing, only the program itself is used to construct a slice. Most static slicing algorithms are based on Weiser's algorithm (Weiser 1984), which uses the data and control dependence relations of the program to compute the set of statements that the slice retains. An end-slice of a program with respect to a variable v is a slice that always returns the same final value for v as the original program when executed from the same input. It has been proved that Weiser's algorithm gives minimal static end-slices (Danicic *et al.* 2005) for linear, free, liberal program schemas. This result has recently been strengthened by allowing function-linear schemas, in which only predicate symbols are required to be non-repeating (Laurence 2005).

— In dynamic program slicing, a path through the program is also used as input. Dynamic slices of programs may be smaller than static slices since they are only required to preserve behaviour in cases where the original program follows a particular path. As originally formulated by Korel and Laski (Korel and Laski 1988), a dynamic slice of a program P is defined by three parameters besides P, namely, a variable set V, an initial input state d and an integer n. The slice with respect to these parameters is required to follow the same path as P up to the nth statement (with statements not lying in the slice deleted from the path through the slice) and give the same value for each element of V as P after the nth statement after execution from the initial state d. Many dynamic slicing algorithms have been written (Agrawal and Horgan 1990; Beszédes *et al.* 2001; Gopal 1991; Kamkar *et al.* 1992; Kamkar 1998; Korel and Laski 1988; Korel 1995; Korel and Rilling 1998), most of which compute a slice using the data and control dependence relations along the given path through the original program. This produces a correct slice, and uses polynomial time, but need not give a minimal or even non-trivial slice, even where one exists.

Our definition of a path-faithful dynamic slice (PFDS) for a linear schema S has two parameters besides S, namely, a path through S and a variable set, but not an initial state. This definition is analogous to that of Korel and Laski since the initial state included in their parameter set is used solely to compute a path through the program in linear schema-based slicing algorithms. We prove, in effect, that the problem of determining whether a particular quotient of a program is a dynamic slice in the sense of Korel and Laskiis is decidable in polynomial time, and that the problem of establishing whether a program has a non-trivial path-faithful dynamic slice is intractable, unless P=NP. This shows that there can be no tractable dynamic slicing algorithm that produces correct slices and always gives a non-trivial slice of a program where one exists.

The requirement of Korel and Laski that the path through the slice be path-faithful seems unnecessarily strong, so we define a more general dynamic slice (DS), in which the sequence of functions and predicates through which the path through the slice passes is a subsequence of that for the path through the original schema, but the path through the slice must still pass the same number of times through the program point at the end of the original path. For this less restrictive definition, we prove that the problem of determining whether a particular slice of a program is a dynamic slice is decidable in Co-NP time, and the problem of establishing whether a program has a non-trivial dynamic slice is NP-hard.

We also give an example to prove that unique minimal dynamic slices (whether or not path-faithful) of a linear schema S do not always exist.

The results of this paper have several practical ramifications. First, we prove that the problem of determining whether a linear schema has a non-trivial dynamic slice is computationally hard, and it is clear that this result must also hold for programs. In addition, since this decision problem is computationally hard, the problem of producing minimal dynamic slices must also be computationally hard. Second, we define a new notion of a dynamic slice that places strictly weaker constraints on the slice than those traditionally used, and which can thus lead to smaller dynamic slices. In Section 4, we explain why these (smaller) dynamic slices can be appropriate, motivating this through a problem in program testing. Naturally, this weaker notion of a dynamic slice is also directly applicable to programs. Finally, we prove that minimal dynamic slices need not be unique. This has consequences when designing dynamic slicing algorithms since it tells us that algorithms that identify and then delete one statement at a time can lead to suboptimal dynamic slices.

It should be noted that much theoretical work on program slicing and program analysis, including that of Müller-Olm's study of dependence analysis of parallel programs (Müller-Olm 2004), and on deciding the validity of relations between variables at given program points (Müller-Olm and Seidl 2004a; Müller-Olm and Seidl 2004b) only considers programs in which branching is treated as non-deterministic, and is thus more 'approximate' than our own in this respect in that we take into account control dependence as part of the program structure.

1.1. *Different classes of schemas*

Many subclasses of schemas have been defined:

Structured schemas: *Goto* commands are forbidden in structured schemas, so loops must be constructed using while statements. *All schemas considered in this paper are structured.*

Linear schemas: There is at most one occurrence of each function and predicate symbol in linear schemas.

Free schemas: All paths in free schemas are executable under some interpretation.

Conservative schemas: Every assignment in a conservative schema is of the form $v := f(v_1, \ldots, v_r)$, where $v \in \{v_1, \ldots, v_r\}$.

Liberal schemas: In liberal schemas, two assignments along any executable path can always be made to assign distinct values to their respective variables by a suitable choice of domain.

It is easy to show that all conservative schemas are liberal.

Paterson (1967) gave a proof that the problem of determining whether a schema is both liberal and free is decidable; and since he also gave an algorithm transforming a schema S into a schema T such that T is both liberal and free if and only if S is liberal, determining whether a schema is liberal is clearly decidable. It is an open problem whether freeness is decidable for the class of linear schemas. However, Paterson also proved, using a reduction from the Post Correspondence Problem, that determining whether a schema is free is not decidable.

1.2. *Previous results on the decidability of schema equivalence*

Most previous research on schemas has focused on schema equivalence. All results on the decidability of showing the equivalence of schemas are either negative or confined to very restricted classes of schemas. In particular, Paterson proved, using a reduction from the halting problem for Turing machines, that equivalence is undecidable for the class of all schemas containing at least two variables (Luckham *et al.* 1970). Ashcroft and Manna (1975) showed that an arbitrary schema, which may include *goto* commands, can be effectively transformed into an equivalent structured schema provided statements such as *while* $\neg p(\mathbf{u})$ *do* T are permitted; hence Paterson's result shows that any class of schemas for which equivalence can be decided cannot contain this class of schemas. Thus, in order to get positive results for this problem, we must clearly define the relevant classes of schema with great care.

Positive results for the decidability of the equivalence of schemas include the following. In an early result in schema theory, Ianov (1960) introduced a restricted class of schemas, the Ianov schemas, for which equivalence is decidable. This problem was later shown to be NP-complete (Rutledge 1964; Hunt *et al.* 1980). Ianov schemas are characterised by being monadic (that is, they contain only a *single* variable) and having only unary function symbols, so Ianov schemas are conservative.

Paterson (1967) proved that equivalence is decidable for a class of schemas called *progressive schemas*, in which every assignment references the variable assigned by the previous assignment along every legal path.

Sabelfeld (1990) proved that equivalence is decidable for the class of schemas called *through schemas*. A through schema satisfies two conditions: first, that on every path from an accessible predicate p to a predicate q that does not pass through another predicate, and every variable x referenced by p, there is a variable referenced by q that defines a term containing the term defined by x; and second, distinct variables referenced by a predicate can be made to define distinct terms under some interpretation.

It has been proved that equivalence is decidable for the class of schemas that are linear, free and conservative (Laurence *et al.* 2003). More recently, the same conclusion was proved to hold under the weaker hypothesis of liberality rather than conservatism (Laurence *et al.* 2004; Danicic *et al.* 2007).

1.3. *Organisation of the paper*

In Section 2 we give basic definitions of schemas. In Section 3 we define path-faithful dynamic slices, and in Section 4, we define general dynamic slices. In Section 5, we give an example to prove that unique minimal dynamic slices need not exist. In Section 6, we prove complexity bounds for problems concerning the existence of dynamic slices. Finally, in Section 7, we discuss some further directions for research in this area.

2. Basic definitions for schemas

Throughout this paper, we use $\mathscr{F}$, $\mathscr{P}$, $\mathscr{V}$ and $\mathscr{L}$ to denote fixed infinite sets of *function symbols*, *predicate symbols*, *variables* and *labels*, respectively. A *symbol* means an element of $\mathscr{F} \cup \mathscr{P}$ in this paper. For example, the schema in Figure 1 has function set $\mathscr{F} = \{f, g, h\}$, predicate set $\mathscr{P} = \{p\}$ and variable set $\mathscr{V} = \{u, v\}$. We assume a function

$$arity : \mathscr{F} \cup \mathscr{P} \to \mathbb{N}.$$

The arity of a symbol x is the number of arguments referenced by x: for example, in the schema in Figure 1, the function f has arity one, the function g has arity zero and p has arity one.

Note that when the arity of a function symbol g is zero, g may be thought of as a constant.

The set $Term(\mathscr{F}, \mathscr{V})$ of *terms* is defined as follows:

— Each variable is a term.

— if $f \in \mathscr{F}$ is of arity n and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

For example, in the schema in Figure 1, the variable u takes the value (term) $h()$; after the first assignment is executed and if we take the true branch, the variable v ends with the value (term) $f(h())$.

We refer to a tuple $\mathbf{t} = (t_1, \ldots, t_n)$, where each t_i is a term, as a vector term. We say $p(\mathbf{t})$ is a predicate term if $p \in \mathscr{P}$ and the number of components of the vector term $\mathbf{t}$ is $arity(p)$.

Schemas are defined recursively as follows:

— *skip* is a schema.

— Any label is a schema.

— An assignment '$y := f(\mathbf{x});$' for a variable y, a function symbol f and an n-tuple $\mathbf{x}$ of variables, where n is the arity of f, is a schema.

— If S_1 and S_2 are schemas, $S_1 S_2$ is a schema.

— If S_1 and S_2 are schemas, p is a predicate symbol and $\mathbf{y}$ is an m-tuple of variables, where m is the arity of p, then $\mathit{if}\ p(\mathbf{y})\ \mathit{then}\ S_1\ \mathit{else}\ S_2$ is a schema.

— If T is a schema, q is a predicate symbol and $\mathbf{z}$ is an m-tuple of variables, where m is the arity of q, the schema $\mathit{while}\ q(\mathbf{z})\ T$ is a schema.

If no function, predicate symbol or label occurs more than once in a schema S, we say that S is linear. If a schema does not contain any predicate symbols, we say it is predicate-free. If a linear schema S contains a subschema $\mathit{if}\ p(\mathbf{y})\ \mathit{then}\ S_1\ \mathit{else}\ S_2$, we refer to S_1 and S_2 as the T-part and F-part, respectively, of p in S. For example, in the schema in Figure 1, the predicate p has T-part '$v := f(u);$' and F-part '$v := g();$'. If a linear schema S contains a subschema $\mathit{while}\ q(\mathbf{z})\ T$, we refer to T as the body of q in S.

Quotients of schemas are defined recursively as follows:

— *skip* is a quotient of every schema.

— If S' is a quotient of S, then $S'T$ is a quotient of ST and TS' is a quotient of TS.

— If T' is a quotient of T, then $\mathit{while}\ q(\mathbf{y})\ T'$ is a quotient of $\mathit{while}\ q(\mathbf{y})\ T$.

— If T_1 and T_2 are quotients of schemas S_1 and S_2, respectively, $\mathit{if}\ p(\mathbf{x})\ \mathit{then}\ T_1\ \mathit{else}\ T_2$ is a quotient of $\mathit{if}\ p(\mathbf{x})\ \mathit{then}\ S_1\ \mathit{else}\ S_2$.

A quotient T of a schema S is said to be non-trivial if $T \neq S$.

We can obtain a quotient in the schema in Figure 1 by replacing the first statement by *skip* or the if statement by *skip*. It is also possible to replace either or both parts of the if statement by *skip* or any combination of these steps.

2.1. *Paths through a schema*

We will express the semantics of schemas using paths through them, so the definition of a path through a schema has to include the variables assigned or referenced by successive function or predicate symbols.

The set of prefixes of a *word* (that is, a sequence) σ over an alphabet is denoted by $pre(\sigma)$. For example, if $\sigma = x_1x_3x_2$ over the alphabet $\{x_1, x_2, x_3\}$, then the set $pre(\sigma)$ consists of the words $x_1x_2x_3, x_1x_2, x_1$ and the empty word. More generally, if Ω is a set of words, we define $pre(\Omega) = \{pre(\sigma) | \ \sigma \in \Omega\}$.

For each schema S there is an associated alphabet $alphabet(S)$ consisting of all elements of $\mathscr{L}$ and the set of letters of the form $\underline{y := f(\mathbf{x})}$ for assignments $y := f(\mathbf{x});$ in S and $\underline{p(\mathbf{y}), Z}$ for $Z \in \{\mathsf{T}, \mathsf{F}\}$, where *if* $p(\mathbf{y})$ or *while* $p(\mathbf{y})$ occurs in S. For example, the schema in Figure 1 has no labels and alphabet

$$\{\underline{y := h()}, \underline{v := f(\mathbf{u})}, \underline{v := g()}, \underline{p(\mathbf{w}), \mathsf{T}}, \underline{p(\mathbf{w}), \mathsf{F}}\}.$$

The set $\Pi(S)$ of terminating paths through S, is defined recursively as follows:

— $\Pi(l) = l$, for any $l \in \mathscr{L}$.
— $\Pi(skip)$ is the empty word.
— $\Pi(y := f(\mathbf{x});) = \underline{y := f(\mathbf{x})}$.
— $\Pi(S_1S_2) = \Pi(S_1)\,\Pi(S_2)$.
— $\Pi(\textit{if } p(\mathbf{x}) \textit{ then } S_1 \textit{ else } S_2) = \underline{p(\mathbf{x}), \mathsf{T}}\,\Pi(S_1) \cup \underline{p(\mathbf{x}), \mathsf{F}}\,\Pi(S_2)$.
— $\Pi(\textit{while}\,(q(\mathbf{y})\ T)) = (\underline{q(\mathbf{y}), \mathsf{T}}\,\Pi(T))^*\ \underline{q(\mathbf{y}), \mathsf{F}}$.

We will sometimes abbreviate $\underline{q(\mathbf{y}), Z}$ to $\underline{q, Z}$ and $\underline{y := f(\mathbf{x})}$ to $\underline{f}$.

We define $\Pi^{\omega}(S)$ to be the set containing $\Pi(S)$, plus all infinite words whose finite prefixes are prefixes of terminating paths. A *path* through S is any (not necessarily strict) prefix of an element of $\Pi^{\omega}(S)$. As an example, if S is the schema in Figure 1, which has no loops, then $\Pi(S) = \Pi^{\omega}(S)$. In fact, $\Pi(S)$ in this case contains exactly two paths, which are defined by $p(\mathbf{w})$ taking the true or false branches, and every path through S is a prefix of one of these paths.

If S' is a quotient of a schema S and $\rho \in pre(\Pi(S))$ (that is, ρ is a path through S), then $proj_{S'}(\rho)$ is the path obtained from ρ by deleting all letters having function or predicate symbols not lying in S' and all labels not occurring in S'. It is easy to prove that $proj_{S'}(\Pi(S)) = \Pi(S')$ in this case.

2.2. *Semantics of schemas*

The symbols used to build schemas are given meaning by defining the notions of a state and an interpretation. It will be assumed that 'values' are given in a single set D, which will be called the *domain*. We are mainly interested in the case in which $D = Term(\mathscr{F}, \mathscr{V})$ (the Herbrand domain) and the function symbols represent the 'natural' functions with respect to $Term(\mathscr{F}, \mathscr{V})$.

Definition 1 (states, (Herbrand) interpretations and the natural state e). Given a domain D, a *state* is either $\perp$ (denoting non-termination) or a function $\mathscr{V} \to D$. The set of all such states will be denoted by State$(\mathscr{V}, D)$. An interpretation i defines, for each function symbol $f \in \mathscr{F}$ of arity n, a function $f^i : D^n \to D$, and for each predicate symbol $p \in \mathscr{P}$ of arity m, a function $p^i : D^m \to \{\mathsf{T}, \mathsf{F}\}$. The set of all interpretations with domain D will be denoted by $Int(\mathscr{F}, \mathscr{P}, D)$.

We call the set $Term(\mathscr{F}, \mathscr{V})$ of terms the *Herbrand domain* and say that a function from $\mathscr{V}$ to $Term(\mathscr{F}, \mathscr{V})$ is a Herbrand state. An interpretation i for the Herbrand domain is said to be *Herbrand* if the functions $f^i : Term(\mathscr{F}, \mathscr{V})^n \to Term(\mathscr{F}, \mathscr{V})$ for each $f \in \mathscr{F}$ are defined as

$$f^i(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$$

for all n-tuples of terms $(t_1, \ldots, t_n)$.

We define the *natural state* $e : \mathscr{V} \to Term(\mathscr{F}, \mathscr{V})$ by $e(v) = v$ for all $v \in \mathscr{V}$.

In the schema in Figure 1, the natural state simply maps variable $\mathbf{u}$ to the name u, variable $\mathbf{v}$ to the name v and variable $\mathbf{w}$ to the name w. The program in Figure 2 can be produced from this schema through the interpretation that maps '$h()$;' to 1, $p(\mathbf{w})$ to $\mathbf{w} > 1$, $f(\mathbf{u})$ to $\mathbf{u} + 1$, and $g()$ to 2, and it is clear that this is not a Herbrand interpretation.

Observe that if an interpretation i is Herbrand, this does not restrict the mappings $p^i : (Term(\mathscr{F}, \mathscr{V}))^m \to \{\mathsf{T}, \mathsf{F}\}$ defined by i for each $p \in \mathscr{P}$.

It is well known (Manna 1974, Section 4-14) that Herbrand interpretations are the only ones that need to be considered for many schema properties. This fact is stated more precisely in Theorem 8. In particular, our semantic slicing definitions may be defined in terms of Herbrand domains.

Given a schema S and a domain D, an initial state $d \in \mathrm{State}(\mathscr{V}, D)$ with $d \neq \bot$ and an interpretation $i \in Int(\mathscr{F}, \mathscr{P}, D)$, we now define the final state $\mathscr{M}[\![S]\!]^i_d \in \mathrm{State}(\mathscr{V}, D)$ and the associated path $\pi_S(i, d) \in \Pi^\omega(S)$. In order to do this, we need to define the predicate-free schema associated with the prefix of a path by considering the sequence of assignments through which it passes.

Definition 2 (the schema $schema(\sigma)$**).** Given a word $\sigma \in (alphabet(S))^*$ for a schema S, we recursively define the predicate-free schema $schema(\sigma)$ by the following rules:

$$\begin{aligned}
schema(skip) &= skip \\
schema(l) &= l \text{ for } l \in \mathscr{L} \\
schema(\sigma \underline{v := f(\mathbf{x})}) &= schema(\sigma)\, v := f(\mathbf{x}); \\
schema(\sigma \underline{p(\mathbf{x}), X}) &= schema(\sigma).
\end{aligned}$$

Consider, for example, the path of the schema in Figure 1 that passes through the true branch of p. This defines a word

$$\sigma = \underline{u := h() p(\mathbf{w}), \mathsf{T} v := f(\mathbf{u})}$$

and

$$schema(\sigma) = \underline{u := h() v := f(\mathbf{u})}.$$

Lemma 3. Let S be a schema. If $\sigma \in pre(\Pi(S))$, the set $\{m \in alphabet(S) | \, \sigma m \in pre(\Pi(S))\}$ is one of the following:

— a label

— a singleton containing an assignment letter $\underline{y := f(\mathbf{x})}$
— a pair $\{\underline{p(\mathbf{x}), \mathsf{T}}, \underline{p(\mathbf{x}), \mathsf{F}}\}$ for a predicate p of S
— or the empty set.

And the last case holds if $\sigma \in \Pi(S)$.

Proof. See Laurence (2005, Lemma 6). □

Lemma 3 reflects the fact that at any point in the execution of a program there is never more than one 'next step' that may be taken, and an element of $\Pi(S)$ cannot be a strict prefix of another.

Definition 4 (semantics of predicate-free schemas). Given a state $d \neq \perp$, the final state $\mathscr{M}[\![S]\!]^i_d$ and associated path $\pi_S(i,d) \in \Pi^\omega(S)$ of a schema S are defined as follows:

— $\mathscr{M}[\![skip]\!]^i_d = d$ and $\pi_{skip}(i,d)$ is the empty word.

— $\mathscr{M}[\![l]\!]^i_d = d$ and $\pi_l(i,d) = l$ for $l \in \mathscr{L}$.

— $\mathscr{M}[\![y := f(\mathbf{x});]\!]^i_d(v) = \begin{cases} d(v) & \text{if } v \neq y \\ f^i(d(\mathbf{x})) & \text{if } v = y \end{cases}$
where the vector term $d(\mathbf{x}) = (d(x_1), \ldots, d(x_n))$ for $\mathbf{x} = (x_1, \ldots, x_n))$, and

$$\pi_{y := f(\mathbf{x});}(i,d) = \underline{y := f(\mathbf{x})}.$$

— For sequences S_1S_2 of predicate-free schemas,

$$\mathscr{M}[\![S_1S_2]\!]^i_d = \mathscr{M}[\![S_2]\!]^i_{\mathscr{M}[\![S_1]\!]^i_d}$$

and

$$\pi_{S_1S_2}(i,d) = \pi_{S_1}(i,d)\pi_{S_2}\left(i, \mathscr{M}[\![S_1]\!]^i_d\right).$$

This uniquely defines $\mathscr{M}[\![S]\!]^i_d$ and $\pi_S(i,d)$ if S is predicate-free. In order to give the semantics of a general schema S, we first define the path $\pi_S(i,d)$ of S with respect to the interpretation i and initial state d.

Definition 5 (the path $\pi_S(i,d)$). Given a schema S, an interpretation i and a state $d \neq \perp$, the path $\pi_S(i,d) \in \Pi^\omega(S)$ is defined by the condition that for all

$$\sigma\, \underline{p(\mathbf{x}), Z} \in pre(\pi_S(i,d)),$$

the equality

$$p^i(\mathscr{M}[\![schema(\sigma)]\!]^i_d(\mathbf{x})) = Z$$

holds.

In other words, the path $\pi_S(i,d)$ has the property that if a predicate expression $p(\mathbf{x})$ along $\pi_S(i,d)$ is evaluated with respect to the predicate-free schema consisting of the sequence of assignments preceding that predicate in $\pi_S(i,d)$, then the value of the resulting predicate term given by i 'agrees' with the value given in $\pi_S(i,d)$. Consider, for example, the schema given in Figure 1 and the interpretation that gives the program in Figure 2. Given a state d in which w has a value greater than one, we obtain the path

$$\underline{u := h()}\ \underline{p(\mathbf{w}), \mathsf{T}}\ \underline{v := f(\mathbf{u})}.$$

By Lemma 3, this defines the path $\pi_S(i,d) \in \Pi^\omega(S)$ uniquely.

Definition 6 (the semantics of arbitrary schemas). If $\pi_S(i,d)$ is finite, we define

$$\mathscr{M}[\![S]\!]^i_d = \mathscr{M}[\![schema(\pi_S(i,d))]\!]^i_d$$

(which is already defined since $schema(\pi_S(i,d))$ is predicate-free), otherwise $\pi_S(i,d)$ is infinite and we define $\mathscr{M}[\![S]\!]^i_d = \perp$. In this last case, we may say that $\mathscr{M}[\![S]\!]^i_d$ is not terminating.

For convenience, if S is predicate-free and $d : \mathscr{V} \to Term(\mathscr{F}, \mathscr{V})$ is a state, we define $\mathscr{M}[\![S]\!]_d = \mathscr{M}[\![S]\!]^i_d$ unambiguously – that is, we assume that the interpretation i is Herbrand if d is a Herbrand state. Also, if ρ is a path through a schema, we may write $\mathscr{M}[\![\rho]\!]_e$ to mean $\mathscr{M}[\![schema(\rho)]\!]_e$.

Observe that

$$\mathscr{M}[\![S_1 S_2]\!]^i_d = \mathscr{M}[\![S_2]\!]^i_{\mathscr{M}[\![S_1]\!]^i_d}$$

and

$$\pi_{S_1 S_2}(i,d) = \pi_{S_1}(i,d)\pi_{S_2}(i, \mathscr{M}[\![S_1]\!]^i_d)$$

hold for all schemas (not just predicate-free ones).

Given a schema S and $\mu \in pre(\Pi(S))$, we say that μ passes through a predicate term $p(\mathbf{t})$ if μ has a prefix μ' ending in $\underline{p(\mathbf{x}), Y}$ for $y \in \{\mathsf{T}, \mathsf{F}\}$ such that $\mathscr{M}[\![schema(\mu')]\!]_e(\mathbf{x}) = \mathbf{t}$ holds. In this case we say that $p(\mathbf{t}) = Y$ is a *consequence* of μ. For example, the path

$$\underline{u := h()}\ \underline{p(\mathbf{w}), \mathsf{T}}\ \underline{v := f(\mathbf{u})}$$

of the schema in Figure 1 passes through the predicate term $p(\mathbf{w})$ since this path has no assignments to w before p.

Definition 7 (path compatibility and executability). Let ρ be a path through a schema S. Then ρ is *executable* if ρ is a prefix of $\pi_S(j,d)$ for some interpretation j and state d. Two paths ρ, ρ' through schemas S, S' are *compatible* if, for some interpretation j and state d, they are prefixes of $\pi_S(j,d)$ and $\pi_{S'}(j,d)$, respectively.

The justification for restricting ourselves to consideration of Herbrand interpretations and the state e as the initial state lies in the fact that Herbrand interpretations are the 'most general' of interpretations. Theorem 8, which is virtually a restatement of Manna (1974, Theorem 4-1), expresses this formally.

Theorem 8. Let χ be a set of schemas, D be a domain, d be a function from the set of variables into D and i be an interpretation using this domain. Then there is a Herbrand interpretation j such that the following hold:

(1) For all $S \in \chi$, the path $\pi_S(j,e) = \pi_S(i,d)$.

(2) If $S_1, S_2 \in \chi$ and v_1, v_2 are variables and $\rho_k \in pre(\pi_{S_k}(j,e))$ for $k = 1,2$ and $\mathscr{M}[\![\rho_1]\!]_e(v_1) = \mathscr{M}[\![\rho_2]\!]_e(v_2)$, then $\mathscr{M}[\![\rho_1]\!]^i_d(v_1) = \mathscr{M}[\![\rho_2]\!]^i_d(v_2)$ also holds.

As a consequence of Theorem 8 (1), we may assume in Definition 7 that $d = e$ and the interpretation j is Herbrand without strengthening the Definition. In the rest of this paper we will assume that all interpretations are Herbrand.

3. The path-faithful dynamic slicing criterion

In this section we adapt the notion of a dynamic program slice to program schemas. Dynamic program slicing is formalised in Korel and Laski's original paper (Korel and Laski 1988). Their definition uses two functions, $Front$ and DEL, in which $Front(T,i)$ denotes the first i elements of a trajectory† T and $DEL(T,\pi)$ denotes the trajectory T with all elements that satisfy predicate π removed. A trajectory is a path through a program, where each node is represented by a line number, so for path ρ we have that $\hat{\rho}$ is the corresponding trajectory.

Korel and Laski use a slicing criterion that is a tuple $c = (x, I^q, V)$ in which x is the program input being considered, I^q denotes the execution of statement I as the qth statement in the path taken when p is executed with input x, and V is the set of variables of interest.

The definition is then as follows‡.

Definition 9. Let $c = (x, I^q, V)$ be a slicing criterion of a program p and T be the trajectory of p on input x. A dynamic slice of p on c is any executable program p' that is obtained from p by deleting zero or more statements such that, when executed on input x, it produces a trajectory T' for which there exists an execution position q' such that:

KL1: $Front(T', q') = DEL(Front(T,q), T(i) \notin N' \wedge 1 \leqslant i \leqslant q)$, where N' is a set of instructions in p'.

KL2: For all $v \in V$, the value of v before the execution of instruction $T(q)$ in T equals the value of v before the execution of instruction $T'(q')$ in T'.

KL3: $T'(q') = T(q) = I$.

In producing a dynamic slice, all we are allowed to do is eliminate statements. We have the requirement that the slice and the original program produce the same value for each variable in the chosen set V at the specified execution position and that the path in p' up to q' followed by using input x is equivalent to that formed by removing from the path T all elements not in the slice. Interestingly, it has been observed that this additional constraint, that

$$Front(T', q') = DEL(Front(T,q), T(i) \notin N' \wedge 1 \leqslant i \leqslant q),$$

means that a static slice is not necessarily a valid dynamic slice (Binkley *et al.* 2006).

† A trajectory is a path in which we do not distinguish between true and false values for a predicate. There is a one-to-one correspondence between paths and trajectories unless there is an if statement that contains only *skip*.

‡ Note that this is an almost exact quote from Korel and Laski (1988) and is taken from Binkley *et al.* (2006)

$$
\begin{array}{ll}
\textit{while}\, p(w) & \{ \\
 & w := g(w); \\
 & v := f(u); \\
 & \textit{if}\; q(w,t)\; \textit{then} \quad u := h(u); \\
 & t := H(t); \\
 & \}
\end{array}
$$

Fig. 3. A linear schema with distinct minimal path-faithful dynamic slices

We can now give a corresponding definition for linear schemas.

Definition 10 (path-faithful dynamic slice). Let S be a linear schema containing a label l and V be a set of variables. Let $\rho\, l \in pre(\Pi(S))$ be executable and S' be a quotient of S containing l. We say that S' is a $(\rho l, V)$-path-faithful dynamic slice (PFDS) of S if:

(1) every variable in V defines the same term after $proj_{S'}(\rho)$ as after ρ in S; and
(2) every maximal path through S' which is compatible with ρ has $proj_{S'}(\rho)$ as a prefix.

If the label l occurs at the end of S, so $S = T\, l$ for a schema T, and S' is a $(\rho l, V)$-dynamic slice of S, so $S' = T'\, l$, we simply say that T' is a (ρ, V)-path-faithful dynamic end slice of T.

Theorem 11. Let S be a linear schema and $\rho l \in pre(\Pi(S))$ be executable. Let V be a set of variables and S' be a quotient of S containing l. Then S' is a $(\rho l, V)$-PFDS of S if and only if $\mathscr{M}[\![\rho]\!]_e(v) = \mathscr{M}[\![proj_{S'}(\rho)]\!]_e(v)$ for all $v \in V$ and every expression $p(\mathbf{t}) = X$ that is a consequence of $proj_{S'}(\rho)$ is also a consequence of ρ.

Proof. The statement follows immediately from the two conditions in Definition 10. □

As an example of a path-faithful dynamic end slice, consider the linear schema of Figure 3.

We assume that $V = \{v\}$ and we have the path

$$\rho = (\, \underline{p,\mathsf{T}}\; \underline{g}\; \underline{f}\; \underline{q,\mathsf{T}}\; \underline{h}\; \underline{H}\,)^2\; \underline{p,\mathsf{F}},$$

which passes twice through the body of p, in each case passing through $\underline{q,\mathsf{T}}$, and then leaves the body of p. Thus the value of v after ρ is $f(h(u))$. Thus any $(\{v\}, \rho)$-DPS S' of S must contain f and h in order that (1) is satisfied, and hence contains p and q. By Theorem 11, S' would also have to contain g, since otherwise $p(w) = \mathsf{F}$ would be a consequence of $proj_{S'}(\rho)$, whereas $p(w) = \mathsf{F}$ is not a consequence of ρ. Also, S' would contain the function symbol H, since otherwise $q(g(w), t) = \mathsf{T}$ would be a consequence of $proj_{S'}(\rho)$, but not of ρ. Thus, S itself is the only $(\{v\}, \rho)$-PFDS of S. Observe that the inclusion of the assignment '$t := H(t)$;' has the sole effect of ensuring that for every interpretation i for which $\pi_{S'}(i, e) = \rho$, we get $\pi_{S'}(i, e)$ passes through $\underline{q,\mathsf{T}}$ instead of $\underline{q,\mathsf{F}}$ during its second pass through the body of p, so deleting '$t := H(t)$;' does not alter the value of v after $\pi_{S'}(i, e)$. This suggests that our definition of a dynamic slice may be unnecessarily restrictive, and this motivates the generalisation of Definition 14.

4. A new form of dynamic slicing

Path-faithful dynamic slices of schemas correspond to dynamic program slices, and in order to produce a dynamic slice of a program, we can produce the path-faithful dynamic slice of the corresponding linear schema. In this section we show how this notion of dynamic slicing can be weakened to produce smaller slices for linear schemas, and hence also for programs.

Consider the schema in Figure 3, the path

$$\rho = \underline{p,\mathsf{T}}\,\underline{g}\,\underline{f}\,\underline{q,\mathsf{T}}\,\underline{h}\,\underline{H}\,\underline{p,\mathsf{T}}\,\underline{g}\,\underline{f}\,\underline{q,\mathsf{T}}\,\underline{h}\,\underline{H}\,\underline{p,\mathsf{F}}$$

and variable v. It is straightforward to see that a dynamic slice has to retain the predicate p since it controls a statement ($u := h(u)$) that updates the value of u, and this can lead to a change in the value of v on the next iteration of the loop. Thus, a dynamic slice with regards to v and ρ must retain predicate q. Also, the assignment $t := H(t)$ affects the value of t, and hence the value of q on the second iteration of the loop in ρ, so a (path-faithful) dynamic slice must retain this assignment.

We can observe that in ρ the value of the predicate q on the last iteration of the loop does not affect the final value of v. In addition, in ρ the assignment $t := H(t)$ only affects the value of q on the last iteration of the loop, and this assignment does not influence the final value of v. In this section we define a type of dynamic slice that allows us to eliminate this assignment. At the end of this section we describe a context in which we might be happy to eliminate such assignments.

Proposition 12. Let S be a linear schema and ρ be a path through S.

(1) Let q be a while predicate in S and μ be a terminal path in the body of q in S. Then a word $\alpha\underline{q,\mathsf{T}}\mu\underline{q,\mathsf{F}}\gamma$ is a path in S if and only if $\alpha\underline{q,\mathsf{F}}\gamma$ is a path in S.

(2) Let q be an if predicate in S and $Z \in \{\mathsf{T},\mathsf{F}\}$, and let μ,μ' be terminal paths in the Z-part and $\neg Z$-part, respectively, of q in S. Then a word $\alpha\underline{q,Z}\mu\gamma$ is a path in S if and only if $\alpha\underline{q,\neg Z}\mu'\gamma$ is a path in S.

Furthermore, in both cases, one path is terminal if and only if the other is also terminal.

Proof. Both assertions follow straightforwardly by structural induction from the definition of $\Pi(S)$ in Section 2.1. □

Definition 13. Let S be a linear schema, l be a label and ρ,ρ' be paths through S. Then we say that ρ is simply l-reducible to ρ' if ρ' can be obtained from ρ by one of the following transformations, which we call simple l-reductions:

(1) Replace a segment $\underline{p,\mathsf{T}}\,\sigma\,\underline{p,\mathsf{F}}$ within ρ by $\underline{p,\mathsf{F}}$, where σ is a terminal path in the body of a while predicate p that does not contain l in its body.

(2) Replace a segment $\underline{p,Z}\,\sigma$ within ρ by $\underline{p,\neg Z}$, where σ is a terminal path in the Z-part of an if predicate p, l does not lie in either part of p and the $\neg Z$-part of p is *skip*.

If ρ' can be obtained from ρ by applying zero or more l-reductions, we say that ρ is l-reducible to ρ'. If the condition on the label l is removed from the definition, we use the terms reduction and simple reduction.

By Proposition 12, the transformations given in Definition 13 always produce paths through S. Note that if ρ is l-reducible to ρ', the sequence of function and predicate symbols through which ρ' passes is a subsequence of the sequence through which ρ passes, ρ and ρ' pass through the label l the same number of times and the length of ρ' is not greater than that of ρ.

Definition 14 (dynamic slice). Let S be a linear schema containing a label l and V be a set of variables. Let $\rho\, l \in pre(\Pi(S))$ be executable and S' be a quotient of S containing l. Then we say that S' is a $(\rho l, V)$-dynamic slice (DS) of S if every maximal path through S' compatible with ρ has a prefix ρ' to which $proj_{S'}(\rho)$ is l-reducible and such that every variable in V defines the same term after ρ' as after ρ in S.

If the label l occurs at the end of S, so $S = T\, l$ for a schema T, and S' is a $(\rho l, V)$-dynamic slice of S, so $S' = T'\, l$, then we simply say that T' is a (ρ, V)-dynamic end slice of T.

Consider again the schema in Figure 3 and the path

$$\rho = \underline{p,\mathsf{T}}\,\underline{g}\,\underline{f}\,\underline{q,\mathsf{T}}\,\underline{h}\,\underline{H}\,\underline{p,\mathsf{T}}\,\underline{g}\,\underline{f}\,\underline{q,\mathsf{T}}\,\underline{h}\,\underline{H}\,\underline{p,\mathsf{F}}.$$

Here the quotient T obtained from S by deleting the assignment '$t := H(t);$' is a (ρ, v)-dynamic end slice of S since the path

$$\rho' = \underline{p,\mathsf{T}}\,\underline{g}\,\underline{f}\,\underline{q,\mathsf{T}}\,\underline{h}\,\underline{H}\,\underline{p,\mathsf{T}}\,\underline{g}\,\underline{f}\,\underline{q,\mathsf{F}}\,\underline{p,\mathsf{F}}$$

is simply reducible from $proj_{S'}(\rho)$ and gives the correct final value for v, and ρ' and $proj_{S'}(\rho)$ are the only maximal paths through S' that are compatible with ρ. This shows that a DS of a linear schema may be smaller than a PFDS.

One area in which it is useful to determine the dependence along a path in a program is in the application of test techniques, such as those based on evolutionary algorithms, which automate the generation of test cases to satisfy a structural criterion. These techniques may choose a path to the point of the program to be covered and then attempt to generate test data that follows the path – see, for example, Harman *et al.* (2002), Jones *et al.* (1996), Wegener *et al.* (1996) and Wegener *et al.* (1997). If we can determine the inputs relevant to this path, we can focus on these variables in the search, which effectively reduces the size of the search space. Current techniques use static slicing, but there is potential for using dynamic slicing, in particular, the type of dynamic slice defined here, to make the dependence information more precise.

5. A linear schema with two minimal path-faithful dynamic slices

Given a linear schema, a variable set V and a path ρ through S, we wish to establish information about the set of all (ρ, V)-dynamic slices, which is partially ordered by set-theoretic inclusion of function and predicate symbols. In particular, it would be interesting to obtain conditions on S that would ensure that minimal slices were unique since under such conditions it may be feasible to produce minimal slices in an incremental manner, deleting one statement at a time until no more statements can be removed. As we now show, however, this cannot be done for arbitrary linear schemas, whether or not slices are

```
while P(v)  {
            if Q(v)    then  {
                             if q(v) then    {
                                             x := g_good();
                                             v := G_good(x, v);
                                             }
                                     else    {
                                             x := g_bad();
                                             v := G_bad(x, v);
                                             }

                             if s1(v) then   x := g1();
                             if s2(v) then   x := g2();

                             if t(x)   then  v := H(v);

                             }
                       else  skip;
            v := J(v);
            }
```

Fig. 4. A linear schema with distinct minimal dynamic and path-faithful dynamic slices

required to be path-faithful. To see this, consider the schema S of Figure 4 and the slicing criterion defined by the variable v and the terminal path ρ that enters the body of P 5 times as follows:

1st time: ρ passes through g_{good} and H, but not through g_1 or g_2.

2nd time: ρ passes through g_{good}, g_1 and H, but not through g_2.

3rd time: ρ passes through g_{good}, g_2 and H, but not through g_1.

4th time: ρ passes through g_{bad}, g_1, g_2 and H.

5th time: ρ passes through $\underline{Q, \mathsf{F}}$.

We define the quotient S_1 of S by deleting the entire if statement guarded by s_2, and define S_2 analogously by interchanging the suffices 1 and 2. By Theorem 11, S_1 and S_2 are both (ρ, v)-PFDS's of S since $t(x)$ will still evaluate to T over the path $proj_{S_1}(\rho)$ or $proj_{S_2}(\rho)$ on paths 2–4. On the other hand, if the if statements guarded by s_1 and s_2 are both deleted, then on the 4th path, $t(x)$ may evaluate to F, since g_{bad} never occurs in the predicate term defined by $t(x)$ along ρ, hence the final value of v may contain fewer occurrences of H in the slice than after ρ. Furthermore, every (ρ, v)-DS of S must contain the function symbols J, H, G_{good} and G_{bad}, and thus g_{good} and g_{bad} since the final term defined by v contains these symbols, so S_1 and S_2 are minimal (ρ, v)-DS's, and are also both path-faithful.

6. Decision problems for dynamic slices

In this section we establish complexity bounds for two problems: whether a quotient S' of a linear schema S is a dynamic slice, and whether a linear schema S has a non-trivial dynamic slice. We consider the problems both with and without the requirement that dynamic slices be path-faithful.

Definition 15 (maximal common prefix of a pair of words). The maximal common prefix of words σ, σ' is denoted by $maxpre(\sigma, \sigma')$. For example, the maximal common prefix of the words $x_1x_2x_3x_4$ and $x_1x_2yx_4$ over the five-word alphabet $\{x_1, x_2, x_3, x_4, y\}$ is x_1x_2: that is, $maxpre(x_1x_2x_3x_4, x_1x_2yx_4) = x_1x_2$.

Lemma 16. Let S be a linear schema containing a label l and let ρ, ρ' be paths through S. Suppose ρ is l-reducible to ρ'. Then there is a sequence $\rho_1 = \rho, \ldots, \rho_n = \rho'$ such that each ρ_i is simply l-reducible to ρ_{i+1}, and $maxpre(\rho_i, \rho_{i+1})$ is always a strict prefix of $maxpre(\rho_{i+1}, \rho_{i+2})$.

Proof. This follows from the fact that the two transformation types commute. Since ρ is l-reducible to ρ', there is a sequence $\rho_1 = \rho, \ldots, \rho_n = \rho'$ such that each ρ_i is obtained from ρ_{i-1} by a simple l-reduction, and we may assume that n is minimal. Thus, for each $i < n$, using the definition of a simple l-reduction, we can write $\rho_i = \alpha_i \underline{p_i, Z_i} \beta_i \gamma_i$ and $\rho_{i+1} = \alpha_i \underline{p_i, \neg Z_i} \gamma_i$. If every α_i is a strict prefix of α_{i+1}, the sequence of paths ρ_i already satisfies the required property. Thus we may assume that for some minimal i, α_i is not a strict prefix of α_{i+1}.

We now compare the two ways of writing

$$\rho_{i+1} = \alpha_i \underline{p_i, \neg Z_i} \gamma_i = \alpha_{i+1} \underline{p_{i+1}, Z_{i+1}} \beta_{i+1} \gamma_{i+1}.$$

Clearly, α_{i+1} is a prefix of α_i. We consider three cases:

(1) $\alpha_i = \alpha_{i+1}$:
So the first letter of ρ_{i+1} after α_i is $\underline{p_i, \neg Z_i} = \underline{p_{i+1}, Z_{i+1}}$. If $p_i = p_{i+1}$ were a while predicate, $Z_i = \mathsf{T}$ would follow from the fact that ρ_i is l-reducible to ρ_{i+1}, and $Z_{i+1} = \mathsf{T}$ would follow similarly from the pair ρ_{i+1}, ρ_{i+2}, giving a contradiction. Hence p_i must be an if predicate, so the $\neg Z_i$-part and the $Z_i = \neg Z_{i+1}$-part of p is *skip* from the definition of l-reduction and, therefore, $\rho_{i+2} = \rho_i$, which contradicts the minimality of n.

(2) α_{i+1} is a strict prefix of α_i and $\alpha_i \underline{p_i, \neg Z_i}$ is a prefix of $\alpha_{i+1} \underline{p_{i+1}, Z_{i+1}} \beta_{i+1}$:
So $\underline{p_i, \neg Z_i}$ occurs in β_{i+1}, and we can write

$$\alpha_i = \alpha_{i+1} \underline{p_{i+1}, Z_{i+1}} \delta_1$$
$$\beta_{i+1} = \delta_1 \underline{p_i, \neg Z_i} \delta_2.$$

Since ρ_i can be obtained by replacing $\underline{p_i, \neg Z_i}$ by $\underline{p_i, Z_i} \beta_i$ after α_i in ρ_{i+1}, we have

$$\rho_i = \alpha_{i+1} \underline{p_{i+1}, Z_{i+1}} \delta_1 \underline{p_i, Z_i} \beta_i \delta_2 \gamma_{i+1}.$$

By our assumption on the pair (ρ_i, ρ_{i+1}), we have that β_i is a terminal path in the body or Z_i-part of p_i, so, by Proposition 12, we have that $\delta_1 \underline{p_i, Z_i} \beta_i \delta_2$ is a terminal path in the body or Z_{i+1}-part of p_{i+1}, so we can get ρ_{i+2} from ρ_i by a simple l-reduction, by replacing $\underline{p_{i+1}, \neg Z_{i+1}}\delta_1 \underline{p_i, Z_i}\beta_i\delta_2$ by $\underline{p_{i+1}, \neg Z_{i+1}}$ in ρ_i, again contradicting the minimality of n.

(3) α_{i+1} is a strict prefix of α_i and $\alpha_i \underline{p_i, \neg Z_i}$ is not a prefix of $\alpha_{i+1}\underline{p_{i+1}, Z_{i+1}}\beta_{i+1}$:
So we can write

$$\alpha_i = \alpha_{i+1}\underline{p_{i+1}, Z_{i+1}}\beta_{i+1}\,\delta.$$

We now change the order of the two reductions by replacing ρ_{i+1} in the sequence by

$$\hat{\rho}_{i+1} = \alpha_{i+1}\,\underline{p_{i+1}, \neg Z_{i+1}}\delta \underline{p_i, Z_i}\beta_i\gamma_i,$$

which, by two applications of Proposition 12, is a path through S. In effect we are replacing $\underline{p_{i+1}, Z_{i+1}}\beta_{i+1}$ by $\underline{p_{i+1}, \neg Z_{i+1}}$ before replacing $\underline{p_i, Z_i}\beta_i$ by $\underline{p_i, \neg Z_i}$, instead of in the original order. Since

$$\rho_{i+2} = \alpha_{i+1}\,\underline{p_{i+1}, \neg Z_{i+1}}\delta \underline{p_i, \neg Z_i}\gamma_i,$$

we have $maxpre(\rho_i, \hat{\rho}_{i+1})$ is a strict prefix of $maxpre(\hat{\rho}_{i+1}, \rho_{i+2})$. Thus, by the minimality of i, after not more than $n-i$ such replacements, the maximal common prefixes of consecutive paths in the resulting sequence will be strictly increasing in length, as required. □

Theorem 17. Let S be a linear schema, l be a label and $\rho, \rho' \in pre(\Pi(S))$. Then the problem of determining whether ρ is l-reducible to ρ' is decidable in polynomial time.

Proof. By Lemma 16, ρ is l-reducible to ρ' if and only if ρ can be simply l-reduced to some $\rho_2 \in pre(\Pi(S))$ such that ρ_2 is l-reducible to ρ' and $maxpre(\rho, \rho_2)$ is a strict prefix of $maxpre(\rho_2, \rho')$, so $maxpre(\rho, \rho_2) = maxpre(\rho, \rho')$. Thus, ρ_2 exists satisfying these criteria if and only if ρ and ρ' have prefixes $\tau\sigma$ and $\tau\sigma'$, respectively, such that σ' is obtained from σ by either of the transformations given in Definition 13, and ρ_2 is obtained from ρ by replacing σ by σ'. Thus σ can be computed in polynomial time if it exists, and this procedure can be iterated using ρ_2 in place of ρ. The number of iterations needed is bounded by the number of letters in ρ', thus proving the Theorem. □

Theorem 18. Let S be a linear schema containing a label l and $\rho l \in pre(\Pi(S))$ be executable. Let V be a set of variables and S' be a quotient of S containing l. Then:

(1) The problem of determining whether S' is a $(\rho l, V)$-path-faithful dynamic slice of S can be decided in polynomial time.
(2) The problem of determining whether S' is a $(\rho l, V)$-dynamic slice of S lies in co-NP.

Proof.

(1) This part follows immediately from the conclusion of Theorem 11 since, given any predicate-free schema T and any variable v, the term $\mathscr{M}[\![T]\!]_e(v)$ is computable in polynomial time.
(2) We proceed as follows. Any path $\rho' l$ through S' such that ρ' is l-reducible from $proj_{S'}(\rho)$ has length $\leqslant |proj_{S'}(\rho)l|$. We compute a path τ through S' of length $\leqslant |proj_{S'}(\rho)l|$, with the strict inequality if and only if τ is terminal. This can be done in NP-time by starting with the empty path and successively appending letters to it until a terminal

path, or one of length $|proj_{S'}(\rho)l|$ is obtained. We then test whether τ is compatible with ρ and does not have a prefix $\rho' l$ through S' such that ρ' is l-reducible from $proj_{S'}(\rho)$ and $\mathscr{M}[\![\rho]\!]_e(v) = \mathscr{M}[\![\rho']\!]_e(v)$ for all $v \in V$. By Theorem 17, this can be done in polynomial time. If no such prefix exists for the given τ, then no longer path through S' having prefix τ has such a prefix either, so S' is not a $(\rho l, V)$-dynamic slice of S. Conversely, if S' is not a $(\rho l, V)$-dynamic slice of S, then a path τ can be computed satisfying the conditions given, which completes the proof of (2). □

Theorem 19. Let S be a linear schema, $\rho l \in pre(\Pi(S))$ be executable and V be a set of variables.

(1) The problem of deciding whether there exists a non-trivial $(\rho l, V)$-path-faithful dynamic slice of S is NP-complete.
(2) The problem of deciding whether there exists a non-trivial $(\rho l, V)$-dynamic slice of S lies in PSPACE and is NP-hard.

Proof. To prove membership in NP for part (1), it is enough to observe that a quotient S' of S can be guessed in NP-time, and using Theorem 11, the problem of determining whether S' is a non-trivial $(\rho l, V)$-path-faithful dynamic slice of S can be decided in polynomial time.

The proof of Membership of PSPACE for part (2) follows similarly from Theorem 18 (2) and the fact that co-NP⊆PSPACE=NPSPACE.

To show NP-hardness of both problems, we use a polynomial-time reduction from 3SAT, which is known to be an NP-hard problem (Cook 1971). An instance of 3SAT can be constructed from a set $\Theta = \{\theta_1, \ldots, \theta_n\}$ and a propositional formula $\alpha = \bigwedge_{k=1}^{m} \alpha_{k1} \vee \alpha_{k2} \vee \alpha_{k3}$, where each α_{ij} is either θ_k or $\neg\theta_k$ for some k. The problem is satisfied if there exists a valuation $\delta : \Theta \to \{\mathsf{T}, \mathsf{F}\}$ under which α evaluates to T. We will construct a linear schema S containing a variable v and a terminal path ρ through S such that S has a non-trivial (ρ, v)-dynamic end slice if and only if α is satisfiable, in which case this quotient is also a (ρ, v)-path-faithful dynamic end slice. The schema S is as in Figure 5.

We let the function symbol g_i correspond to θ_i and g_i' correspond to $\neg\theta_i$. The terminal path ρ passes a total of $4 + 3n + 6n(n-1) + m$ times through the body of S, and then leaves the body. The paths within the body of S are of fourteen types, and are listed as follows, in the order in which they occur along ρ; note that only those of type (5) depend on the value of α.

(0) There are 3 paths of this type:

(0.1) ρ passes through g_{good}, g_{link} and $F_{\mathrm{link/reset}}$, and through no other assignment apart from H.

(0.2) ρ passes through g_{reset} and $F_{\mathrm{link/reset}}$, and through no other assignment apart from H.

(0.3) ρ passes through g_{bad}, g_{link} and $F_{\mathrm{link/reset}}$, and through no other assignment apart from H.

(1) There is 1 path of this type:
ρ passes through g_{good} and F_{test}, and through no other assignment apart from H.

$$
\begin{array}{lll}
\textit{while}\ p(v) & \{ & \\
& v := H(v); & \\
& \textit{if}\ q_{\mathrm{good}}(v)\ \textit{then} & x := g_{\mathrm{good}}(); \\
& \textit{if}\ q_{\mathrm{bad}}(v)\ \textit{then} & x := g_{\mathrm{bad}}(); \\
& & \\
& \textit{if}\ q_{\mathrm{link}}(v) & \textit{then}\ b := g_{\mathrm{link}}(x); \\
& \textit{if}\ q_{reset}(v) & \textit{then}\ b := g_{\mathrm{reset}}(); \\
& \textit{if}\ Q_{\mathrm{link/reset}}(v) & \textit{then}\ v := F_{\mathrm{link/reset}}(b, v); \\
& & \\
& \textit{if}\ q_1(v) & \textit{then}\ x := g_1(b); \\
& \textit{if}\ q_1'(v) & \textit{then}\ x := g_1'(b); \\
& \vdots & \\
& \textit{if}\ q_n(v) & \textit{then}\ x := g_n(b); \\
& \textit{if}\ q_n'(v) & \textit{then}\ x := g_n'(b); \\
& & \\
& \textit{if}\ Q_{\mathrm{test}}(v) & \textit{then}\ \ \textit{if}\ q_{\mathrm{test}}(x)\ \textit{then}\ v := F_{\mathrm{test}}(v); \\
& \} &
\end{array}
$$

Fig. 5. The linear schema used in Theorem 19 to establish the NP-hardness of deciding the existence of dynamic slices

(2) There are n paths of this type:
For each $i \leqslant n$, ρ passes through g_{good}, g_{reset}, g_i and F_{test} and through no other assignment apart from H.

(2′) There are n paths of this type:
They are the same as for type (2), but with g_i' in place of g_i.

(3) There are n paths of this type:
For each $i \leqslant n$, ρ passes through g_{good}, g_{link}, g_i' and F_{test}, and through no other assignment apart from H.

(4) There are $6n(n-1)$ paths of this type:
For each $i \neq j \leqslant n$, ρ passes 6 times consecutively through the body of S, as follows:

  (4.1) The first time, it passes through g_{good}, g_{reset} and g_i, but not through q_{test} or any other assignment apart from H.

  (4.2) The second time, it passes through g_{link} and g_i', but not through q_{test} or any other assignment apart from H.

  (4.3) The third time, it passes through g_{reset} and g_j and F_{test}, but through no other assignment apart from H.

(4′) There are $6n(n-1)$ paths of this type:

  (4.1′) As for types (4.1), but with g_j' in place of g_j.

  (4.2′) As for types (4.2), but with g_j' in place of g_j.

  (4.3′) As for types (4.3), but with g_j' in place of g_j.

(5) There are m paths of this type:

For each $i \leqslant m$, ρ passes through g_{bad} and g_{reset}, and then through the 3 function symbols corresponding to the implicants $\alpha_{i1}, \alpha_{i2}, \alpha_{i3}$, and then through F_{test} and through no other assignment apart from H.

Before continuing with the proof, we first record the following facts about the terminal path ρ.

(*a*) ρ passes through all three assignments to v and through both assignments to b.

(*b*) All three assignments to v in S also reference v, and hence if there exists a terminal path σ through any slice T of S such that $\mathscr{M}[\![\rho]\!]_e(v) = \mathscr{M}[\![\sigma]\!]_e(v)$, then the following hold:

(*b*0) By (*a*), T contains H, F_{test}, $F_{\text{link/reset}}$ and thus also g_{link}, g_{reset}, g_{good} and g_{bad} because of the type (0) paths, so it contains the predicates controlling these function symbols.

(*b*1) By (*a*), σ passes through all the assignments to v in S in the same order as ρ does.

(*b*2) σ and ρ enter the body of p the same number of times, which is the depth of the nesting of H in the term $\mathscr{M}[\![\rho]\!]_e(v)$.

(*b*3) For any function symbol f in S assigning to v and for all $k \geqslant 0$, we have v defines the same term after the kth occurrence of f in ρ and σ since this term is the unique subterm of $\mathscr{M}[\![\rho]\!]_e(v)$ containing k nested occurrences of f whose outermost function symbol is f.

(*b*4) For any predicate q in T and for all $k \geqslant 0$, we have σ and ρ pass the same way through q at the kth occurrence of q. For $q \neq q_{\text{test}}$, this follows from (*b*3) applied to H or $F_{\text{link/reset}}$. For $q = q_{\text{test}}$, it follows from (*b*1) and (*b*4) applied to Q_{test}.

(*b*5) $proj_T(\rho) = \sigma$. To see this, we assume $\sigma' \underline{q, Z} \in pre(\sigma)$, whereas $\sigma' \underline{q, \neg Z} \in pre(proj_T(\rho))$, where $\sigma' \underline{q, Z}$ contains k q's, which contradicts (*b*4) immediately, so $proj_T(\rho) = \sigma$ follows from Lemma 3 and the fact that $proj_T(\rho)$ and σ are both terminal paths through T.

(*c*) ρ never passes through the predicate terms $q_{\text{test}}(g_{\text{bad}}())$ or $q_{\text{test}}(g_i'(g_{\text{link}}(g_i(g_{\text{reset}}()))))$.

(*d*) For any prefix ρ' of ρ, the term $\mathscr{M}[\![\rho']\!]_e(v)$ does not contain any g_i or g_i'. This is because these symbols, which do not occur on the type (0) paths, assign to x, whereas $F_{\text{link/reset}}$, which does not occur on ρ after the type (0) paths, is the only assignment to v referencing a variable other than v.

We now continue the proof:

($\Rightarrow$) Let T be a non-trivial (ρ, v)-DS of S. By (*b*5), T is a (ρ, v)-PFDS of S, and by (*b*0), T contains all symbols in S, with the possible exception of some of those having the form g_i, g_i' and the if predicates q_i, q_i' controlling them. Thus we only need to show that α is satisfiable.

We first show that if T does not contain a symbol g_j, then for all $i \neq j$, it cannot contain both g_i and g_i'. Consider the type (4.3) path for the values i, j. If T contains g_i and g_i', but not g_j, then when q_{test} is reached on path (4.3), the predicate term thus defined, built up over paths (4.1), (4.2), (4.3), is $q_{\text{test}}(g_i'(g_{\text{link}}(g_i(g_{\text{reset}}()))))$, which does not occur along the path ρ, contradicting Theorem 11. By considering type (4.3′) paths, the same assertion holds for the symbols g_j'. Since $T \neq S$ holds, this implies that T contains at most one element of each set $\{g_i, g_i'\}$.

We now show that for each $i \leqslant m$, T contains at least one symbol corresponding to an element in $\{\alpha_{i1}, \alpha_{i2}, \alpha_{i3}\}$. If this is false, then the predicate term $q_{\text{test}}(g_{\text{bad}}())$, which does not occur along the path ρ, would be defined on the ith type (5) path, contradicting Theorem 11.

Thus, α is satisfied by any valuation δ such that for all $i \leqslant n$, T contains $g_i \Rightarrow \delta(\theta_i) = \mathsf{T}$ and T contains $g_i' \Rightarrow \delta(\neg\theta_i) = \mathsf{T}$. But since T contains at most one element of each set $\{g_i, g_i'\}$, such a valuation exists.

($\Leftarrow$) Conversely, suppose α is satisfiable by a valuation $\delta : \Theta \to \{\mathsf{T}, \mathsf{F}\}$, and let T be the quotient of S that contains each g_i and q_i if and only if $\delta(\theta_i) = \mathsf{T}$, and contains g_i' and q_i' otherwise, and contains all the other symbols of S. We show that T is a (ρ, v)-DPS of S. By Theorem 11, it suffices to show that all predicate terms occurring along $proj_T(\rho)$ also occur along ρ with the same associated value from $\{\mathsf{T}, \mathsf{F}\}$, since by ($d$), $\mathscr{M}[\![proj_T(\rho)]\!]_e(v) = \mathscr{M}[\![\rho]\!]_e(v)$.

By (d), all predicate terms occurring in $proj_T(\rho)$ but not ρ must occur at q_{test} rather than at a predicate referencing v. We consider each path type separately and show that no such predicate terms exist.

(0) These paths do not pass through q_{test}.

(1) $proj_T(\rho)$ defines $q_{\text{test}}(g_{\text{good}}())$, which also occurs along ρ in the type (1) path.

(2) If T does not contain g_i then $proj_T(\rho)$ defines $q_{\text{test}}(g_{\text{good}}())$, which occurs along ρ in the type (1) path. Otherwise $proj_T(\rho)$ defines $q_{\text{test}}(g_i(g_{\text{reset}}()))$, which occurs along ρ in a type (2) path.

(2′) This type is similar to type (2).

(3) If T does not contain g_i', then $proj_T(\rho)$ defines $q_{\text{test}}(g_{\text{good}}())$, which occurs along ρ in the type (1) path. Otherwise, $proj_T(\rho)$ defines $q_{\text{test}}(g_i'(g_{\text{link}}(g_{\text{good}}())))$, which also occurs along ρ in a type (3) path.

(4) If T contains g_i but not g_j or g_i', then $proj_T(\rho)$ defines $q_{\text{test}}(g_i(g_{\text{reset}}()))$, which occurs along ρ in a type (2) path. If T contains g_i' but not g_j or g_i, then $proj_T(\rho)$ defines $q_{\text{test}}(g_i'(g_{\text{link}}(g_{\text{good}}())))$, which also occurs along ρ in a type (3) path. Finally, if T contains g_j, then $proj_T(\rho)$ defines $q_{\text{test}}(g_j(g_{\text{reset}}()))$, which occurs along ρ in a type (2) path.

(4′) This type is similar type (4).

(5) Since the valuation δ satisfies α, for each $k \leqslant m$, we have T contains at least one of the 3 function symbols corresponding to the implicants $\alpha_{k1}, \alpha_{k2}, \alpha_{k3}$, so $proj_T(\rho)$ defines $q_{\text{test}}(g_i(g_{\text{reset}}()))$ or $q_{\text{test}}(g_i'(g_{\text{reset}}()))$ for some $i \leqslant n$, which occur along ρ in a type (2) or (2′) path.

Since it is clear that the schema S and the path ρ can be constructed in polynomial time from the formula α, this concludes the proof of the theorem. □

7. Conclusion and further directions

We have reformulated Korel and Laski's definition of a dynamic slice of a program as applied to linear schemas, which is the normal level of program abstraction assumed by slicing algorithms, and have also given a less restrictive slicing definition. In addition, we have given P and co-NP complexity bounds for the problem of deciding whether a given quotient of a linear schema satisfies them. We conjecture that the problem of determining whether a quotient S' of a linear schema S is a general dynamic slice with respect to a given path and variable set is co-NP-complete. Future work should attempt to resolve this.

We have also shown that it is not possible to decide in polynomial time whether a given linear schema has a non-trivial dynamic slice using either definition, assuming P$\neq$NP. It is possible that this NP-hardness result can be strengthened to PSPACE-hardness for general dynamic slices, since in this case the problem does not appear to lie in NP.

We have also shown that minimal dynamic slices (whether they are path-faithful or not) are not unique. Placing further restrictions on either the schemas or the paths may ensure the uniqueness of dynamic slices or lower the complexity bounds proved in Section 6, and this should be investigated.

Schemas correspond to single programs/methods, so results for schemas cannot be directly applied when analysing a program that has multiple procedures, and thus the results in this paper do not apply to inter-procedural slicing. It would be interesting to extend schemas with procedures and then analyse both dynamic slicing and static slicing for such schemas.

These results have several practical ramifications. First, since the problem of deciding whether a linear schema has a non-trivial dynamic slice is computationally hard, this result must also hold for programs. A further consequence is that the problem of producing minimal dynamic slices must also be computationally hard. We have also defined a new notion of a dynamic slice for linear schemas (and thus for programs) that places strictly weaker constraints on the slice and hence can lead to smaller dynamic slices. Finally, the fact that minimal dynamic slices need not be unique suggests that algorithms that identify and then delete one statement at a time can lead to suboptimal dynamic slices.

References

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

Agrawal, H. and Horgan, J. R. (1990) Dynamic program slicing. In: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation. *SIGPLAN Notices* **25** 246–256.

Ashcroft, E. A. and Manna, Z. (1975) Translating program schemas to while-schemas. *SIAM Journal on Computing* **4** (2) 125–146.

Beszédes, A., Gergely, T., Szabó, Z. M., Csirik, J. and Gyimóthy, T. (2001) Dynamic slicing method for maintenance of large C programs. In: *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, IEEE Computer Society 105–113.

Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, Á. and Korel, B. (2006) Theoretical foundations of dynamic program slicing. *Theoretical Computer Science* **360** (1-3) 23–41.

Canfora, G., Cimitile, A., De Lucia, A. and Lucca, G. A. D. (1994) Software salvaging based on conditions. In: *Proceedings of the International Conference on Software Maintenance (ICSM'96)*, IEEE Computer Society 424–433.

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

Cook, S. A. (1971) The complexity of theorem-proving procedures. In: *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, ACM Press 151–158.

Danicic, S., Fox, C., Harman, M., Hierons, R., Howroyd, J. and Laurence, M. R. (2005) Static program slicing algorithms are minimal for free liberal program schemas. *The Computer Journal* **48** (6) 737–748.

Danicic, S., Harman, M., Hierons, R., Howroyd, J. and Laurence, M. R. (2007) Equivalence of linear, free, liberal, structured program schemas is decidable in polynomial time. *Theoretical Computer Science* **373** (1-2) 1–18.

De Lucia, A., Fasolino, A. R. and Munro, M. (1996) Understanding function behaviours through program slicing. In: *Proceedings of the* 4^{th} *IEEE Workshop on Program Comprehension*, IEEE Computer Society 9–18.

Gallagher, K. B. (1992) Evaluating the surgeon's assistant: Results of a pilot study. In: *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society 236–244.

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

Gopal, R. (1991) Dynamic program slicing based on dependence graphs. In: *Proceedings IEEE Conference on Software Maintenance*, IEEE Computer Society 191–200.

Greibach, S. (1975) Theory of program structures: schemes, semantics, verification. *Springer-Verlag Lecture Notes in Computer Science* **36**.

Harman, M., Fox, C., Hierons, R. M., Hu, L., Danicic, S. and Wegener, J. (2002) Vada: A transformation-based system for variable dependence analysis. In: *Proceedings IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, IEEE Computer Society 55–64.

Harman, M., Hierons, R. M., Danicic, S., Howroyd, J. and Fox, C. (2001) Pre/post conditioned slicing. In: *Proceedings IEEE International Conference on Software Maintenance (ICSM'01)*, IEEE Computer Society 138–147.

Hunt, H. B., Constable, R. L. and Sahni, S. (1980) On the computational complexity of program scheme equivalence. *SIAM J. Comput* **9** (2) 396–416.

Ianov, Y. I. (1960) The logical schemes of algorithms. In: *Problems of Cybernetics* **1**, Pergamon Press 82–140.

Jones, B., Sthamer, H.-H. and Eyres, D. (1996) Automatic structural testing using genetic algorithms. *The Software Engineering Journal* **11** 299–306.

Kamkar, M. (1993) *Interprocedural dynamic slicing with applications to debugging and testing*, Ph.D. Thesis, Department of Computer Science and Information Science, Linköping University, Sweden. (Available as Linköping Studies in Science and Technology, Dissertations, Number 297.)

Kamkar, M. (1998) Application of program slicing in algorithmic debugging. In: Harman, M. and Gallagher, K. (eds.) *Information and Software Technology* **40** (Special Issue on Program Slicing) 637–645.

Kamkar, M., Shahmehri, N. and Fritzson, P. (1992) Interprocedural dynamic slicing. In: Bruynooghe, M. and Wirsing, M. (eds.) Programming Language Implementation and Logic

Programming, 4th International Symposium, PLILP'92, Proceedings. *Springer-Verlag Lecture Notes in Computer Science* **631** 370–384.

Korel, B. (1995) Computation of dynamic slices for programs with arbitrary control flow. In: Ducassé, M. (ed.) 2^{nd} *International Workshop on Automated Algorithmic Debugging (AADEBUG'95)* 71–86.

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

Korel, B. and Rilling, J. (1998) Dynamic program slicing methods. In: Harman, M. and Gallagher, K. (eds.) *Information and Software Technology* **40** (Special Issue on Program Slicing) 647–659.

Laurence, M. R. (2005) Characterising minimal semantics-preserving slices of function-linear, free, liberal program schemas. *Journal of Logic and Algebraic Programming* **72** (2) 157–172.

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

Laurence, M. R., Danicic, S., Harman, M., Hierons, R. and Howroyd, J. (2004) Equivalence of linear, free, liberal, structured program schemas is decidable in polynomial time. Technical Report ULCS-04-014, University of Liverpool. (Available at `http://www.csc.liv.ac.uk/research/techreports/`.)

Luckham, D. C., Park, D. M. R. and Paterson, M. S. (1970) On formalised computer programs. *Journal of Computer and System Sciences* **4** (3) 220–249.

Lyle, J. R. and Weiser, M. (1987) Automatic program bug location by program slicing. In: *Proceedings* 2^{nd} *International Conference on Computers and Applications*, IEEE Computer Society 877–882.

Manna, Z. (1974) *Mathematical Theory of Computation*, McGraw–Hill.

Müller-Olm, M. (2004) Precise interprocedural dependence analysis of parallel programs. *Theoretical Computer Science* **31** (1) 325–388.

Müller-Olm, M. and Seidl, H. (2004a) Computing polynomial program invariants. *Information Processing Letters* **91** (5) 233–244.

Müller-Olm, M. and Seidl, H. (2004b) Precise interprocedural analysis through linear algebra. In Proceedings of Principles of Programming Languages (POPL'04). *SIGPLAN Notices* **39** (1) 330–341.

Paterson, M. S. (1967) *Equivalence Problems in a Model of Computation*, Ph.D. thesis, University of Cambridge.

Rutledge, J. D. (1964) On Ianov's program schemata. *J. ACM* **11** (1) 1–9.

Sabelfeld, V. K. (1990) An algorithm for deciding functional equivalence in a new class of program schemes. *Theoretical Computer Science* **71** 265–279.

Wegener, J., Grimm, K., Grochtmann, M., Sthamer, H. and Jones, B. F. (1996) Systematic testing of real-time systems. In: *Proceedings 4th International Conference on Software Testing Analysis and Review (EuroSTAR 96)*.

Wegener, J., Sthamer, H., Jones, B. F. and Eyres, D. E. (1997) Testing real-time systems using genetic algorithms. *Software Quality* **6** (2) 127–135.

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

Weiser, M. and Lyle, J. R. (1985) Experiments on slicing-based debugging aids. In: Soloway, E. and Iyengar, S. (eds.) *Empirical studies of programmers*, chapter 12, Ablex Publishing Corporation 187–197.