Semantic Characterisation of Dead Control-Flow Paths

Ian Hayes[2] Colin Fidge[1] Karl Lermer[1]

[1]Software Verification Research Centre, and
[2]School of Information Technology and Electrical Engineering,
The University of Queensland, Queensland 4072, Australia.

Abstract

Many program verification, testing and performance prediction techniques rely on analysis of statically-identified control-flow paths. However, some such paths may be 'dead' because they can never be followed at run time, and should therefore be excluded from analysis. It is shown how the formal semantics of those statements comprising a path provides a sound theoretical foundation for identification of dead paths.

Keywords and phrases: Program semantics; Weakest liberal preconditions; Dead/false/infeasible control-flow paths; Static program analysis.

1 Introduction

Many techniques in program analysis, testing and timing prediction work by statically identifying all of the possible control-flow paths through a program and then studying each one in isolation. However, due to the boolean expressions guarding entry to conditional and iterative constructs, and limits on the ranges of input values, many statically-valid paths can never be followed at run time. Including these 'dead' control-flow paths in program analysis may lead to inaccurate results and wasted effort. Program timing prediction, for example, will produce overly pessimistic estimates if the execution times for paths that can never be followed are included. There is thus a strong incentive to identify dead paths, so that they can be excluded from program analysis [1, 2].

The role of Dijkstra's weakest preconditions [3] in defining the semantics of programming language statements is widely appreciated. Here we explain that weakest (liberal) precondition semantics can also serve as a formal basis for identifying dead control-flow paths. To achieve this, the component parts of compound statements, such as conditional and iterative constructs, must be treated as separate primitive statements in their own right, because particular control-flow paths traverse only part of the overall statement.

Some of the ideas presented below are well known in the formal methods community, but are unfamiliar to practising programmers. One aim here is to bring this knowledge to a wider audience in a stand-alone form. Section 2 briefly describes situations in which dead-path analysis is useful, and Section 3 reviews related work. Section 4 introduces a small motivational example. Section 5 presents our formal characterisation of what it means for a path to be 'dead'. Section 6 completes the formalism by giving semantic definitions for the programming language statements found within paths. Sections 7 and 8 then return to the example and show how various paths through the program can be profitably analysed. Section 9 concludes by briefly discussing practical issues.

2 Motivation

There are many situations in which we want to statically extract control-flow paths from imperative program code for separate analysis.

- In open-box testing we need to identify control-flow paths in order to ensure, for instance, that each path through the program has been exercised at least once [4, p. 309].

- To predict the worst-case execution time (WCET) of a real-time program, we typically need to take the maximum of the execution times over all possible control-flow paths [5, 6].

- When analysing a real-time program to identify the WCET constraints that it places on the generated object code, we must explore all possible paths which end at critical timing points [7].

- To identify coding errors, such as uninitialised variables appearing in expressions, or ineffective assignments, we need to apply data-flow analysis to each possible control-flow path [8].

In each of these applications it is important that we can eliminate those paths that will never be followed at run time. In the literature, such paths are often referred to as *false* [5] or *infeasible* [9]. Here we favour the term *dead* [6] which avoids confusion with the (related but different) predicate-transformer notion of infeasibility [10, p. 11].

Applying analysis techniques to dead paths wastes time and effort. For example, when selecting paths to be exercised during program testing, we must ensure that dead paths are excluded, to avoid futile attempts to force the program to traverse them [1]. More seriously, apparent errors detected in dead paths during, for instance, data-flow analysis may cause correct programs to be diagnosed as incorrect. Similarly, performance analyses that incorporate the time required to follow dead paths may overestimate the program's execution time—overly pessimistic results are a major problem for current methods of timing analysis [2]. Many algorithms have been proposed for identifying and eliminating dead paths [11]. One of our goals is to provide a semantic basis for the correctness of such algorithms.

3 Previous work

Some analysis techniques place the burden of identifying dead paths on the programmer. Based on their understanding of the program's *intended* control flow, the programmer is required to annotate the program to explicitly indicate which paths are dead or, conversely, which paths are possible. For instance, Park [12] defines a 'path language' in which regular expressions over statement labels can be used to explicitly describe possible control-flow paths. Similarly, Li et al. [13] and Puschner and Schedl [9] allow the programmer to provide integer linear constraint equations based on how many times each statement label is expected to be passed. A similar capability is achieved by Chapman et al. [6] who allow 'mode annotations' in the source program to explicitly state under what circumstances a conditional alternative is dead, and the number of iterations expected for loops starting in different states. Clearly, however, approaches which rely on the programmer to decide which paths are dead are labour-intensive and error-prone.

Dead path identification can also be achieved automatically in some situations. A number of techniques use symbolic execution to identify (some) dead paths [14, 15]. For instance, the SPADE program analysis tool uses symbolic execution to identify dead paths and, significantly, the correctness of its algorithm for calculating 'path traversal conditions' is justified in terms of weakest precondition semantics [8, §8.4.1]. Following a path in which conditional statements make contradictory choices will cause the predicate representing the system state to become 'false' [6]. Similarly, some algorithms identify *correlations* between alternatives in consecutive branching statements, using the logical relationships between the conditions that choose which branches to follow [1]. Paths following mutually-exclusive alternatives can be identified as dead, and thus reduce the number of paths to be analysed [11]. It is, however, impossible in general to eliminate all dead paths using these approaches since the broad *ranges* of values associated with system variables during symbolic execution may not be sufficiently

```
(1)   declare
(2)       d : Integer
(3)   begin
(4)       if 0 ⩽ c then   -- set d equal to magnitude of c
(5)           d := c
(6)       else
(7)           d := −c
(8)       end if ;
(9)       if 0 ⩽ b then   -- set a equal to magnitude of b
(10)          a := b
(11)      else
(12)          a := −b
(13)      end if ;
(14)      while d ⩽ a loop   -- repeatedly subtract d from a
(15)          a := a − d
(16)      end loop
(17)  end ;
(18)  if b < 0 then   -- retain sign of b
(19)      a := −a
(20)  end if
```

Figure 1: Program to implement 'a **:=** b **rem** c'.

discerning to accurately identify the actual paths that will be followed [16]. Formal verification techniques can be used to identify dead paths through backward substitution or by solving fixed point equations [11, 8, 17]. Most of these algorithms work on high-level language code, although dead path elimination algorithms for assembler programs have also been proposed [18] and implemented [11].

Our goal is not to develop another dead path detection algorithm, but to define a semantic characterisation of the dead paths themselves. Such an outcome provides a formal foundation for verifying the correctness of existing and proposed dead path algorithms.

4 Control flow paths

As a simple motivational example, consider the program in Figure 1. For concreteness, we use an Ada-like syntax (although we reserve semi-colons for sequential composition, rather than as statement terminators). The program aims to use simple arithmetic operators and comparisons to implement the assignment 'a **:=** b **rem** c', where a, b and c are integers and '**rem**' is the integer remainder operator. Variables b and c are inputs to the code segment and a is the output.

The block statement starting at line 1 declares a local variable d, which will be used to hold the magnitude of the divisor c, because variable c may not be changed by this program. The scope of new variable d is entered at line 3 and left at line 17. Within the block, the conditional statement at line 4 sets d equal to the magnitude of c, and that at line 9 sets a equal to the magnitude of b. This is done so that both operands are non-negative. The iterative construct at line 14 then calculates the integer remainder by repeated subtraction of the divisor from the dividend. The loop

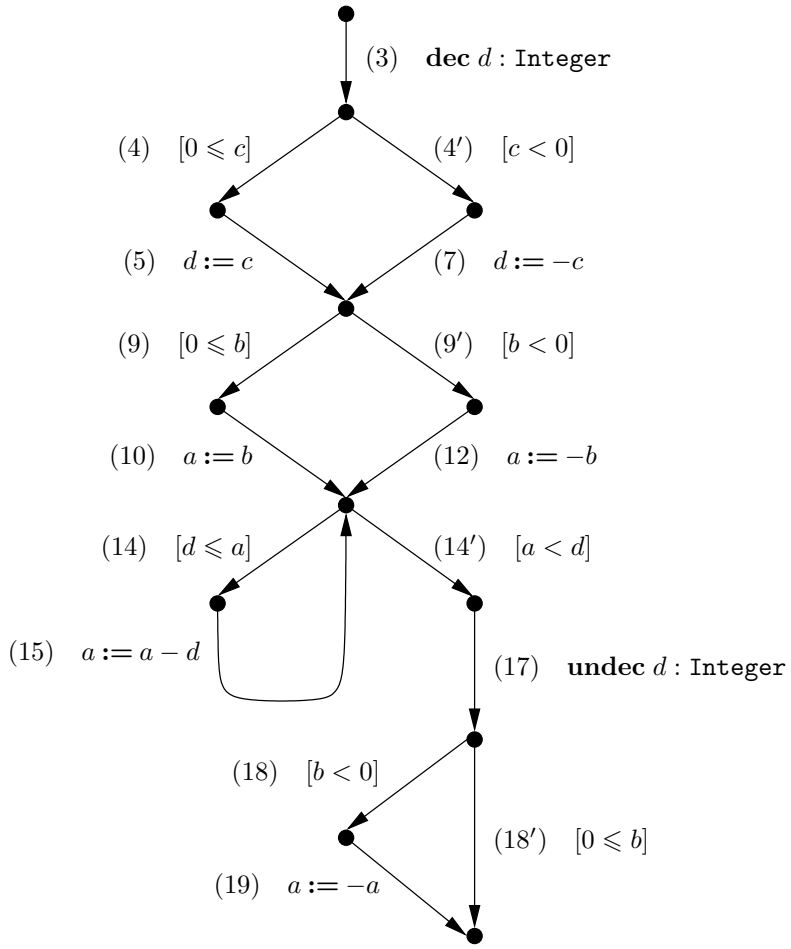

Figure 2: Flow graph for the program in Figure 1.

invariant is '$0 \leqslant d \wedge 0 \leqslant a \wedge |b \,\mathbf{rem}\, c| = a \,\mathbf{rem}\, d \wedge d = |c|$', where $|\cdot|$ denotes magnitude. Finally, another conditional statement at line 18 ensures that the final value of a has the same sign as the dividend b. This matches the semantics of Ada's **rem** operator [19, §4.5.5], for which the sign of the result is independent of the sign of the divisor. Although trivial, even this small program contains a number of dead paths. Some are paths that are never intended to be followed, and others are due to a programming oversight (revealed below).

Control-flow paths through program code are best illustrated graphically. However traditional program flowcharts, with their diamond-shaped symbol for choices, are inappropriate for our purposes because this single node does not indicate whether the boolean expression was true or false [17]. Instead we use a graphical notation in which control flow is shown as a directed graph with arcs labelled by programming language statements. (The T-graph notation used by Puschner and Schedl [9] for timing analysis is similar, except that they label arcs with execution times.) To clearly describe control flow through compound statements, we use some additional language primitives (formally defined in Section 6). Primitive statements '**dec**' and '**undec**' denote entry to and exit from the scope of a variable declaration, respectively. Primitive statement '$[G]$' denotes evaluation of the boolean expression G to 'true' in an **if** or **while** statement.

Figure 2 shows the control-flow graph associated with the remainder program. Each arc is labelled with the associated statement; line numbers refer to the program in Figure 1, with primed numbers

on **if** and **while** conditions indicating that the boolean expression appearing in the program text evaluated to 'false'. Alternative routes through conditional and iterative statements are shown by diverging paths, each beginning with an arc labelled by the condition that must be true for this path to be followed.

To find a potential control-flow path through the program we merely need to start at any node and follow the arcs, recording each label passed along the way. All the semantic information needed to analyse the path is contained within the accumulated labels. For instance, the path below can be found by following arcs and accumulating the sequence of labels, starting from the second left-hand alternative in Figure 2, bypassing the loop, and following the third left-hand alternative.

Path 1

(9) $[0 \leqslant b]$

(10) $a := b$

(14′) $[a < d]$

(17) **undec** d : `Integer`

(18) $[b < 0]$

(19) $a := -a$

Notice how the boolean expressions in square brackets explicitly document which conditions must have been true at various points for this path to have been followed.

Although Path 1 is statically a valid control flow path of the program, it is dynamically 'dead' because it can *never* be followed at run time, regardless of the initial values of b and c. For this path to be taken it is necessary for condition $0 \leqslant b$ to hold initially, but this is followed later by contradictory condition $b < 0$, even though none of the intervening statements changes variable b. Our goal below is to show that the semantics of the statements in the path formally identifies when such paths are dead.

5 Dead paths

Ultimately, it is the semantics of the individual program statements comprising a path that determines whether it is dead or not. In this section we explore the relationship between program semantics and dead paths in depth.

Recall that Dijkstra introduced both weakest and weakest liberal preconditions as a way of characterising the semantics of programs [3]. Given some statement S and postcondition predicate R, then predicate wp.S.R is the *weakest precondition* of S with respect to R. It is a predicate characterising those initial states from which statement S is guaranteed to terminate in a state satisfying predicate R [3, p. 16]. The full stops denote left-associative function application; 'wp.S.R' is equivalent to '(wp(S))(R)' [20, p. 128]. Function wp.S is a *predicate transformer*, i.e., a function from predicates (postconditions) to predicates (preconditions). The *weakest liberal precondition* wlp.S.R is the weaker constraint characterising those initial states from which S will achieve R provided that S terminates [3, p. 21]. However there is no guarantee that statement S will terminate from a state satisfying wlp.S.R. Given two predicates P and Q with free variables v, let '$P \Rrightarrow Q$' mean that P implies Q for all values of these variables, i.e., $(\forall v \bullet P \Rightarrow Q)$ [10, p. 23]. Similarly for predicate equivalence, '$P \equiv Q$'. Then it is always the case that wp.S.$R \Rrightarrow$ wlp.S.R for any statement S and postcondition R.

Our aim is to represent a path as a statement and reuse Dijkstra's definitions to characterise dead control-flow paths. We must therefore decide what forms of statement may appear in a path, and which semantic definition is suitable for characterising 'dead' paths.

Originally, Dijkstra required that all statements be *non-miraculous* [3, p. 18]. That is, there must be no initial state from which a statement can achieve the impossible postcondition 'false'. Although this is sufficient for conventional imperative programming language statements, many authors subsequently argued that this restriction should be relaxed so that component parts of compound statements can be considered in isolation [21]. This capability is needed for our dead path analysis. For example, Path 1 above contains bracketed expressions '[G]' representing evalua-

tion of the conditions that guard entry to **if** and **while** statement alternatives [22], but not the entire compound statements themselves. When analysed in isolation, such a *guard* $[G]$ is considered miraculous in those states where predicate G is false.

A path, therefore, is a statement S constructed from conventional programming language statements, such as assignments, and 'partial' statements [21], such as guards. (Normally our paths do not contain whole **if** and **while** statements, but these can be accommodated as explained in Section 8.) To define what it means for such a path to be classed as 'dead', we note that there are two ways in which a path may fail to execute to completion: either it contains 'unfollowable' statements, or it never terminates.

For all practical purposes it is satisfactory to consider either of these situations as constituting a dead path. During program testing, for instance, it is impossible to generate a test case for an unfollowable path, and we cannot test the outcome of a non-terminating one. In worst-case execution time analysis, it is meaningless to refer to the execution time of an unfollowable path, and the execution time of a non-terminating path is always infinity, which never satisfies any reasonable execution time requirement. When extracting program timing constraints, there are no meaningful constraints associated with unfollowable paths or paths that never terminate. Finally, when searching for simple coding errors, such as ineffective assignments, there is no need to consider paths that are never followed, and code following a non-terminating path is unreachable, so such analysis is useful for code preceded by terminating paths only.

We can now use Dijkstra's semantics to formally characterise these two situations. Firstly, if the path contains a statement, or sequence of statement, that cannot be followed at run time, then the whole path is miraculous. For instance, Path 1 above is miraculous because it contains contradictory guards. Given a path S, the weakest precondition wp.S.false characterises the set of initial states from which execution of S is impossible, i.e., from which S is miraculous [3, p. 18]. If this precondition set contains all states, i.e., wp.S.false $\equiv$ true, then there are no initial states from which execution of S is possible, so we consider it dead.

Secondly, if the path cannot execute to completion because it contains an infinite loop, or some equivalent construct, then the path is non-terminating. For *non-miraculous* statements, Dijkstra identified those initial states from which a statement fails to terminate by the weakest *liberal* precondition wlp.S.false [3, p. 21]. To extend this to allow miraculous statements, we use the fact that wp.S.false $\Rightarrow$ wlp.S.false to note that wlp.S.false also includes those initial states from which path S is miraculous. Hence, wlp.S.false is sufficient to characterise all states from which path S is dead.

Definition 1 (Dead initial states) *An initial state from which statement (or path) S is* dead *is one characterised by the following weakest liberal precondition.*

$$\text{wlp}.S.\text{false}$$

Using this definition we can say that a statement, or control-flow path, is dead if it is miraculous or fails to terminate from *any* initial state.

Definition 2 (Dead statement) *A programming language statement (or path) S is* dead *if and only if the following property holds.*

$$\text{wlp}.S.\text{false} \quad \equiv \quad \text{true}$$

6 Semantics

In this section we give weakest liberal precondition definitions for the statements that may appear in a path, suitable for use in dead path analysis.

The weakest liberal precondition semantics for typical programming language constructs are shown below. (Iteration is considered at the end of this section.) Let S be a statement in our programming language, v a variable, T a type, E an expression, B a boolean-valued expression, and R a predicate. Let $R[E/v]$ denote predicate R with all free occurrences of 'v' replaced by 'E'. Let $\forall D \bullet P$ denote univer-

sal quantification of predicate P for all values of declaration D. Let predicate 'def.X' characterise those states in which expression X has a well-defined value, i.e., where X does not involve ill-defined operations such as division by zero, out-of-bounds array indices, or other operations that may not terminate [3, p. 28].

Definition 3 (Simple statements)

$$\text{wlp.}\mathbf{null}.R \quad \equiv \quad R$$

$$\text{wlp.}(v := E).R \quad \equiv \quad \text{def.}E \Rightarrow R[E/v]$$

$$\text{wlp.}(S_1 \,;\, S_2).R \quad \equiv \quad \text{wlp.}S_1.(\text{wlp.}S_2.R)$$

$$\begin{aligned}&\text{wlp.}(\mathbf{if}\, B\, \mathbf{then}\, S\, \mathbf{end\ if}).R\\ \equiv\ &\text{def.}B \Rightarrow ((B \Rightarrow \text{wlp.}S.R) \wedge\\ &\qquad (\neg B \Rightarrow R))\end{aligned}$$

$$\begin{aligned}&\text{wlp.}(\mathbf{if}\, B\, \mathbf{then}\, S_1\, \mathbf{else}\, S_2\, \mathbf{end\ if}).R\\ \equiv\ &\text{def.}B \Rightarrow ((B \Rightarrow \text{wlp.}S_1.R) \wedge\\ &\qquad (\neg B \Rightarrow \text{wlp.}S_2.R))\end{aligned}$$

$$\begin{aligned}&\text{wlp.}(\mathbf{declare}\ v : T\ \mathbf{begin}\, S\, \mathbf{end}).R\\ \equiv\ &\text{'provided '}v\text{' is not free in }R\text{'}\\ &\forall v : T \bullet \text{wlp.}S.R\end{aligned}$$

These weakest liberal precondition definitions differ from their weakest precondition equivalents only in their treatment of undefined expressions. For instance, the weakest precondition definition for assignment requires that expression E is defined, i.e., $\text{wp.}(v := E).R \equiv \text{def.}E \wedge R[E/v]$.

The definition for '**declare**' blocks above does not allow references to variable v in the postcondition R. However, such references may be necessary if a global variable 'v' is declared in the surrounding scope. This can be accommodated, and the proviso avoided, by appropriate renaming of the locally scoped variable [10, p. 185].

Combined with Definition 1 from Section 5, we can now determine the conditions under which these individual statements are dead. For instance, we can show that an assignment statement is dead when its expression is undefined.

$$\begin{aligned}&\text{wlp.}(v := E).\text{false}\\ \equiv\ &\text{'by Definition 3'}\\ &\text{def.}E \Rightarrow \text{false}\\ \equiv\ &\neg\text{def.}E\end{aligned}$$

We also allow our language to include *assertions* with which the programmer can express properties of the system state that are believed to hold at certain points in the program [23, p. 4]. (Morgan uses the term *assumption* [10, p. 65].) A typical application is to express *domain constraints* concerning the initial values of variables. If predicate A holds when an assertion $\{A\}$ is reached then the statement has no effect, but if A does not hold then the assertion's behaviour cannot be guaranteed, and it may never terminate [10, pp. 65–6]. We also require that the assertion expression is well defined.

Definition 4 (Assertions)

$$\text{wlp.}\{A\}.R \quad \equiv \quad \text{def.}A \Rightarrow (A \Rightarrow R)$$

The weakest liberal precondition of an assertion thus requires R to hold only if A already holds. The corresponding weakest precondition definition requires that predicate A must be true, i.e., $\text{wp.}\{A\}.R \equiv \text{def.}A \wedge A \wedge R$.

Definition 1 then tells us that an assertion is considered dead if its predicate is ill-defined or untrue.

$$\begin{aligned}&\text{wlp.}\{A\}.\text{false}\\ \equiv\ &\text{'by Definition 4'}\\ &\text{def.}A \Rightarrow (A \Rightarrow \text{false})\\ \equiv\ &\neg\text{def.}A \vee \neg A\end{aligned}$$

As noted above, representing control-flow paths presents us with the difficulty that paths traverse distinct parts of compound programming language statements. For instance, Path 1 in Section 4 includes evaluation of the boolean condition and execution of the first alternative of the conditional statement at line 9, but not execution of its second alternative. Also, since the path starts on line 9 and passes the keyword on line 17 which marks the end of the declaration block, this path exits (but did not enter) the scope of local variable d.

Our approach, therefore, is to define separate semantics for distinct components of compound programming language statements, so that each component can be considered in isolation [21]. To support analysis of the program in Figure 1, we need to

be able to separately reason about evaluation of the boolean expressions guarding conditional and iterative statements, and entry to and exit from the variable block.

The first of these requirements can be satisfied by adding *guards* [22], to our language. (Morgan uses the term *coercion* [10, p. 67]; Back and von Wright use the term *assumption* [23, p. 5].) Guards are a requirement to make some predicate true at this point in the program. A guard $[G]$ is satisfied trivially if G is already true, otherwise it is a miraculous statement because it makes the predicate true but without changing the program state [10, p. 67]. Its weakest liberal precondition is as follows.

Definition 5 (Guards)

$$\text{wlp.}[G].R \quad \equiv \quad \text{def.}G \Rightarrow (G \Rightarrow R)$$

Although this weakest liberal precondition is the same as Definition 4 above, the weakest precondition definition of coercions differs from that of assertions: $\text{wp.}[G].R \equiv \text{def.}G \wedge (G \Rightarrow R)$.

In defining control-flow paths, a guard is a suitable construct for representing evaluation of the boolean expressions on conditional and iterative statements because it records the fact that for the particular path to be followed, the corresponding condition must have been true. In Figure 1, for instance, the condition on line 4 must be true to follow a path containing the assignment statement on line 5, so guard $[0 \leqslant c]$ can be added to the path at the point where the expression is evaluated. Similarly, to follow a path containing the assignment on line 7, the condition on line 4 must have been false, so guard $[c < 0]$ can be added to the path.

To allow for paths that enter or exit the scope of a declaration, we need separate primitives for these two actions [23, §5.6]. When analysing isolated paths, rather than whole statements, it is not always the case that these primitives appear in pairs. For a variable name v and type T, let primitive statements '**dec** $v : T$' and '**undec** $v : T$' represent entry to and exit from the scope of this variable declaration, respectively. The **dec** statement can achieve postcondition R provided that R holds for any value of the new variable v within its type T. (The statement is miraculous if T is the empty set.) We assume the language implementation ensures that freshly declared variables are initialised with an arbitrary value from their type domain.

Definition 6 (Variable allocation)

$$\text{wlp.}(\textbf{dec}\ v : T).R \quad \equiv \quad \forall v : T \bullet R$$

After an **undec** statement has been executed, variable v can no longer be accessed, so its semantics is meaningful only if postcondition R does not refer to v [23, p. 102].

Definition 7 (Variable deallocation)

$$\begin{array}{ll} & \text{wlp.}(\textbf{undec}\ v : T).R \\ \equiv & \text{'provided '}v\text{' is not free in }R\text{'} \\ & R \end{array}$$

Again, we have excluded the possibility that a global variable v may be referenced in predicate R. Appropriate renaming must be introduced if nested declarations using the same name are desired [10, p. 185].

Finally, we give a semantics for iterative statements that takes advantage of the guard primitive introduced above [20, p. 185]. For an iterative statement with boolean condition B and body S, note that each iteration involves first evaluating expression B and then executing statement S. Thus each iteration of the loop can be represented by the sequence of statements '$[B]\ ;\ S$'. Furthermore, with respect to some desired postcondition R, each iteration will either leave boolean condition B true, in which case the iterative construct will loop again, or should make postcondition R true, in which case the iterative construct may terminate. Let $\mathbb{N}$ be the set of natural numbers. For a predicate-transformer function $\text{wlp.}S$ and natural number n, let $(\text{wlp.}S)^n$ denote functional composition of n copies of $\text{wlp.}S$, with $(\text{wlp.}S)^0$ the identity function on predicates.

Definition 8 (Iteration)

$$\begin{array}{ll} & \text{wlp.}(\textbf{while}\ B\ \textbf{loop}\ S\ \textbf{end loop}).R \\ \equiv & \forall n : \mathbb{N} \bullet \\ & \quad (\text{wlp.}([B]\ ;\ S))^n.(\text{def.}B \Rightarrow (B \vee R)) \end{array}$$

In other words, we require that after any number of executions of guard $[B]$ followed by statement S then, provided the guard expression is well defined, either B must still be true (and hence iteration continues) or R holds (and if B is false the loop will terminate, satisfying R). See Appendix B for the derivation of this expression.

7 Path analyses

We can now use the definitions from Sections 5 and 6 to perform dead path analysis on our example program. Firstly, we determine whether Path 1 from Section 4 is dead or not. Starting with our target postcondition R equal to 'false' we work backwards up the path to compute the path's weakest liberal precondition.

$$\begin{array}{lcl} R_0 & \equiv & \text{false} \\ R_1 & \equiv & \text{wlp.}(a := -a).R_0 \\ & \equiv & \text{def.}(-a) \Rightarrow \text{false}[-a/a] \\ & \equiv & \text{true} \Rightarrow \text{false} \\ & \equiv & \text{false} \\ R_2 & \equiv & \text{wlp.}[b < 0].R_1 \\ & \equiv & \text{def.}(b < 0) \Rightarrow (b < 0 \Rightarrow \text{false}) \\ & \equiv & \text{true} \Rightarrow (b < 0 \Rightarrow \text{false}) \\ & \equiv & 0 \leqslant b \\ R_3 & \equiv & \text{wlp.}(\textbf{undec } d : \texttt{Integer}).R_2 \\ & \equiv & 0 \leqslant b \\ R_4 & \equiv & \text{wlp.}[a < d].R_3 \\ & \equiv & a < d \Rightarrow 0 \leqslant b \\ R_5 & \equiv & \text{wlp.}(a := b).R_4 \\ & \equiv & b < d \Rightarrow 0 \leqslant b \\ R_6 & \equiv & \text{wlp.}[0 \leqslant b].R_5 \\ & \equiv & 0 \leqslant b \Rightarrow (b < d \Rightarrow 0 \leqslant b) \\ & \equiv & \text{true} \end{array}$$

Thus, according to Definition 2, we have formally proven that Path 1 is indeed dead. (We assume that the type of variables a, b and c is `Integer`. From this, we can conclude that def.$X \equiv$ true for every expression X in Figure 1. For instance, in calculating predicate R_2, we needed to know that expression '$b < 0$' is defined. This will be so provided that the program is well typed, and we usually omit the 'def' tests below.)

Proving that Path 1 is dead did not require us to extend the path all the way back to the beginning of the program. If we had done so, however, the result would have been the same, thanks to the following theorem which tells us that if any subpath of a path is dead, then the whole path is also dead (see Appendix A for its proof).

Theorem 1 (Dead subpaths) *Let S_1, S_2 and S_3 be arbitrary program statements. If statement S_2 is dead, then sequence $S_1 ; S_2 ; S_3$ is also dead. Formally:*

$$\begin{array}{l} \text{if wlp.}S_2\text{.false} \equiv \text{true} \\ \text{then wlp.}(S_1 ; S_2 ; S_3)\text{.false} \equiv \text{true.} \end{array}$$

In practice, therefore, analysis can stop as soon as predicate 'true' is reached. Statements S_1 or S_3 may be '**null**', so this theorem also suffices for the cases where S_2 begins or ends a path.

As well as proving general properties, we can also introduce additional assertions to a path to test particular situations of interest. For instance, assume that we know that the remainder program will only ever be used with an initial value of dividend b greater than zero. This additional domain constraint can be added as an assertion to the start of a path which begins by entering the scope of variable d, follows the first alternative of the conditional statement at line 4, and then follows the second alternative of the statement at line 9.

Path 2

$$\begin{array}{ll} & \{0 < b\} \quad \texttt{--} \text{ new input constraint} \\ (3) & \textbf{dec } d : \texttt{Integer} \\ (4) & [0 \leqslant c] \\ (5) & d := c \\ (9') & [b < 0] \\ (12) & a := -b \end{array}$$

Analysis of Path 2 reveals that the new assertion makes the assignment statement at line 12 unreachable.

$$\begin{array}{lcl} R_0 & \equiv & \text{false} \\ R_1 & \equiv & \text{wlp.}(a := -b).R_0 \\ & \equiv & \text{false} \end{array}$$

$$
\begin{array}{lcl}
R_2 & \equiv & \text{wlp}.[b < 0].R_1 \\
& \equiv & 0 \leqslant b \\
R_3 & \equiv & \text{wlp}.(d := c).R_2 \\
& \equiv & 0 \leqslant b \\
R_4 & \equiv & \text{wlp}.[0 \leqslant c].R_3 \\
& \equiv & 0 \leqslant c \Rightarrow 0 \leqslant b \\
R_5 & \equiv & \text{wlp}.(\textbf{dec}\ d : \texttt{Integer}).R_4 \\
& \equiv & \forall d : \texttt{Integer} \bullet 0 \leqslant c \Rightarrow 0 \leqslant b \\
& \equiv & 0 \leqslant c \Rightarrow 0 \leqslant b \\
R_6 & \equiv & \text{wlp}.\{0 < b\}.R_5 \\
& \equiv & \text{`since `}0 < b\text{' is well defined'} \\
& & 0 < b \Rightarrow (0 \leqslant c \Rightarrow 0 \leqslant b) \\
& \equiv & \text{true}
\end{array}
$$

The ability to treat parts of compound statements separately also allows us to 'cut' iterative statements into separate control-flow paths for loop entry, loop exit, and one or more iterations. For instance, having entered the iterative statement at line 14 in Figure 1, consider the question of whether it is possible to exit with variable a negative. The following path performs one iteration of the loop and then exits, and we append the condition of interest as an assertion.

Path 3

$$
\begin{array}{ll}
(14) & [d \leqslant a] \\
(15) & a := a - d \\
(14') & [a < d] \\
& \{a < 0\} \quad \texttt{--} \text{ desired exit state}
\end{array}
$$

Again, our analysis proves that this path is dead.

$$
\begin{array}{lcl}
R_0 & \equiv & \text{false} \\
R_1 & \equiv & \text{wlp}.\{a < 0\}.R_0 \\
& \equiv & 0 \leqslant a \\
R_2 & \equiv & \text{wlp}.[a < d].R_0 \\
& \equiv & a < d \Rightarrow 0 \leqslant a \\
R_3 & \equiv & \text{wlp}.(a := a - d).R_2 \\
& \equiv & a - d < d \Rightarrow 0 \leqslant a - d \\
& \equiv & a < 2d \Rightarrow d \leqslant a \\
R_4 & \equiv & \text{wlp}.[d \leqslant a].R_3 \\
& \equiv & d \leqslant a \Rightarrow (a < 2d \Rightarrow d \leqslant a) \\
& \equiv & \text{true}
\end{array}
$$

This is an interesting outcome because it is not obvious from inspection of Path 3 that a cannot be negative at the end—the loop exit condition is merely that a is less than d, but the path contains no explicit information about the value of d at all. The key is the assignment statement at line 15. Since it subtracts d from a, a negative value of d would cause a to *increase* in value. However, the guards in Path 3 require the state to change from one where a is no less than d to one where a is less than d. It is thus implicit that d must be positive for this path to followed. The knowledge that a is initially at least as great as d then allows us to conclude that Path 3 will leave a non-negative.

Furthermore, since Path 3 is the suffix of any path that exits the loop after one or more iterations, we can use Theorem 1 to conclude that *any* path which enters the loop cannot subsequently exit with a negative. (Of course, had it been documented by the programmer, this result could have been seen directly in the loop invariant mentioned in Section 4.)

8 Compound statements

So far we have defined paths to be program fragments constructed from 'primitive' statements. However, the granularity of a path can be coarsened to include whole compound statements, when required. This is helpful when looking for paths that are dead due to infinite iteration. Statements that loop endlessly make any subsequent subpaths unreachable.

To illustrate this, we use Definition 1 to answer the question of whether there are any input values for b and c which will make the iterative statement at line 14 dead in the context of our remainder program. This capability is valuable since non-termination cannot be checked via traditional testing methods. To do so, we need to include the whole loop statement in the path and, for illustration, treat the preceding conditional statements similarly.

Path 4

$$
(4\text{–}8) \quad \textbf{if}\ 0 \leqslant c\ \textbf{then}\ \cdots\ \textbf{end if}
$$

(9–13) **if** $0 \leqslant b$ **then** $\cdots$ **end if**

(14–16) **while** $d \leqslant a$ **loop** $\cdots$ **end loop**

As before, we work backwards along the path, using the semantic definitions of the statements encountered. Since we are dealing with compound statements, however, the predicates to be manipulated are more complex than those encountered so far. As we are interested only in the situation where the postcondition R is 'false', we therefore begin by noting the following special case of loop semantics for this situation (see Appendix C for its proof).

Theorem 2 (Dead iteration)

$$\begin{array}{ll} & \text{wlp.}(\textbf{while}\ B\ \textbf{loop}\ S\ \textbf{end loop}).\text{false} \\ \equiv & \forall n : \mathbb{N} \bullet \\ & \quad (\text{wlp.}([\text{def}.B]\ ;\ S))^n.(\text{def}.B \Rightarrow B) \end{array}$$

Furthermore, if it is known that expression B is well-defined, as is the case for all expressions in our example program, we obtain a further simplification.

Corollary 3 (Defined loop guards)

$$\begin{array}{ll} & \text{wlp.}(\textbf{while}\ B\ \textbf{loop}\ S\ \textbf{end loop}).\text{false} \\ \equiv & \text{'provided def}.B \equiv \text{true'} \\ & \forall n : \mathbb{N} \bullet (\text{wlp}.S)^n.B \end{array}$$

The following calculation determines the weakest liberal precondition which makes Path 4 dead. Let $\mathbb{N}_1$ be the set of positive natural numbers (excluding zero). For a predicate R, variable v, expression E and natural number n, let $R[E/v]^n$ denote n repeated substitutions of 'E' for 'v' in R.

$$\begin{array}{lll} R_0 & \equiv & \text{false} \\ R_1 & \equiv & \text{wlp.}(\textbf{while}\ d \leqslant a\ \textbf{loop} \\ & & \qquad a := a - d \\ & & \quad \textbf{end loop}).R_0 \\ & \equiv & \text{'by Corollary 3'} \\ & & \forall n : \mathbb{N} \bullet \\ & & \quad (\text{wlp.}(a := a - d))^n.(d \leqslant a) \\ & \equiv & \forall n : \mathbb{N} \bullet (d \leqslant a)[a - d/a]^n \\ & \equiv & \forall n : \mathbb{N} \bullet d \leqslant a - n * d \\ & \equiv & \forall n : \mathbb{N}_1 \bullet n * d \leqslant a \\ & \equiv & d \leqslant a \wedge d \leqslant 0 \end{array}$$

$$\begin{array}{lll} R_2 & \equiv & \text{wlp.}(\textbf{if}\ 0 \leqslant b\ \textbf{then} \\ & & \qquad a := b \\ & & \quad \textbf{else} \\ & & \qquad a := -b \\ & & \quad \textbf{end if}).R_1 \\ & \equiv & \text{def.}(0 \leqslant b) \Rightarrow \\ & & \quad ((0 \leqslant b \Rightarrow \text{wlp.}(a := b).R_1) \wedge \\ & & \quad (b < 0 \Rightarrow \text{wlp.}(a := -b).R_1)) \\ & \equiv & (0 \leqslant b \Rightarrow (d \leqslant b \wedge d \leqslant 0)) \wedge \\ & & (b < 0 \Rightarrow (d \leqslant -b \wedge d \leqslant 0)) \\ & \equiv & d \leqslant 0 \wedge \\ & & (0 \leqslant b \Rightarrow d \leqslant b) \wedge \\ & & (b < 0 \Rightarrow d \leqslant -b) \\ & \equiv & d \leqslant 0 \wedge d \leqslant |b| \\ & \equiv & d \leqslant 0 \\ R_3 & \equiv & \text{wlp.}(\textbf{if}\ 0 \leqslant c\ \textbf{then} \\ & & \qquad d := c \\ & & \quad \textbf{else} \\ & & \qquad d := -c \\ & & \quad \textbf{end if}).R_2 \\ & \equiv & (0 \leqslant c \Rightarrow \text{wlp.}(d := c).R_2) \wedge \\ & & (c < 0 \Rightarrow \text{wlp.}(d := -c).R_2) \\ & \equiv & (0 \leqslant c \Rightarrow c \leqslant 0) \wedge \\ & & (c < 0 \Rightarrow -c \leqslant 0) \\ & \equiv & c \leqslant 0 \wedge 0 \leqslant c \\ & \equiv & c = 0 \end{array}$$

Our formal analysis has thus revealed the programmer's oversight in the remainder program in Figure 1. It will loop endlessly, and hence the program is 'dead', if the divisor c is zero. This weakness of the program should have been documented by an initial assertion stating the precondition that input c is expected to be non-zero.

9 Approximations

In general dead path analysis is an incomputable problem because it includes determining whether or not a loop will terminate (the halting problem). In practice, therefore, dead path algorithms can produce approximate results only.

To show the correctness of such a dead path analysis algorithm, we must therefore show that it gives a satisfactory approximation to the calculation required by Definition 2. The type of approximation

that is acceptable depends on the application. In most situations we require that the approximation should correctly identify as many dead paths as possible, but not mistakenly classify any followable path as dead. In other words, if a dead path algorithm identifies a path S as dead then it *must* be the case that wlp.S.false $\equiv$ true. However, there *may* be a path U such that wlp.U.false $\equiv$ true, but the algorithm fails to recognise that U is dead. This means that the approximation may identify only a *subset* of the dead paths as dead. Nevertheless this is acceptable in most situations. During worst-case execution time analysis, for instance, the failure to exclude some dead paths from the analysis may produce unnecessarily pessimistic timing estimates, but this is safer than producing unrealistically optimistic values. Similarly, when searching for simple coding errors, it is better to waste some effort analysing a dead path, than to miss an error in a followable one. This approach to approximations is assumed in the discussion below.

However, the situation of program testing is an exception. It is futile to try testing dead paths, so a satisfactory approximation here is instead one which identifies a *superset* of the dead paths as dead. It is adequate during testing to use an approximation that identifies some, but hopefully not too many, followable paths as dead, and thus exclude them from testing, provided that we still get sufficient code coverage.

In practice, proving the correctness of a dead path analysis algorithm could be done in either of two ways. One approach is to show that the algorithm has the same effect as replacing the path S with a (pessimistic) approximation path S' such that the following property holds for any postcondition R.

$$\text{wlp.}S'.R \quad \Rrightarrow \quad \text{wlp.}S.R$$

If path S' is shown to be dead then S is also dead, so a subset of the dead paths will be correctly identified as dead. For instance, some specification languages allow a generalised form of assignment statement $v :\in X$ where v is a variable and X is a set of values of the same type as v [24]. Such a statement selects a particular value x from set X and assigns it to variable v, but we cannot predict which. If we can show that a path containing the assignment approximation $v :\in X$ is dead, then we know that the path with specific assignment $v := x$, where $x \in X$, is also dead. The semantics for generalised assignments is as follows.

Definition 9 (General assignment)

$$\begin{array}{ll} & \text{wlp.}(v :\in X).R \\ \equiv & \text{`provided `}w\text{' is not free in } X \\ & \text{or } R \text{ and } v \text{ is of type } T\text{'} \\ & \text{def.}X \Rightarrow \\ & \quad (\forall w : T \bullet w \in X \Rightarrow R[w/v]) \end{array}$$

As an example, consider the following alternative to statement 15 in the remainder program.

$$(15') \quad a :\in \{n : \mathbb{N}_1 \mid n * d \leqslant a \bullet a - n * d\}$$

The set comprehension on the right-hand side returns all values $a - n * d$ where n is a positive natural number such that $n * d \leqslant a$. In other words, rather than just subtracting d from a, statement $15'$ subtracts some whole multiple of d, not exceeding a. (Statement $15'$ is a specification of, or a less 'refined' [10] version of, statement 15.) If our programming language allowed us to use this statement in place of statement 15, the remainder program would have exactly the same effect. The only difference is that it may take fewer iterations of the **while** loop to complete the calculation.

In Section 7 we proved that Path 3 containing statement 15 was dead. Now consider the same path, but with our generalised assignment.

Path 5

$$\begin{array}{ll} (14) & [d \leqslant a] \\ (15') & a :\in \{n : \mathbb{N}_1 \mid n * d \leqslant a \bullet a - n * d\} \\ (14') & [a < d] \\ & \{a < 0\} \end{array}$$

Using Definition 9, we can prove that Path 5 is dead. Let $\mathbb{Z}$ be the set of integers.

$$\begin{array}{lll} R_0 & \equiv & \text{false} \\ R_1 & \equiv & 0 \leqslant a \end{array}$$

$$
\begin{array}{lcl}
R_2 & \equiv & a < d \Rightarrow 0 \leqslant a \\
R_3 & \equiv & \mathrm{wlp}.(a :\in \{n : \mathbb{N}_1 \mid n * d \leqslant a \bullet \\
 & & \qquad a - n * d\}).R_2 \\
 & \equiv & \text{'since the assignment expression is well defined'} \\
 & & \forall e : \mathbb{Z} \bullet \\
 & & \quad e \in \{n : \mathbb{N}_1 \mid n * d \leqslant a \bullet \\
 & & \qquad a - n * d\} \Rightarrow \\
 & & \quad (e < d \Rightarrow 0 \leqslant e) \\
R_4 & \equiv & \mathrm{wlp}.[d \leqslant a].R_3 \\
 & \equiv & d \leqslant a \Rightarrow \\
 & & \quad (\forall e : \mathbb{Z} \bullet \\
 & & \qquad e \in \{n : \mathbb{N}_1 \mid n * d \leqslant a \bullet \\
 & & \qquad\quad a - n * d\} \Rightarrow \\
 & & \qquad (e < d \Rightarrow 0 \leqslant e)) \\
 & \equiv & d \leqslant a \Rightarrow \\
 & & \quad (\forall e : \mathbb{Z}; n : \mathbb{N}_1 \bullet \\
 & & \qquad (n * d \leqslant a \wedge \\
 & & \qquad\ e = a - n * d) \Rightarrow \\
 & & \qquad (e < d \Rightarrow 0 \leqslant e)) \\
 & \equiv & d \leqslant a \Rightarrow \\
 & & \quad (\forall n : \mathbb{N}_1 \bullet \\
 & & \qquad n * d \leqslant a \Rightarrow \\
 & & \qquad (a - n * d < d \Rightarrow \\
 & & \qquad\quad 0 \leqslant a - n * d)) \\
 & \equiv & d \leqslant a \Rightarrow \\
 & & \quad (\forall n : \mathbb{N}_1 \bullet \\
 & & \qquad n * d \leqslant a \Rightarrow \\
 & & \qquad (a - n * d < d \Rightarrow \\
 & & \qquad\quad n * d \leqslant a)) \\
 & \equiv & \mathrm{true}
\end{array}
$$

Since Path 5 is an approximation of Path 3, this proof that Path 5 is dead is also sufficient to conclude that Path 3 is dead.

In particular, the ability of generalised assignments to include an arbitrary set of values on their right-hand side makes them a suitable basis for checking the correctness of path analysis algorithms that associate ranges or sets [5] of values with variables as a means of symbolic path execution.

An alternative, but equivalent, approach to showing the correctness of dead path algorithms is to show that the algorithm implements an 'approximate' weakest liberal precondition function 'awlp' which, for any path S and postcondition R, has the following property.

$$\mathrm{awlp}.S.R \quad \Rrightarrow \quad \mathrm{wlp}.S.R$$

Thus, if $\mathrm{awlp}.S.\mathrm{false} \equiv \mathrm{true}$ holds then so does $\mathrm{wlp}.S.\mathrm{false} \equiv \mathrm{true}$, and it is safe to conclude that path S is dead.

For example, consider a **while** statement guarded by boolean expression B and with a body S, where S updates the set of variables x. Recall that a predicate I is an invariant of such a loop if, when the guard B is true, execution of the loop body S maintains I.

$$I \quad \Rrightarrow \quad \mathrm{wlp}.([B]\,;S).I$$

Via Definitions 5 and 3, this can be reexpressed as follows.

$$I \wedge \mathrm{def}.B \wedge B \quad \Rrightarrow \quad \mathrm{wlp}.S.I$$

For a loop with an invariant I that holds when the loop begins, then at each iteration the invariant will be reestablished and either the (well-defined) guard B will still be true or the loop's postcondition R will be established. From this intuition, we can define an approximate weakest liberal precondition for loops that offers a way of checking whether the loop is dead, but that is easier to calculate than using Theorem 2 above. (Formally, the following definition can be derived by induction via Definition 8.)

Definition 10 (Loop approximation)

$$
\begin{array}{lcl}
 & & \mathrm{awlp}.(\textbf{while}\, B\, \textbf{loop}\, S\, \textbf{end loop}).R \\
 & \equiv & \text{'for well-defined loop invariant } I \text{ and if } S \text{ updates variables } x \text{ only'} \\
 & & I \wedge (\forall x \bullet I \Rightarrow \mathrm{def}.B \Rightarrow (B \vee R))
\end{array}
$$

This definition can be used in path analyses in exactly the same way as the wlp definition for loops was used in Section 8. Unlike Theorem 2, it does not require us to reason over natural number n, or to examine in detail the loop body S. However, the number of dead paths detectable using this definition depends on the choice of invariant. The stronger the loop invariant, the better the approximation, and the more dead paths will be identified.

For example, when analysing Path 4 in Section 8 we calculated the weakest liberal precondition of the **while** statement, with respect to postcondition 'false', to be

$d \leqslant a \wedge d \leqslant 0$. To show an approximate analysis of this loop, consider the following choice of invariant. (This is part of the full invariant given in Section 4.)

$$J \stackrel{\text{def}}{=} 0 \leqslant d \wedge 0 \leqslant a$$

We can then apply the awlp definition above to find an approximation to the precondition that makes the loop dead as follows.

$$\begin{array}{ll} & \text{awlp.}(\textbf{while } d \leqslant a \textbf{ loop} \\ & \qquad a := a - d \\ & \qquad \textbf{end loop}).\text{false} \\ \equiv & \text{'by Definition 10 using invariant } J\text{'} \\ & (0 \leqslant d \wedge 0 \leqslant a) \wedge \\ & (\forall a \bullet (0 \leqslant d \wedge 0 \leqslant a) \Rightarrow \\ & \qquad \text{def.}(d \leqslant a) \Rightarrow \\ & \qquad (d \leqslant a \vee \text{false})) \\ \equiv & \text{'since } d \leqslant a \text{ is well defined'} \\ & (0 \leqslant d \wedge 0 \leqslant a) \wedge \\ & (\forall a \bullet (0 \leqslant d \wedge 0 \leqslant a) \Rightarrow d \leqslant a) \\ \equiv & (0 \leqslant d \wedge 0 \leqslant a) \wedge \\ & \neg(\exists a \bullet 0 \leqslant d \wedge 0 \leqslant a \wedge a < d) \\ \equiv & 0 \leqslant a \wedge d = 0 \\ \equiv & d \leqslant a \wedge d = 0 \end{array}$$

This new condition is stronger than the one calculated in Section 8. It requires $d = 0$, rather than just $d \leqslant 0$, so this approximation identifies only a subset of the dead paths found earlier.

10 Conclusion

We have shown that weakest liberal precondition semantics offers a formal way of characterising control-flow paths that can never be followed to completion at run time. This was done by treating primitive parts of compound programming language statements as distinct statements in their own right and allowing domain constraints to be expressed as assertions. Existing or planned dead-path identification algorithms can now be justified in terms of this formalism by proving that they behave conservatively and identify only genuinely dead paths as 'dead' (although they may still fail to identify some dead paths).

Formally verifying that paths are dead using, for instance, an interactive theorem prover would be unacceptably inefficient in most practical situations. Nevertheless, the formalism could be used in this way to eliminate paths missed by the automatic algorithms in those high-integrity applications that justify the additional effort. This would be particularly important for eliminating lengthy dead paths that were missed by the automatic algorithms during timing analysis of safety-critical programs.

In particular, this study was motivated in part by research into formal methods of deriving real-time programs from their specifications [25]. In this situation, additional information about the program, such as pre and post-conditions and loop invariants, is already available and can be readily exploited during formal dead-path analysis.

Acknowledgements

We wish to thank Axel Wabenhorst for correcting errors in this paper and Albert Nymeyer, Carroll Morgan, Ray Nickson and Ken Robinson for pointing out relevant references. We also wish to thank the anonymous reviewers for suggesting improvements to earlier versions of this work. This research was funded by Australian Research Council Large Grant A49937045: *Effective Real-Time Program Analysis.*

References

[1] BODÍK, R., GUPTA, R., and SOFFA, M. L.: 'Refining data flow information using feasible paths.' In M. Jazayeri and H. Schauer, editors, *Software Engineering — ESEC/FSE'97*, volume 1301 of *Lecture Notes in Computer Science*, pages 361–377. Springer-Verlag, 1997.

[2] LUNDQVIST, T., and STENSTRÖM, P.: 'An integrated path and timing analysis method based on cycle-level symbolic execution.' *Real-Time Systems*, 17(2/3), November 1999.

[3] DIJKSTRA, E. W.: '*A Discipline of Programming.*' Prentice-Hall, 1976.

[4] PFLEEGER, S. L.: '*Software Engineering: The Production of Quality Software.*' Macmillan, 1991. Second edition.

[5] ERMEDAHL, A., and GUSTAFSSON, J.: 'Deriving annotations for tight calculation of execution time.' In C. Lengauer, M. Griebel, and S. Gorlatch, editors, *Euro-Par'97: Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 1298–1307. Springer-Verlag, 1997.

[6] CHAPMAN, R., BURNS, A., and WELLINGS, A.: 'Combining static worst-case timing analysis and program proof.' *Real-Time Systems*, 11:145–171, 1996.

[7] FIDGE, C. J., HAYES, I. J., and WATSON, G.: 'The deadline command.' *IEE Proceedings—Software*, 146(2):104–111, April 1999.

[8] CARRÉ, B.: 'Program analysis and verification.' In C. T. Sennett, editor, *High-Integrity Software*, chapter 8, pages 176–197. Plenum Press, 1989.

[9] PUSCHNER, P., and SCHEDL, A. V.: 'Computing maximum task execution times: A graph-based approach.' *Real-Time Systems*, 13(1):67–91, July 1997.

[10] MORGAN, C.: '*Programming from Specifications.*' Prentice-Hall, 1990.

[11] HEALY, C. A., and WHALLEY, D. B.: 'Automatic detection and exploitation of branch constraints for timing analysis.' *IEEE Transactions on Software Engineering*, 2001. Accepted for publication.

[12] PARK, C. Y.: 'Predicting program execution times by analyzing static and dynamic program paths.' *Real-Time Systems*, 5:31–62, 1993.

[13] LI, Y.-T., MALIK, S., and WOLFE, A.: 'Cinderella: A retargetable environment for performance analysis of real-time software.' In C. Lengauer, M. Griebel, and S. Gorlatch, editors, *Euro-Par'97: Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 1308–1315. Springer-Verlag, 1997.

[14] ENGBLOM, J., and ERMEDAHL, A.: 'Modeling complex flows for worst-case execution time analysis.' In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 163–174. IEEE Computer Society, 2000.

[15] ALTENBERND, P.: 'On the false path problem in hard real-time programs.' In *Proc. 8th Euromicro Workshop on Real-Time Systems (WRTS)*, pages 102–107, 1996.

[16] LUNDQVIST, T., and STENSTRÖM, P.: 'Integrating path and timing analysis using instruction-level simulation techniques.' In F. Mueller and A. Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, volume 1474 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1998.

[17] GUNTER, E. L., and PELED, D.: 'Path exploration tool.' In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 405–419. Springer-Verlag, 1999.

[18] ALJIFRI, H., TAPIA, M., and PONS, A.: 'Computation of WCET by detecting feasible paths.' In W. C. H. Cheng and A. S. M. Sajeev, editors, *PART'99: Proceedings of the 6th Australasian Conference on Parallel And Real-Time Systems*, pages 377–385. Springer-Verlag, 1999.

[19] TUCKER TAFT, S., and DUFF, R. A., editors: '*Ada 95 Reference Manual: Language and Standard Libraries*,' volume 1246 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[20] DIJKSTRA, E. W., and SCHOLTEN, C. S.: '*Predicate Calculus and Program Semantics.*' Springer-Verlag, 1990.

[21] NELSON, G.: 'A generalization of Dijkstra's calculus.' *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.

[22] BACK, R.-J.: 'Refinement calculus, lattices and higher order logic.' In M. Broy, editor, *Program Design Calculi*, pages 53–72. Springer-Verlag, 1993.

[23] BACK, R.-J., and VON WRIGHT, J.: '*Refinement Calculus: A Systematic Introduction.*' Springer-Verlag, 1998.

[24] MORGAN, C., and ROBINSON, K.: 'Specification statements and refinement.' *IBM Journal of Research and Development*, 31(5), September 1987.

[25] HAYES, I. J., and UTTING, M.: 'A sequential real-time refinement calculus.' *Acta Informatica*, 37(6):385–448, 2001.

[26] BACK, R.-J., and VON WRIGHT, J.: 'Reasoning algebraically about loops.' *Acta Informatica*, 36(4):295–334, July 1999.

[27] TARSKI, A.: 'A lattice theoretical fixed point theorem and its application.' *Pacific Journal of Mathematics*, 5:285–309, 1955.

A Dead subpaths

In this appendix we prove Theorem 1 which states that if any subpath of a path is dead, then the whole path is dead. In practice this property can save a considerable amount of effort since it allows proof that a path is dead to stop as soon as some part of the path is shown to be so.

We begin by noting two properties of wlp.S for each statement S in our programming language. Firstly, the predicate-transformer functions for programming language statements are monotonic with respect to the entailment relation '$\Rrightarrow$'.

Proposition 1 (Monotonicity)

$$\text{If } Q \Rrightarrow R \text{ then } \text{wlp}.S.Q \Rrightarrow \text{wlp}.S.R\,.$$

This is so because if S terminates in a state satisfying Q, then that state will also satisfy R.

Secondly, the following property holds for each statement S in our target programming language, because any state satisfies the predicate 'true' [3, p. 22].

Proposition 2 (Wlp true)

$$\text{wlp}.S.\text{true} \quad \equiv \quad \text{true}$$

Our goal is then to prove that if statement S_2 is dead then the sequence of statements 'S_1 ; S_2 ; S_3' is also dead. Assuming that S_2 is dead gives us the following property.

$$\begin{array}{ll} & \text{wlp}.S_2.\text{false} \\ \equiv & \text{'by Definition 2, since } S_2 \text{ is dead'} \\ & \text{true} \end{array}$$

Observe that 'false $\Rrightarrow$ wlp.S.false' for any statement S. Therefore the following property must hold.

$$\begin{array}{ll} & \text{wlp}.S_2.\text{false} \\ \Rrightarrow & \text{'by Proposition 1'} \\ & \text{wlp}.S_2.(\text{wlp}.S_3.\text{false}) \end{array}$$

Hence from our assumption that wlp.S_2.false is 'true', the following property holds.

$$\text{wlp}.S_2.(\text{wlp}.S_3.\text{false}) \quad \equiv \quad \text{true}$$

Finally, we use this property to complete the proof that the whole path is dead.

$$\begin{array}{ll} & \text{wlp}.(S_1 \,;\, S_2 \,;\, S_3).\text{false} \\ \equiv & \text{'by the semantics of '}\mathbf{;}\text{''} \\ & \text{wlp}.S_1.(\text{wlp}.S_2.(\text{wlp}.S_3.\text{false})) \\ \equiv & \text{wlp}.S_1.\text{true} \\ \equiv & \text{'by Proposition 2'} \\ & \text{true} \end{array}$$

B Semantics of iteration

For completeness, this appendix derives the semantics of **while** loops introduced in Section 6, and the special case used in Section 8. Our approach is informed by Dijkstra and Scholten's extensive study

$$
\begin{array}{lcl}
F^0.\text{true} & \equiv & \text{true} \\
F^1.\text{true} & \equiv & \text{wlp.}([B]\,;S).\text{true} \wedge (\text{def}.B \Rightarrow (B \vee R)) \\
 & \equiv & \text{`by Proposition 2'} \\
 & & \text{def}.B \Rightarrow (B \vee R) \\
 & \equiv & \text{`since } (\text{wlp.}([B]\,;S))^0 \text{ is the identity function'} \\
 & & (\text{wlp.}([B]\,;S))^0.(\text{def}.B \Rightarrow (B \vee R)) \\
F^2.\text{true} & \equiv & \text{wlp.}([B]\,;S).(F^1.\text{true}) \wedge (\text{def}.B \Rightarrow (B \vee R)) \\
 & \equiv & \text{wlp.}([B]\,;S).((\text{wlp.}([B]\,;S))^0.(\text{def}.B \Rightarrow (B \vee R))) \wedge \\
 & & (\text{def}.B \Rightarrow (B \vee R)) \\
 & \equiv & (\text{wlp.}([B]\,;S))^1.(\text{def}.B \Rightarrow (B \vee R)) \wedge \\
 & & (\text{wlp.}([B]\,;S))^0.(\text{def}.B \Rightarrow (B \vee R)) \\
F^3.\text{true} & \equiv & \text{wlp.}([B]\,;S).(F^2.\text{true}) \wedge (\text{def}.B \Rightarrow (B \vee R)) \\
 & \equiv & \text{wlp.}([B]\,;S).((\text{wlp.}([B]\,;S))^1.(\text{def}.B \Rightarrow (B \vee R)) \wedge \\
 & & \qquad (\text{wlp.}([B]\,;S))^0.(\text{def}.B \Rightarrow (B \vee R))) \wedge \\
 & & (\text{def}.B \Rightarrow (B \vee R)) \\
 & \equiv & \text{`by Proposition 3'} \\
 & & \text{wlp.}([B]\,;S).((\text{wlp.}([B]\,;S))^1.(\text{def}.B \Rightarrow (B \vee R))) \wedge \\
 & & \text{wlp.}([B]\,;S).((\text{wlp.}([B]\,;S))^0.(\text{def}.B \Rightarrow (B \vee R))) \wedge \\
 & & (\text{def}.B \Rightarrow (B \vee R)) \\
 & \equiv & (\text{wlp.}([B]\,;S))^2.(\text{def}.B \Rightarrow (B \vee R)) \wedge \\
 & & (\text{wlp.}([B]\,;S))^1.(\text{def}.B \Rightarrow (B \vee R)) \wedge \\
 & & (\text{wlp.}([B]\,;S))^0.(\text{def}.B \Rightarrow (B \vee R)) \\
 & \vdots & \\
F^n.\text{true} & \equiv & \forall i : 0 \ldots n-1 \bullet (\text{wlp.}([B]\,;S))^i.(\text{def}.B \Rightarrow (B \vee R))
\end{array}
$$

Figure 3: Calculation for fixed point of iteration.

of weakest liberal preconditions for loops [20, Ch. 9–10] and Back and von Wright's algebraic treatment [26].

Consider an iterative statement W with condition B and body S defined as follows.

$$W \stackrel{\text{def}}{=} \textbf{while}\, B\, \textbf{loop}\, S\, \textbf{end loop}$$

Informally, the behaviour of such a statement is usually explained via a recursive equation [10, §7.1].

$$W = \textbf{if}\, B\, \textbf{then}\, (S\,;W)\, \textbf{end if}$$

Using this approach, we can derive a recursive form of W's weakest liberal precondition with respect to an arbitrary postcondition R.

$$
\begin{array}{lcl}
 & & \text{wlp.}W.R \\
 & \equiv & \text{wlp.}(\textbf{if}\, B\, \textbf{then}\, (S\,;W)\, \textbf{end if}).R \\
 & \equiv & \text{`by the semantics of `;' and \textbf{if}'} \\
 & & \text{def}.B \Rightarrow \\
 & & \quad ((B \Rightarrow \text{wlp.}S.(\text{wlp.}W.R)) \wedge \\
 & & \quad (\neg B \Rightarrow R)) \\
 & \equiv & (\text{def}.B \Rightarrow \\
 & & \quad (B \Rightarrow \text{wlp.}S.(\text{wlp.}W.R))) \wedge \\
 & & (\text{def}.B \Rightarrow (\neg B \Rightarrow R)) \\
 & \equiv & \text{`by the semantics of guard } [B]\text{'} \\
 & & (\text{wlp.}[B].(\text{wlp.}S.(\text{wlp.}W.R))) \wedge \\
 & & (\text{def}.B \Rightarrow (\neg B \Rightarrow R)) \\
 & \equiv & \text{wlp.}([B]\,;S).(\text{wlp.}W.R) \wedge \\
 & & (\text{def}.B \Rightarrow (B \vee R))
\end{array}
$$

We now have a recursive equation of the form '$X \equiv F.X$', where the weakest solu-

$$
\begin{array}{ll}
& (\text{wlp}.([B]\,;S))^{n+1}.(\text{def}.B \Rightarrow B) \\
\equiv & \text{wlp}.([B]\,;S).((\text{wlp}.([B]\,;S))^{n}.(\text{def}.B \Rightarrow B)) \\
\equiv & \text{`by Definition 3'} \\
& \text{wlp}.[B].(\text{wlp}.S.((\text{wlp}.([B]\,;S))^{n}.(\text{def}.B \Rightarrow B))) \\
\equiv & \text{`by Definition 5'} \\
& \text{def}.B \Rightarrow (B \Rightarrow (\text{wlp}.S.((\text{wlp}.([B]\,;S))^{n}.(\text{def}.B \Rightarrow B)))) \\
\equiv & \text{`since def}.B \Rightarrow B \text{ from the base case'} \\
& \text{def}.B \Rightarrow (\text{wlp}.S.((\text{wlp}.([B]\,;S))^{n}.(\text{def}.B \Rightarrow B))) \\
\equiv & \text{`by the inductive hypothesis'} \\
& \text{def}.B \Rightarrow (\text{wlp}.S.((\text{wlp}.([\text{def}.B]\,;S))^{n}.(\text{def}.B \Rightarrow B))) \\
\equiv & \text{`since def.(def.}X\text{) is true for any } X\text{'} \\
& \text{def.(def.}B) \Rightarrow (\text{def}.B \Rightarrow (\text{wlp}.S.((\text{wlp}.([\text{def}.B]\,;S))^{n}.(\text{def}.B \Rightarrow B))))) \\
\equiv & \text{`by Definition 5'} \\
& \text{wlp}.[\text{def}.B].(\text{wlp}.S.((\text{wlp}.([\text{def}.B]\,;S))^{n}.(\text{def}.B \Rightarrow B))) \\
\equiv & \text{`by Definition 3'} \\
& \text{wlp}.([\text{def}.B]\,;S).((\text{wlp}.([\text{def}.B]\,;S))^{n}.(\text{def}.B \Rightarrow B)) \\
\equiv & (\text{wlp}.([\text{def}.B]\,;S))^{n+1}.(\text{def}.B \Rightarrow B)
\end{array}
$$

Figure 4: Inductive step for dead iteration proof.

tion for 'X' is 'wlp.W.R' [20, p. 171], and F is a monotonic predicate-transformer function.

$$
\begin{array}{lll}
F.X & \equiv & \text{wlp}.([B]\,;S).X \wedge \\
& & (\text{def}.B \Rightarrow (B \vee R))
\end{array}
$$

Our goal is to find the weakest solution of this equation in the predicate lattice defined by the entailment relation $\Rrightarrow$, with top 'true' and bottom 'false'. To do so, we first note that for each statement S in our programming language, predicate-transformer function wlp.S is $\wedge$-continuous [20].

Proposition 3 (And-continuity) *Let I be an arbitrary indexing set, and for each i in I let R_i be a predicate.*

$$
\begin{array}{ll}
& \text{wlp}.S.(\bigwedge i : I \bullet R_i) \\
\equiv & (\bigwedge i : I \bullet \text{wlp}.S.R_i)
\end{array}
$$

Fixed point theory [27, 23] then allows us to determine the weakest solution (i.e., the greatest fixed point) of equation $X \equiv F.X$ as follows. Let F^n denote functional composition of n copies of function F, with F^0 being the identity function.

Proposition 4 (Weakest solution) *The weakest solution (greatest fixed point) of recursive equation $X \equiv F.X$ is given by the following expression.*

$$\forall n : \mathbb{N} \bullet F^n.\text{true}$$

To evaluate the expression in Proposition 4 for the recursive definition of wlp.W.R above, we must determine 'F^n.true', for any natural number n. This is most clearly demonstrated via cases as shown in Figure 3. From this we then obtain the weakest liberal precondition of **while** statements given in Section 6.

$$
\begin{array}{ll}
& \text{wlp}.(\textbf{while}\, B\, \textbf{loop}\, S\, \textbf{end loop}).R \\
\equiv & \text{`by Proposition 4'} \\
& \forall n : \mathbb{N} \bullet F^n.\text{true} \\
\equiv & \text{`from Figure 3'} \\
& \forall n : \mathbb{N} \bullet \\
& \quad (\forall i : 0 \ldots n-1 \bullet \\
& \qquad (\text{wlp}.([B]\,;S))^{i}.(\text{def}.B \Rightarrow \\
& \qquad\qquad (B \vee R))) \\
\equiv & \forall n : \mathbb{N} \bullet \\
& \quad (\text{wlp}.([B]\,;S))^{n}.(\text{def}.B \Rightarrow (B \vee R))
\end{array}
$$

C Dead iteration

Above we defined the weakest liberal precondition for an iterative statement given some arbitrary postcondition R. Here we prove Theorem 2, which allows for the special case where postcondition R is 'false', as needed for detecting dead iterations.

The proof is by induction. Before starting, we reexpress the right-hand side of Theorem 2 to make the base case explicit (recalling that $(\text{wlp}.S)^0$ is the identity function for any S).

$$
\begin{array}{ll}
 & \forall n : \mathbb{N} \bullet \\
 & \quad (\text{wlp}.([\text{def}.B] \,;\, S))^n.(\text{def}.B \Rightarrow B) \\
\equiv & \text{'by conjoining case where } n = 0\text{'} \\
 & (\text{def}.B \Rightarrow B) \wedge \\
 & (\forall n : \mathbb{N} \bullet \\
 & \quad (\text{wlp}.([\text{def}.B] \,;\, S))^n.(\text{def}.B \Rightarrow B))
\end{array}
$$

Proof of this restated theorem then begins by reexpressing the left-hand side in the same way.

$$
\begin{array}{ll}
 & \text{wlp}.(\textbf{while}\, B\, \textbf{loop}\, S\, \textbf{end loop}).\text{false} \\
\equiv & \text{'by Definition 8'} \\
 & \forall n : \mathbb{N} \bullet \\
 & \quad (\text{wlp}.([B] \,;\, S))^n.(\text{def}.B \Rightarrow B) \\
\equiv & \text{'by conjoining case where } n = 0\text{'} \\
 & (\text{def}.B \Rightarrow B) \wedge \\
 & (\forall n : \mathbb{N} \bullet \\
 & \quad (\text{wlp}.([B] \,;\, S))^n.(\text{def}.B \Rightarrow B))
\end{array}
$$

The proof then proceeds by pointwise equivalence on n. For the base case, where $n = 0$, the two predicates derived above are trivially equivalent. For the inductive step, we assume that $\text{def}.B \Rightarrow B$ holds and that the universally quantified predicates are equivalent for the nth case, and then show that this is also true for $n + 1$, as in Figure 4.