Chapter 7

Computing Dataflow Dependencies of Schemas with Loops

7.1 Introduction

In this chapter, the algorithms introduced in Chapter 6, which compute dataflow dependencies of loop–free schemas, are extended to compute the *DTVD*[1] of schemas that contain loops.

The algorithm works by *unfolding* all the loops within a schema. A schema, which has had all its loops replaced by an unfolding is loop–free and hence *DTVD* can be computed using the techniques described in Chapter 6.

It is proved that if a schema containing loops is unfolded *sufficiently*[2], the resulting loop–free schema will have the same *DTVD* as the original schema with loops from which it was derived.

The problem of computing *DTVD* has thus been reduced to the problem of recognising when a schema with loops has been sufficiently unfolded.

At the time of writing, unfortunately, we are not certain how to recognise when this *maximal* number of unfoldings has been reached. There are three possibilities.

Possibility 1 A 'maximal unfolding number[3]' of a schema can be computed recursively from 'crude' information about the structure of its abstract syntax tree.

---

[1] and hence, the *DTLD*
[2] A finite number of times.
[3] It need not be the least. *Any* one will be sufficient

Hausler [52] proved[4] that for every program p, a maximal number of unfoldings of p that are necessary to capture all dependence information is computable. His algorithm computes the maximal unfolding number of each compound syntactic construct in a program recursively in terms of the maximal unfolding numbers of each of its components. The maximum unfolding number for each basic syntactic construct is constant (For example, *one* for an assignment statement).

Possibility 2 There is some relationship between schemas such that if, two successive iterations of the unfolding process, give rise to schemas which are related in this way, then further iterations cannot introduce new dependencies.

Possibility 3 Recognising when a schema has been maximally unfolded is not computable.

For the dataflow dependencies introduced in this thesis, it seems very likely that the first of these is true. For example, in the case of a 'tiny' loop consisting of a single assignment statement, it can be seen that after two unfoldings all the dependence information has been gathered and further unfolding cannot add any new dependencies. As in the case of Hausler's work, it is highly likely that the maximal unfolding number for each syntactic construct in the language of schemas can also be expressed as a function of the maximal unfolding numbers of each of its components.

Taking advantage of this, an algorithm for $DTVD$ would compute a maximal unfolding number n_i for each syntactic component s_i of the schema and then simply unfold each component s_i, n_i times. The methods introduced in Chapter 6 could then be used to compute the $DTVD$ of the resulting loop–free schema.

Since at the time of writing, we do not know how to compute the maximal unfolding numbers for[5] dataflow dependencies, we cannot use them in our algorithms. The algorithms that we have implemented rely on the second assumption being true. The relationship that we use is that the two schemas have the same $DTD(= DTVD \cup DTLD)$[6]. As is demonstrated, unfolding is monotonic, i.e. further unfolding cannot reduce the $DTVD$. It also bounded

[4] We cannot necessarily assume that his result is true in our case, since we are computing different dependencies from Hausler.

[5] although, we believe them to be the same as Hausler's.

[6] There are probably others. Perhaps, for example, data dependence on its own is sufficient. It certainly works in all the examples that have been tested and is much more efficient.

above by a finite object. *If* it had been proved that no change in the *DTD* in *one* iteration of the unfolding process implied that further iterations *could not* introduce further changes to the *DTVD*, then the algorithm *would have been* proved correct.

The first and third possibilities are mutually contradictory. If the ability to recognise when a loop is maximally unfolded were not, in general decidable, it would mean there could be no connection between the 'size' of a loop and the number of iterations of it that were required before all dependence information was 'gathered'. It would mean for some classes of programs, no such maximum upper limit based on crude syntactic properties involving the numbers of statements would exist. In view of Hausler's work, this seems highly unlikely.

7.2 Unfoldings

Hausler [52] defines the denotational slice of a while loop in terms of unfolding the loop as a nested conditional. A very similar approach is used in this thesis.

Definition 7.2.1 (Unfolding)
Given a schema $[\![\textit{while } b \textit{ do } S]\!]$, define the sequence of schemas:

$$W_0(b,S) = \textit{if } b \textit{ then } FAIL \textit{ else skip}$$
$$W_{n+1}(b,S) = \textit{if } b \textit{ then } S; W_n(b,S) \textit{ else skip.}$$

We call $W_i(b,S)$ the *ith unfolding* of $[\![\textit{while } b \textit{ do } S]\!]$.

7.2.1 Example of unfolding

Consider the schema:-

$$\begin{array}{l} \textit{while } b1(x,y) \\ \textit{do if } b2(x) \\ \quad \textit{then } x := f(x,y) \\ \quad \textit{else } y := g(y) \end{array}$$

The zeroth unfolding: W_0=

$$\begin{array}{l} \textit{if } b1(x,y) \\ \textit{then } FAIL \\ \textit{else skip} \end{array}$$

The first unfolding is given by : $W_1 =$

if $b1(x,y)$ *then* $S; W_0$ *else skip*

where S is the body of the loop i.e. $W_1 =$

if $b1(x,y)$
then *if* $b2(x)$
    then $x := f(x,y)$
    else $y := g(y)$;
    W_0
else skip

i.e. $W_1 =$

if $b1(x,y)$
then *if* $b2(x)$
    then $x := f(x,y)$
    else $y := g(y)$;
    [*if* $b1(x,y)$ / *then FAIL* / *else skip*]
else skip

The second unfolding, $W_2 =$

if $b1(x,y)$
then *if* $b2(x)$
    then $x := f(x,y)$
    else $y := g(y)$;
    W_1
else skip

The second unfolding, W_2 is thus

```
if b1(x, y)
then  if b2(x)
      then x := f(x, y)
      else y := g(y);
          if b1(x, y)
          then   if b2(x)
                 then x := f(x, y)
                 else y := g(y);
                   if b1(x, y)
                   then FAIL
                   else skip
          else skip
else skip
```

Observation 7.2.1 *To get from $W_n(b, S)$ to $W_{n+1}(b, S)$, the occurrence of FAIL is replaced by*

$$[\![S; \textit{if } b \textit{ then FAIL else skip}]\!]$$

7.3 The *DTVD* Algorithm for Loop Schemas

The way the algorithm works with loops is that loops are repeatedly unfolded. Eventually a stage will be reached where the *DTVD* is *maximal* in the sense that further unfoldings will not cause further changes in the *DTVD*.

7.3.1 Example

Consider, again, the schema,

while $b1(x,y)$
do *if* $b2(x)$
  then $x := f(x,y)$
  else $y := g(y)$

. Using the semantics of loop free schemas given in Chapter 4, the zeroth unfolding, W_0=

if $b1(x,y)$
then *FAIL*
else *skip*

has symbolic execution tree $\mathcal{S}[\![W_0]\!]$ given in Figure 7.1(page 212).

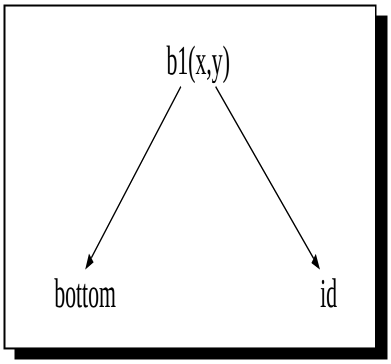

Figure 7.1: $\mathcal{S}[\![W_0]\!]$: The symbolic execution tree of W_0

The data dependence DTslice (Definition 6.6.5(page 201)) is computed The results are shown in Figure 7.2(page 212): The variables x and y are not dataflow terminating dependent on any variable or label[7].

Variable	Data	Control
x	$\emptyset$	$\emptyset$
y	$\emptyset$	$\emptyset$

Figure 7.2: DTslice of $\mathcal{S}[\![W_0]\!]$

[7] If we had used non-terminating control dependence (Definition 6.6.4(page 200)), x and y would both have depended on x, y and b_1.

The loop is unfolded once more to give $W_1 =$

```
if b1(x, y)
then if b2(x)
     then x := f(x, y)
     else y := g(y);
     ┌──────────────┐
     │ if b1(x, y)  │
     │ then FAIL    │
     │ else skip    │
     └──────────────┘
else skip
```

W_1 has the symbolic execution tree $\mathcal{S}[\![W_1]\!]$ given in Figure 7.3(page 213).

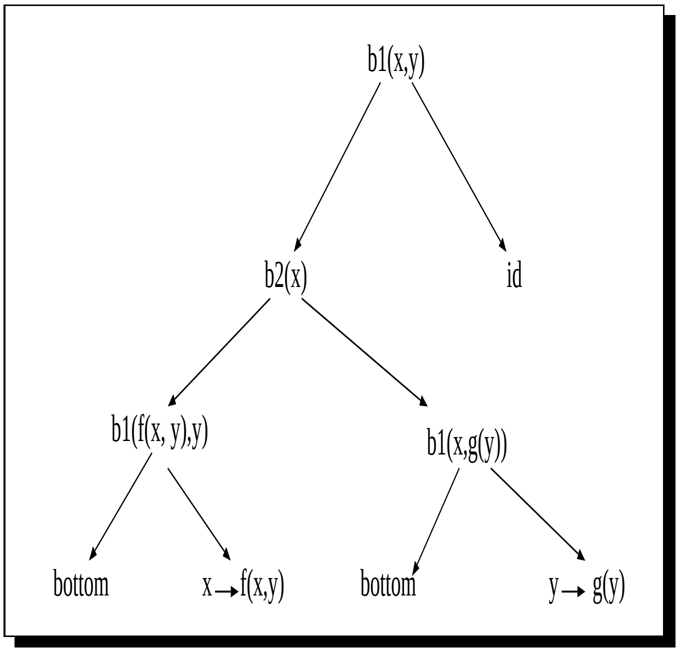

Figure 7.3: $\mathcal{S}[\![W_1]\!]$: The symbolic execution tree of W_1

Again, the DTslice is computed. The results are shown in Figure 7.4(page 213).

Variable	Control	Data
x	$\{b_1, b_2, x, y\}$	$\{f, x, y\}$
y	$\{b_1, b_2, x, y\}$	$\{g, y\}$

Figure 7.4: DTslice of $\mathcal{S}[\![W_1]\!]$

The second unfolding, $W_2 =$

```
if b1(x, y)
then if b2(x)
     then x := f(x, y)
     else y := g(y);
     if b1(x, y)
     then if b2(x)
          then x := f(x, y)
          else y := g(y);
           if b1(x, y)
           then FAIL
           else skip
     else skip
else skip
```

has the symbolic execution tree $\mathcal{S}[\![W_2]\!]$ given in Figure 7.5(page 214).

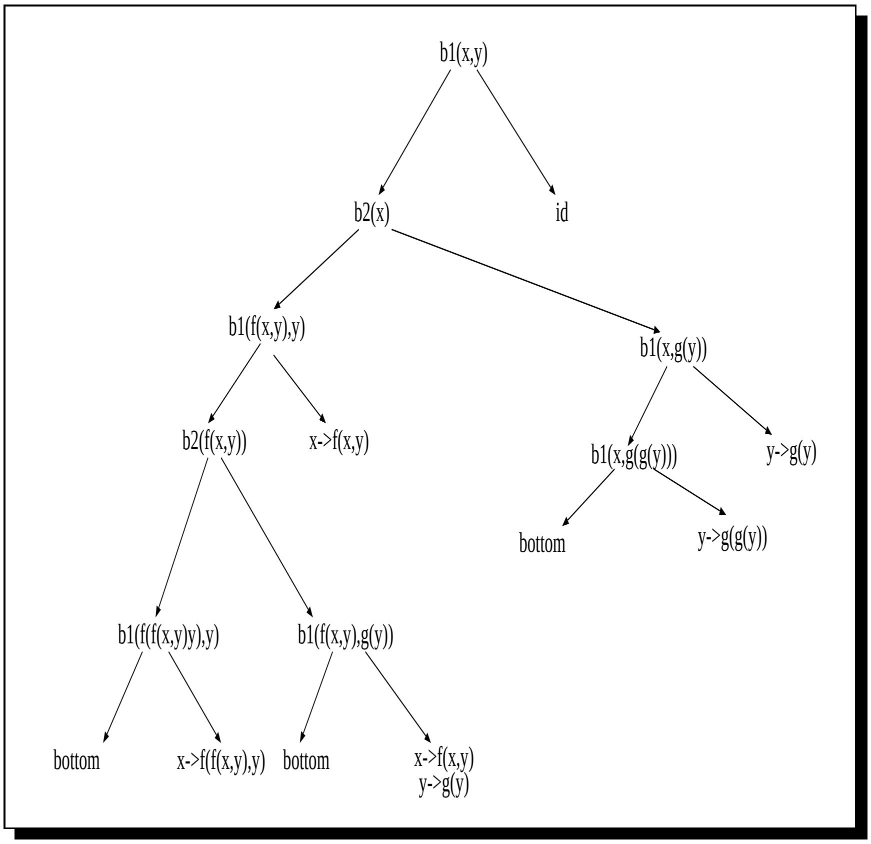

Figure 7.5: The symbolic execution tree, $\mathcal{S}[\![W_2]\!]$ of W_2

Notice, by Observation 7.2.1(page 211), to get from one unfolding to the next we replace each $\perp$ by the symbolic execution tree, $\mathcal{S}[\![S; \mathit{if}\, b1(x,y)\ \mathit{then\ FAIL\ else\ skip}]\!]$ namely, the symbolic execution tree given in Figure 7.6(page 215), evaluated in the state immediately to the right of the $\perp$ being replaced.

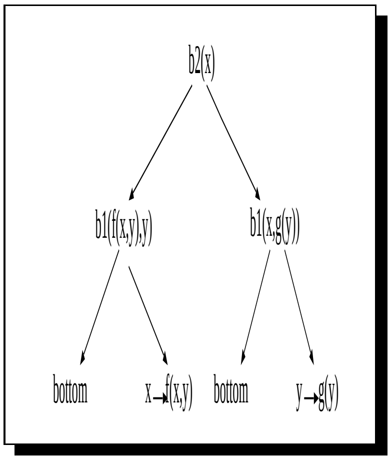

Figure 7.6: $\mathcal{S}[\![S;\, \mathit{if}\, b1(x,y)\ \mathit{then}\ \mathit{FAIL}\ \mathit{else}\ \mathit{skip}]\!]$

Notice also, that at this stage, some simplification (Section 4.4.4) has taken place. The symbolic execution tree corresponding to $\mathcal{S}[\![W_2]\!]$ before pruning is given in Figure 7.7(page 216).

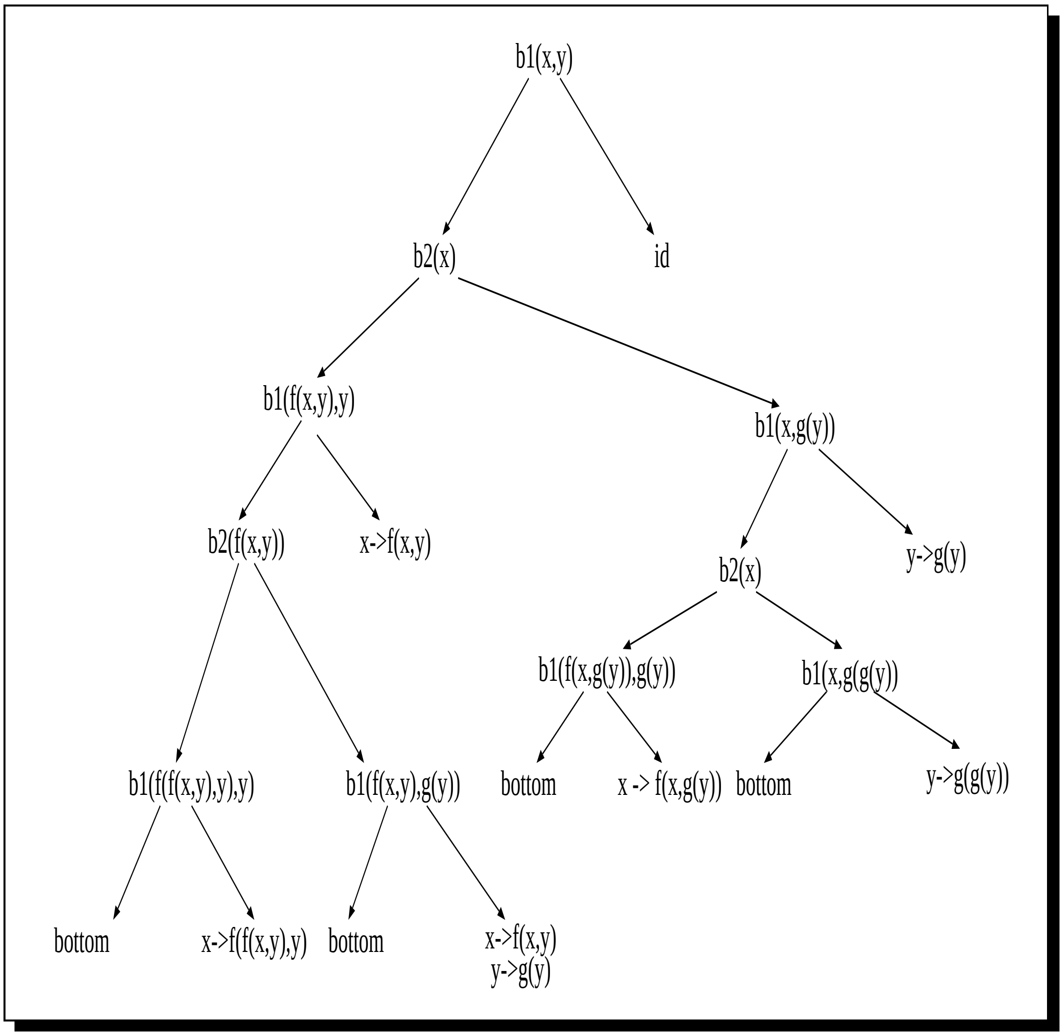

Figure 7.7: The symbolic execution tree, $\mathcal{S}[\![W_2]\!]$ before pruning

The subtree to the left of the lower occurrence of $b_2(x)$ has been removed since it is represents impossible paths.

Again, this time using $\mathcal{S}[\![W_2]\!]$, the DTslice is computed The results are shown in Figure 7.8(page 216).

Variable	Control	Data
x	$\{b_1, b_2, f, x, y\}$	$\{f, x, y\}$
y	$\{b_1, b_2, f, x, y\}$	$\{g, y\}$

Figure 7.8: DTslice of $\mathcal{S}[\![W_2]\!]$

A new control dependence of x on f has emerged.

The process is repeated once more. W_3 is the schema which has the symbolic execution tree $\mathcal{S}[\![W_3]\!]$ given in Figure 7.9(page 217).

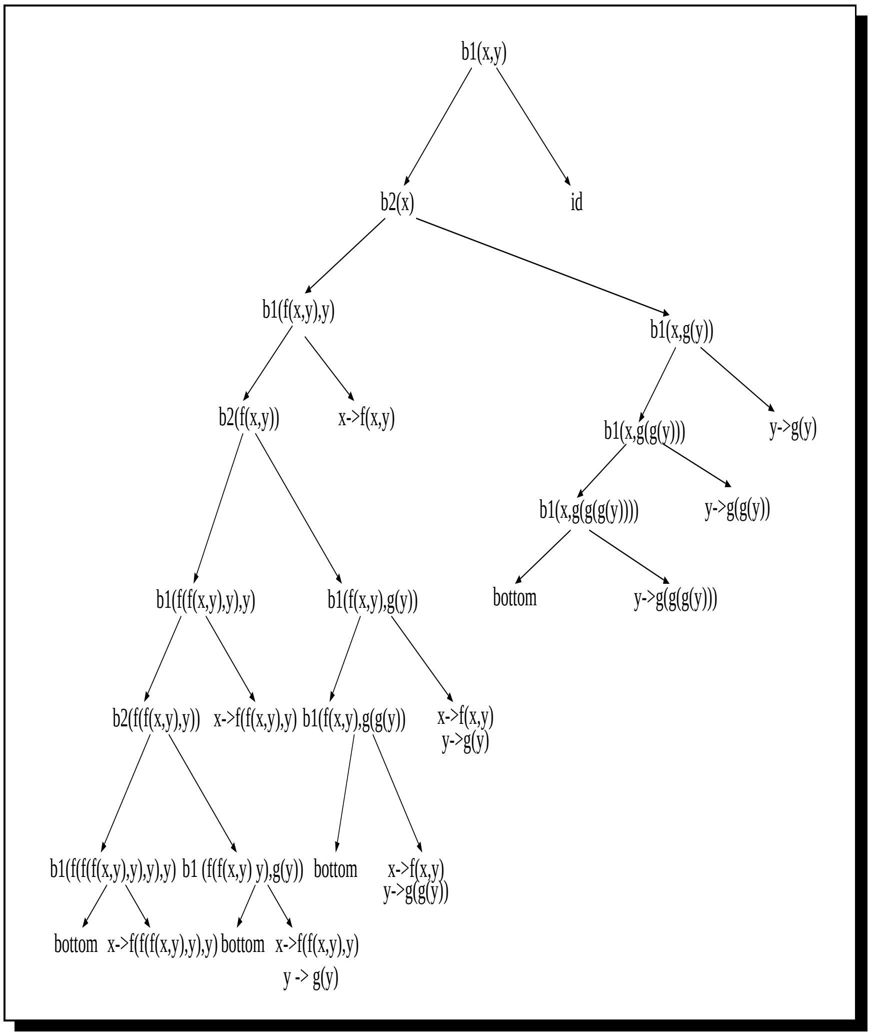

Figure 7.9: The Symbolic Execution Tree, $\mathcal{S}[\![W_3]\!]$

This time we find there is no change in the DTslice (Figure 7.10(page 217)). The schema has reached its maximal unfolding so the algorithm terminates.

Variable	Control	Data
x	$\{b_1, b_2, f, x, y\}$	$\{f, x, y\}$
y	$\{b_1, b_2, f, x, y\}$	$\{g, y\}$

Figure 7.10: DTslice of $\mathcal{S}[\![W_3]\!]$

Since it is loop–free, to produce the *DTVD* for each variable(see Figure 7.11(page 218)), we simply compute the DTVslice (Definition 6.4.4(page 195)) of this maximal unfolding ($\mathcal{S}[\![W_3]\!]$) (or simply restrict the range of the DTslice to just variables as described in Chapter 6).

Variable	$DTVD$
x	$\{x, y\}$
y	$\{x, y\}$

Figure 7.11: $DTVD$ of the Loop Schema

and to produce the $DTLD$ for each variable(see Figure 7.12(page 218)), we simply compute the DTLslice (Definition 6.2.4(page 182)) of $\mathcal{S}[\![W_3]\!]$ (or simply restrict the range of the DTslice to just labels as described in Chapter 6).

Variable	$DTLD$
x	$\{b_1, b_2, f\}$
y	$\{b_1, b_2, f, g\}$

Figure 7.12: $DTLD$ of the Loop Schema

Notice that the DTLslice with respect to x does not contain g.

7.4 Dataflow Dependence of Unfoldings

The relationship between a loop schema and its ith unfolding is embodied in the following lemma. Lemma 7.4.1 states that a program represented by a loop schema and the corresponding program in the ith unfolding of the loop schema agree in all states where the loop terminates in i or fewer iterations. In all other states the latter program will not terminate.

Lemma 7.4.1 *Let $p \in [while\ b\ do\ S]$ and let p' be the 'corresponding' program in $[W_i(b, S)]$. Then for all states σ in which p terminates in i or fewer iterations*

$$\mathcal{M}[\![p]\!]\sigma = \mathcal{M}[\![p']\!]\sigma$$

and in all other states $\mathcal{M}[\![p']\!]\sigma = \perp$.

Corollary 7.4.1 *Let $[\![while\ b\ do\ S]\!]$ be a loop schema. Let i and j be natural numbers with $i > j$. Let p_i and p_j be 'corresponding' programs in its ith unfolding,$[W_i(b, S)]$ and jth*