Scaling up Property Checking

A thesis submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

by

Shrawan Kumar
(Roll No. 09405701)

Under the guidance of
Prof Amitabha Sanyal
and
Prof Uday Khedker

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
2019

To my parents

Thesis Approval

The thesis entitled

Scaling up Property Checking

by

Shrawan Kumar
(Roll No. 09405701)

is approved for the degree of

Doctor of Philosophy

Examiner

Examiner

Guide

Co Guide

Chairman

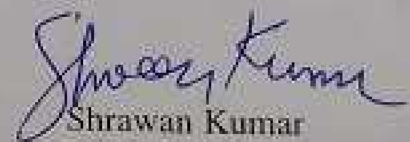
Date: 22-07-2019

Place: IIT Bombay

Declaration

I declare that this written submission represents my ideas in my own words and where other's ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date: 22-07-2019



Shrawan Kumar

Roll number: 09405701

Abstract

Historical data shows that there is an exorbitant cost involved in terms of loss of human lives and/or financial loss when software bugs escape to production runs, particularly for mission and safety critical systems. As a result, checking software correctness by static analysis has been widely accepted as means to mitigate such losses. Since proving absolute correctness is almost impossible, use of property checking is a widely accepted practice to prove correctness in parts. There are plenty of state-of-the-art tools and techniques (verifiers) available for property checking, such as CBMC [22], SLAM [5], SATABS [23] and ARMC [64], to name a few. However, these verifiers fall short of scaling to large real life applications, particularly in the presence of arrays of large size and loops with large bounds. In this thesis, we present two approaches which help the existing verifiers to scale up to such software systems.

In our first approach, we created a new slicing technique, called *value slice*, which helps reduce the program size considerably, while the resulting loss in precision is quite insignificant. While a backward slice is a commonly used pre-processing step for scaling property checking, for large programs, the reduced size of the slice may still be too large for verifiers to handle. Our idea of value slice is an aggressive slicing method that, apart from slicing out the same statements as backward slice, also eliminates computations that only decide whether the point of property assertion is reachable. However, for precision, we also carefully identify and retain *all* computations that influence the values of the variables in the property. The resulting slice is smaller and scales better for property checking than backward slice.

We carried out experiments on property checking of industry strength programs using three comparable slicing techniques: backward slice, value slice and thin slice, an even more aggressive slicing technique slice that retains only those statements on which the variables in the property are data dependent. While backward slicing enables highest precision and thin

slice scales best, value slice based property checking comes close to the best in both scalability and precision. This makes value slice a good compromise between backward and thin slice for property checking.

Our second approach is focused on handling scalability issues arising due to the presence of arrays of large size (> 100000). Most verifiers find it difficult to prove properties of programs containing loops that process arrays of such large or unknown size. These verifiers can be broadly classified into those that are abstract interpretation based and the ones that use model checkers equipped with array theories. As part of this approach, we present two techniques to handle the issue of scalability arising due to presence of such large arrays.

The first technique is based on a notion called *loop shrinkability*, which we have defined for loops. In this technique, an array processing loop, that is *shrinkable*, is transformed to a loop of much smaller bound that processes only a few non-deterministically chosen elements of the array. The result is a finite state program with a drastically reduced state space that can be analyzed by bounded model checkers. While the same bounded model checkers would have failed to check the property on the original program. We show that the proposed transformation is an over-approximation, i.e. if the property is true of the transformed program then it is also true of the original. In addition, whenever applicable, the method is impervious to the size,or existence of a bound, of the array. As an assessment of usefulness, a tool, *VeriAbs*, based on our method, could successfully verify 87 of the 93 programs of the *ArraysReach* category of SV-COMP 2017 benchmarks, with properties quantified over array indices. While several of these programs had multiple loops, there were no nested loops.

The second technique that is part of the approach to address the issue of scalability arising due to presence of such large arrays, is based on a notion called *loop pruning*. We have defined the notion of loop pruning also for loops of a program. In this technique, each loop is transformed to execute only the first few iterations of the loop from the beginning, and the property is also checked only for those array elements which are accessed in the resulting pruned loop. We present the criteria under which such transformation can be carried out and we show that the transformation is a sound abstraction with respect to the property being checked. We have implemented this technique also in the tool VeriAbs. The tool VeriAbs, equipped with both the techniques (loop shrinking and loop pruning), stood first in the *ArraysReach* category of SV-COMP 2018 competition.

Contents

Abstract **vii**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **3**
- 1.1 Scalability and precision in property checking 4
- 1.2 Some motivating examples 5
- 1.3 Our thesis 7
- 1.4 Related work 18
- 1.5 Organization 21

I Scaling up Property Checking Using Value Slice **23**

2 Background **25**
- 2.1 Control flow graph 25
- 2.2 Program states and traces 26
- 2.3 Subprograms 27
- 2.4 Backward slice 27
- 2.5 Data and control dependence 28

3 Value slice : A new slicing concept **31**
- 3.1 Concept of value slice 31
- 3.2 Value-impacting statements 34
- 3.3 Value slice from value impacting statements 36
- 3.4 Identifying VI statements using data and control dependence 39
- 3.5 Value slice computation 47

4 Implementation and measurements **53**
- 4.1 Implementation 53
- 4.2 Experiments 57

5 Related work **67**

II Scaling up Property Checking of Array Programs **71**

6 Background **73**
6.1 Imperative programs and states . . . 73
6.2 Bounded model checking . . . 74
6.3 Loop acceleration . . . 75
6.4 Programs and properties of interest . . . 75
6.5 State approximation for residual loops . . . 78

7 Loop shrinkability **81**
7.1 Definition of shrinkable loops . . . 81
7.2 Identifying shrinkable loops . . . 84
7.3 Checking shrinkability of iteration sequences of a size . . . 92
7.4 Determining loop shrinkability empirically . . . 94
7.5 Property checking for shrinkable loops . . . 96
7.6 Multiple loops and nested loops . . . 97

8 Implementation and measurements **101**
8.1 Implementation . . . 101
8.2 Experiments . . . 102

9 Loop pruning **107**
9.1 Basic idea . . . 107
9.2 Programs of interest . . . 110
9.3 Loop dependence graph and semantic constraints . . . 114
9.4 Replaying last value computations in the pruned loop . . . 120
9.5 Last value assignments and bound on their numbers . . . 122
9.6 Proof of soundness . . . 131
9.7 Implementation and measurements . . . 138

10 Related work **143**

III Concluding Remarks **147**

11 Conclusion and future directions **149**
11.1 Scaling up property checking through value slice . . . 149
11.2 Reducing the size of the loops . . . 153

List of Figures

1.1 Motivating examples to illustrate scalability issues 6
1.2 Usual backward slice, value slice and thin slice 10
1.3 Loop shrinking abstraction illustration 12
1.4 Loop pruning abstraction illustration 14

2.1 Example illustrating CFG and control dependence 26

3.1 Illustration of value slice 32
3.2 Generalisation of value slice 33
3.3 Example illustrating value impacting condition 35
3.4 (a) A property of CFG paths. (b)-(d) Situations that make a predicate value-impacting. In Fig (c), path π_1 is $c \rightarrow t \rightarrow d \rightarrow c \rightarrow u \rightarrow l$ 40
3.5 Example illustrating value impacting condition of type $cond_1$ 40
3.6 Example illustrating value impacting condition of type $cond_2$ 41
3.7 Example illustrating value impacting condition of type $cond_3$ 43
3.8 Algorithm to compute VI 50

4.1 Program to illustrate interprocedural data dependence computation 56
4.2 A graphical view of scalability and precision of value slice 60
4.3 Example showing limitation of value slice over backward slice 62

6.1 Illustration of loop acceleration and property checking loops 76
6.2 Illustration of residual loop for iteration sequence [2,4] 77
6.3 Illustration of residual property for iteration sequence [2,4] 78

7.1 Illustration of contrapositive view of shrinkable loops definition 82
7.2 Illustration of programs with no loop carried dependence 83
7.3 Examples illustrating similar loops having different shrinkability 84
7.4 Illustration of contrapositive view of sequence shrinkability definition 86
7.5 Illustration of problem with adapted definition of sequence shrinkability 87
7.6 Illustration of contrapositive view of revised definition of sequence shrinkability 89
7.7 Program construction for determining shrinkability. Note that `X` and `X_initial` are vectors of variables, and `nondet()`, accordingly, generates a vector of values. 93
7.8 Process to determine loop shrinkability and finding shrink-factor. 95
7.9 Example illustrating the residual of a shrinkable loop. Program in (b) is an abstract description of the residual, presented for ease of explanation. 97

7.10 Illustration of multiple loops that can be coalesced 98
7.11 Illustration of handling nested loops 99

9.1 Loop pruning abstraction illustration 108
9.2 Illustration of loop pruning approach 108
9.3 Grammar to describe the programs of interest 110
9.4 Illustration of loop dependence graph and cyclic dependence 115
9.5 Illustration of self control dependence 116
9.6 Illustration of constraints on conditions 117
9.7 Algorithm for span computation 119
9.8 Illustration of value reproducibility of variables 121
9.9 Illustration of computing a theoretical bound on #LVA 126
9.10 Algorithm and illustration for computing *instance-count* 128
9.11 Illustration of bound computation 130
9.12 Additional code patterns where loop pruning works 140

11.1 Illustration of limitation of value slice 151
11.2 Illustration of limitation of value slice when property encoded as unreachable error state 152
11.3 Illustration of property outcome reproduction based loop pruning 155

List of Tables

3.1 Description of variables used in the value slice computing algorithm 48

4.1 Program size and complexity 63
4.2 Scalability and precision of property checking based on different kinds of slices 64
4.3 Comparison of scalability and (%) loss in precision 65
4.4 Impact of increase in timeout, and change in CEGAR iterations and time taken 66

8.1 Experimental results for SV-COMP 2017 ArraysReach benchmarks 104

9.1 Experimental results for SV-COMP 2018 ArraysReach benchmarks 142

Nomenclature

σ	A program state as a map $Var \rightarrow Val$, during execution of a program, where Var is set of variables in the program, and Val is set of possible values, which the variables in Var can take
$\lfloor\sigma\rfloor_X$	An X-*restriction* map $X \rightarrow Val$ of a program state $\sigma : Var \rightarrow Val$, with respect to $X \subseteq Var$, such that $\forall x \in X.\lfloor\sigma\rfloor_X(x) = \sigma(x)$
Υ	A slicing criterion as a pair $\langle l, V\rangle$ where l is a statement label (location) and V is a set of variables
$REF(s)$	Set of variables referred in the statement s of a program
$LV(s)$	The slicing criterion $\langle l', REF(s)\rangle$, where l' is the label of statement s
$DU(l, X)$	Set of definitions on which variables in X are dependent at location l
$c \overset{b}{\rightarrowtail} n$	Node n is control dependent on conditional out-edge b of predicate node c
$c \overset{b}{\rightsquigarrow} n$	Node n is transitively control dependent on conditional out-edge b of predicate node c
(l, σ)	An execution state, in a trace, where l is a program location
$[(l_i, \sigma_i)],\ 0 \leq i \leq k$	Sequence of execution states $[(l_0, \sigma_0), (l_1, \sigma_1), ..., (l_k, \sigma_k)]$, representing a trace
π	A path in the control flow graph between two nodes
$\langle\pi_1, \pi_2, t\rangle$	A witness for a predicate c to be value impacting for a given slicing criterion. Here, π_1 and π_2 are paths from the node c to slicing criterion point, and t is first value impacting node on π_1, but not so on π_2
$VI(\Upsilon)$	Set of nodes value impacting the slicing criterion Υ
$AP(\Upsilon)$	Set of predicates that are not value impacting for Υ, but other value-impacting nodes are transitively control dependent on these predicates
P^{VS}	Subprogram constructed from program P, using statements of $VI(\Upsilon)$, and predicates of $AP(\Upsilon)$ in abstract form
AVI	Set of concrete statements in subprogram P^{VS}

$AREF(s)$	Set of variables referred in statement s of an augmented program. It is $REF(s)$ when $s \in VI(\Upsilon)$, V when s is $SKIP$ and $\{\}$ when s is *ENTRY*, where V is set of variables in the slicing criterion Υ
Φ	A set of program states, usually used for denoting a pre-condition
ψ	A set of program states, usually used for denoting a post-condition or a property as a formula
$\{\Phi\}P\{\psi\}$	A Hoare triple
$i : T$	A sequence whose first element is i and the suffix of the sequence, excluding the first element, is same as T
$\mathcal{P}_k(T)$	Set of all k-sized subsequences of a sequence T
L_T	Residual loop of a loop L with respect to a sequence T of loop iteration numbers
ψ_T	Residual property of a property ψ with respect to a sequence T of loop iteration numbers
φ	Set of program states at the beginning of first iteration of a loop
φ_i	An approximation of the set of program states at the beginning of i_{th} iteration of a loop
ℓ	Loop counter variable in a **for** loop
τ	Trace of original program
τ^p	Trace of pruned program
$((l, i), \sigma)$	An execution state, in a trace, where l is a program location, and i is loop counter variable's value
a_c	An array reference $a[\ell + c]$, or a variable introduced for a loop dependence graph to represent all array accesses $a[\ell + c]$ in the loop
ω	An execution state
(l, i)	A *trace-point*, in a trace, where l is a program location, and i is loop counter variable's value
$((l, i), \sigma)_\tau$	An execution state belonging to the trace τ
l_a	Location of the `assert` statement in the program
i_a	Iteration of interest of the property loop
δ_low	Minimum of low values of the spans of all array operands in a program
δ_high	Maximum of high values of the spans of all array operands in a program

δ	$\delta_{\mathsf{high}} - \delta_{\mathsf{low}}$
lcv	Loop counter value
c_{init}	The initial value of the loop counter varaiable of a **for** loop
c_{final}	Value of the loop counter varaiable in last iteration of a **for** loop
c_{ub}	Bound appearing in loop exit condition of a **for** loop
VDG	Variable dependence graph
$\mathcal{L}$	Set of loops in the program
L_a	The property loop containing **assert** statement
$\mathcal{L}_\tau$	Set of loops appearing in a given trace τ of the program
$\mathcal{P}_\mathcal{T}$	A program produced from the sequence of statements corresponding to the trace τ
$\mathcal{P}_{\mathcal{T}_s}$	Program created from $\mathcal{P}_\mathcal{T}$ by replacing all the self-controlling condition tests and all the occurrences of the **assert** statement, except the one at a given trace-point $(l_\mathsf{a}, i_\mathsf{a})_\tau$, by a $SKIP$ statement
$\bar{l}_L$	Location of the first statement in the loop body of a loop L
$\hat{l}_L$	Location of the last statement in the loop body of a loop L
$(\hat{l}_L, \hat{i}_L)$	In a given trace, last trace-point belonging to the loop $L \in \mathcal{L}_\tau$
$live_at(l, i)$	Set of strongly live variables at a trace-point (l, i) in $\mathcal{P}_{\mathcal{T}_s}$
$all_lvas(l, i)$	Set of assignments appearing in the loops that are not dead in the program $\mathcal{P}_{\mathcal{T}_s}$, when considering values of variables that are strongly live at a trace-point (l, i)
(l_L, i_L)	A trace-point belonging a loop $L \in \mathcal{L}_\tau$
$loop_lvas(x, l_L, i_L)$	Set of last value assignments, belonging to a loop L, of a variable x, at a trace-point (l_L, i_L)
$loop_lvas(U, l_L, i_L)$	Set of last value assignments, belonging to a loop L, of a set of variables U, at a trace-point (l_L, i_L)
$loop_lvas_live(l_L, i_L)$	Set of last value assignments, belonging to a loop L, for the set of strongly live variables at (l_L, i_L), with respect to the trace-point (l_L, i_L)
$(\bar{l}_{L_a}, i_\mathsf{a})$	Trace-point at the beginning of the loop iteration, of the property loop, corresponding to $(l_\mathsf{a}, i_\mathsf{a})$
LVA	lcv appearing in the trace-point of a last value assignment

$\mathcal{G}_L$	The VDG of a loop L
$\mathcal{G}$	Graph representing integrated view of the VDGs of all the loops in the program, considering the inter-loop dependence edges
lvas-count	Maximum number of last value assignments needed within the loop to compute a value of a variable belonging to VDG of a loop
$\mathit{lvas_count}(v, \mathcal{G}_L)$	Maximum number of last value assignments needed within the loop to compute a value of a variable v belonging to the VDG of a loop L
$\mathit{lvas_count}(U, \mathcal{G}_L)$	Sum of *lvas-count* of the variables in a set U of variables in $\mathcal{G}_L$
$\mathcal{C}_L$	Sum of product of *lvas-count* and *instance-count* of each node in $\mathcal{G}_L$
$\mathcal{C}$	One more than the sum of $\mathcal{C}_L$ of all the loops L in the program
β	Largest constant index used in array accesses of a program
$\mathcal{K}_{const}$	Largest lcv for a loop with which some array element, with index β, can be accessed inside the loop
$\mathcal{K}^c$	Maximum of $\mathcal{K}_{const}$ of all the loops of the program
$\overline{\mathsf{c}_{init}}$	Minimum of c_{init} across all the loops
Θ	$\mathbf{lcm}$ of steps of all the loops of the program
$\widehat{\mathbb{N}}$	Maximum c_{final} across all the loops
$\overline{\mathbb{N}}$	Minimum c_{final} across all the loops
Γ	The computed loop counter value bound, for the pruned loop, corresponding to the loop with maximum c_{final}
σ_0	Initial state of trace τ
σ_0^p	Initial state of a trace τ^p
loop-LVAs	Set of the LVAs corresponding to the last value assignments $\mathit{all_lvas}(l_{\mathsf{a}}, i_{\mathsf{a}})$
χ	An index map such that for an array element $a[n]$, $\sigma_0^p(a[n]) = \sigma_0(a[\chi(n)])$ and for a scalar variable x, $\sigma_0^p(x) = \sigma_0(x)$
χ^{-1}	Inverse map of χ
$\mathcal{P}^r$	A program created from a trace, as the sequence of all the statements, corresponding to only those trace-points which are either outside a loop or whose lcv corresponds to an LVA. In addition, in $\mathcal{P}^r$, the statements corresponding to initialisation, test and increment of the loop counter variables are replaced by SKIP statement

$\mathcal{P}^r_s$	A program created from a $\mathcal{P}^r$, by replacing all conditional checks by $SKIP$ statement
$LV(\mathcal{P}, (l, i))$	Set of live variables at (l, i) in the trace program $\mathcal{P}$, assuming variables used in the $\mathbf{assert}$ statement at $(l_\mathsf{a}, i_\mathsf{a})$ are live at $(l_\mathsf{a}, i_\mathsf{a})$
$\mathbb{V}_{sc}(l)$	Set of self-controlled variables at the location l in a program
$RLV(\mathcal{P}^r, (l, i))$	For a given trace-point (l, i), if l is outside any loop then it is same as $LV(\mathcal{P}^r, (l, i))$, else it is $(LV(\mathcal{P}^r, (l, i)) \setminus \mathbb{V}_{sc}(l)) \cup LV(\mathcal{P}^r_s, (l, i))$
$\omega' \preceq \omega$	Execution state ω' is *modulo-index equivalent* to execution state ω, where ω' and and ω are from a trace of pruned program and original program, respectively
$[\![e]\!]_\sigma$	Value of expression e, when evaluated in a state σ
first	Location of the loop exit condition check of a loop
last	Location of the loop counter increment operation of a loop
$\mathit{firstst}(\tau, i)$	Exceution state $((\mathit{first}, i), \sigma)$ in the trace τ
$\mathit{lastst}(\tau, i)$	Exceution state $((\mathit{last}, i), \sigma)$ in the trace τ

Chapter 1

Introduction

Software touches the lives of almost every human being in a variety of ways. On one hand it is the backbone of large commercial systems like banking, and plays a key role in the management of large scale enterprises like the Indian Railways. On the other, it is present in personal devices like mobile phones and household gadgets like washing machines, as well as in large mechanical systems like automobiles and aircrafts. Obviously, any malfunctioning of such systems will also adversely affect human beings in varying measures.

The impact of malfunctioning of software can vary from being mild, like mere annoyance, to very severe, e.g. huge financial losses or loss of human lives. History has several such instances of severe losses due to bugs in software systems. An illustrative compilation of such failures and resulting losses can be found in work by Huckle [47] and at wikipedia [73]. Here are a couple of well-known examples. In January 1987, a computer controlled radiation therapy machine, called the *Therac25*, massively overdosed six people due to a software bug and three people lost their lives. A decade later, in September 1997, the Arian 5 rocket of European Space Agency exploded just after 36.7 seconds of its launch due to a bug in the software. It had taken 10 years and $7 billion to build the rocket.

Many of these failures happen because some bugs escape the testing cycle of the software development, particularly the ones that are hard to reproduce. For example, the Arian failure was due to the truncation of a large computed value which could not fit in the storage allocated for the variable storing the computed value. In the case of Therac25, the mishap was due to an 8 bit counter wrapping around to value zero when incremented after it had value 255, whereas

the expected count after the increment was 256. Therefore, verifying or proving the programs correct before deployment becomes crucial, particularly for safety and mission critical systems. This is done through the use of techniques that are collectively termed as static analysis. However, proving a program correct is very hard, if not impossible, particularly for large systems. Therefore, the correctness or safety of the system is described as a collection of properties which a program must satisfy for it to be acceptable with respect to a set of critical and safety requirements. For example, a critical property for a banking system could be that a savings account balance will never be negative. Or for a rail road crossing system, it could be that when the gate is open no train is allowed to pass. Given such a set of properties, programs are anlaysed to check if these properties indeed hold. Such properties are called *safety properties*.

Property checking has been a well studied area and there has been lot of development in tools and techniques ranging from abstract interpretation [26, 25, 65] to predicate abstraction [23, 5, 4, 43], symbolic model checking [21, 59, 64], and bounded model checking [22]. However, when it comes to proving properties on real life large programs, most of these tools and techniques are found wanting in scalability. This happens mainly due to two reasons: (1) the sheer size of the programs, and (2) the presence of loops, recursion, and arrays. In this thesis, we describe two approaches: the first helps in tackling challenges posed by large size of the program to be property checked, and the other handles the complexity arising out of the use of large arrays that get processed by loops having bounds of the same order as the size of these arrays.

1.1 Scalability and precision in property checking

There is almost always a trade-off between scalability and precision. As most of the techniques for property checking have some kind of abstraction or approximation at their core, a very coarse abstraction will enable a very high scalability but percentage of false positive results will be too high to be of any use. For example, techniques based on abstract interpretation (e.g. ASTREE [27], POLYSPACE [58]) are highly scalable with coarser abstract domains like interval domain. But with coarse abstraction, one loses information and this results in imprecision. This results in a large number of false positives, i.e., although the property to be checked holds, the technique is not able to verify it. On the other hand, more precise techniques like those based on

predicate abstraction or model checking (e.g. CBMC [22], SATABS [23], SLAM [5]) are able to verify more properties but do not scale to large or complex programs containing, for example, loops and arrays. Therefore, when we shall refer to scalability, we shall also have an acceptable level of precision in mind i.e. scalability with precision. Furthermore, the size, as a measure of scalability, can refer to the physical size of a program as well as the number and size of loops and arrays used in it. While several methods exist that achieve scalability with precision, these do not cover the entire space of programs, and the problem still remains challenging for newer techniques to fill the space.

1.2 Some motivating examples

We now present a couple of examples that illustrate the challenges of scalability faced even by state-of-art verifiers. However, in doing so, we also give a hint to the reader that it might be possible to address the challenges in certain situations. Consider the C program given in Figure 1.1 (a). For brevity, we have not shown the body of functions `fn1`, `fn2` and `fn3`. Assume that none of these functions have any side effects. In addition, assume that the functions `fn1` and `fn2` are large and complex. The program has an `assert` involving the variables `u` and `k` at line 15. It is obvious that, depending upon the value of `st` assigned at line 4, if line 11 is executed in some iterations of the while loop then line 13 is never executed; and similarly, if line 13 is executed in some iterations of the while loop then line 11 never gets executed. Therefore, the value of `j` and `k` will either be same (when line 11 is executed), or value of `j` will be twice of value of `k` (when line 13 is executed). Therefore the `assert` at line 15 is always satisfied. However, SATABS (version 3.0) [23], a robust and scalable predicate abstraction based property checking tool, times out on this program even when a limit of 20 minutes is given. We observed that it gets entangled in its refinement cycle due to the large and complex code present in functions `fn1` and `fn2`. This was further established when we made the functions `fn1` and `fn2` very simple and the tool was able to verify the property within a few seconds. Observe, however, that the value of `u` does not depend on the values of `i` or `t` which are only used in the *while* condition at line 5 and the *if* conditions at line 8 and 9. Since condition at line 8 influences only modification of variable `l`, this condition is not relevant to the `assert` at line 15. The other two conditions, at lines 5 and 9, only control whether the

```
int main() {
 int i,j,k,st,t,l,u;
 t=i=j=k=l=0;
 st = fn3();
 while (i<1000) {
  i= i+ fn2();
  t = fn1(i,j);
  if (t>10) l++;
  if (t>100) {
   if (st ==1)
    { j++; k++; }
   else
    {j+=2; k+=1;}
   u = j-k;
   assert(u==0||u==k);
  }
 }
 return 0;
}
int fn1(),fn2(),fn3();
```

(a) Large and complex program

```
#define N 7
main()
{
 int i, m;
 int a[N]={8,4,6,2,11,2,2};
 m = a[0];
 i=0;
 while(i < N)
 {
  if(m >= a[i]-1)
     m = a[i];
  i++;
 }
 assert ∀j ∈ [0..N − 1].(m ≤ a[j]);
}
```

(b) Program with use of arrays

Figure 1.1: Motivating examples to illustrate scalability issues

assert is reachable or not. As a result, the variables i and t merely decide the reachability of line 15 during an execution. Therefore, the statements computing these two variables do not affect the value of u or k, and may be considered irrelevant. As a result, the call to the functions fn1 and fn2 can be sliced out to achieve scalability and the verifier would still be able to verify the property.

We now illustrate the problems that arise due to use of arrays. Consider the program in Figure 1.1(b). The loop in the program purportedly computes a variable m, the minimum element, denoted min, of an array a. However, due to a programmer error at line 10 (a[i]-1 instead of a[i]), the program actually computes the last value in the longest subsequence[1] a[i_1], a[i_2],..., a[i_p] of the array, such that a[i_1]$= min$, and for any two consecutive elements

[1]Note that a subsequence of a given array is obtained by deleting zero or more elements of the array. So a subsequence is not necessarily a sub-array.

a[i_k] and a[i_{k+1}] of the subsequence, a[i_{k+1}] $\leq$ a[i_k]+1. [2] Although, for simplicity of exposition, we have shown a small size array, the issues explained here will be amplified when the array size is large. Notice that for ease of exposition, we have used universal quantification in the assert expression to express the property; in reality, a loop will be used instead. The property holds for the example because the longest subsequence of the array with the stated properties is {2,2,2}, and the last element (2) happens to be the same as min. However the assertion will fail if, for example, the last two elements of the array are changed to 3 and 5, so that the longest subsequence is now {2,3}. The last element of this subsequence is no longer the minimum of the array.

Abstraction based verifiers as well as bounded model checkers fail to verify this program when the array size is increased to 1000. For example, CBMC 5.8 [22] reports "out of memory" when it is run with an unwinding count of 20. Abstraction based verifiers like SATABS 3.0 [23] and CPAchecker 1.6 [10] keep on iterating in their abstraction refinement cycle in search of an appropriate loop invariant, until they run out of memory. Therefore, it is worthwhile to look for an abstraction of the property checking problem for array processing loops that can be verified by a bounded model checker (BMC).

1.3 Our thesis

In this thesis, we present ideas which address the problems of scalable property checking arising out of two distinct issues. The first issue arises in programs in which some complex fragments of code may not be relevant to the property being checked, but their presence is hindrance to the verifiers in verifying the property. The second issue arises in programs that process arrays, and thus the property being checked is often quantified. In this class of programs we are interested in programs where array sizes are very large, and as a result, the loops processing them also have very large bounds.

Given a program P, and a property ψ, our approach in both the problems considered is to get a program P' and property ψ', such that if ψ' holds in P' then it guarantees that ψ will hold in P. The first problem, arising due to the presence of complex code fragments that are irrelevant to the property being checked, is demonstrated earlier by the example in Figure 1.1(a).

[2]There is a unique such subsequence for a given array.

In this case ψ is the same as ψ', and the program P' is obtained from P after identifying and eliminating the code fragments that are irrelevant (with respect to reachability) to the property ψ. We call the program P' a *value slice*.

The issues arising due to the presence of large arrays and consequently the presence of large loops is shown by the example in Figure 1.1(b). An array processing loop is a common occurrence in a program, and a guarantee of reliability often requires the program developer to prove properties that are quantified over the elements of the array being processed. This is, in general, difficult because such programs have a very large, at times infinite[3], state space. So, while static analysis techniques like smashing and partitioning [13, 14, 41, 42, 28, 33] fail due to abstractions that are too coarse, attempts with bounded model checkers or theorem provers armed with array theories [22, 10, 23, 60, 61, 49, 37] tend to fail for lack of scalability or their inability to synthesize the right quantified invariant.

In certain situations, the decidability of property-checking of finite state programs can be used to prove properties of infinite state space programs. As part of this thesis, we present two such transformations for programs that process arrays using loops. The property ψ is usually a $\forall$ or an $\exists$ property over the elements of the array, but it can also be a property over scalar variables modified in the loop. In the first transformation, called *loop shrinking*, P' is a program in which we replace loops that manipulate an array of possibly large or even unknown size with smaller loops that operate only on a few non-deterministically chosen elements of the array. In the second transformation, called *loop pruning*, we replace array manipulating loops with smaller loops that operate only on the first few elements of the array. The property ψ' is also changed accordingly for expressing the property over the smaller array.

In subsequent three sections, we give a brief of our ideas about *value slice*, *loop shrinking*, and *loop pruning*.

1.3.1 Scalable property checking using value slice

Given a program and a set of variables at a program point of interest, *program slicing* [71] pares the program to contain only those statements that are likely to influence the values of the vari-

[3] The infinite state space can happen when a loop is unbounded because of a non-deterministic loop exit criterion.

ables at that program point. The set of variables and the program point, taken together, is called the *slicing criterion*. Several variants of the original slicing technique, called *backward slicing*, have since been proposed [67]. These have been used for program understanding, debugging, testing, maintenance, software quality assurance and reverse engineering. A survey of applications of program slicing appears in a survey paper by Binkley et al. [12]. However, our interest is in use of some form of slicing for scaling up property checking.

Among slicing techniques, backward slicing is the natural choice to reduce the program size to enable scale up of the property checking techniques. While computation of backward slice is efficient and scalable, the size of the slice is a matter of concern. Empirical studies [38] have shown that the size of the backward slice, on an average, is about 30% of the program size. This size is still too large for analysis of large programs. In addition, the statements sliced out are irrelevant to the asserted property and their elimination does not reduce the load on the verifier significantly. To remedy this, we propose an alternate notion of slicing (called *value slice*), which eliminates additional parts of the program without affecting verifiability. Our idea is based on the observation that a backward slice consists of two categories of statements: (i) statements that decide whether the slicing criterion will be reached during execution, and (ii) statements that decide the values of variables in the slicing criterion. Our experience shows that, in significant number of cases, the first category of statements have no bearing on property in question, and that the second category of statements, called *value-impacting* statements, are often enough for property checking.

Earlier, a similar slicing technique called *thin slicing* [68] was proposed in the context of program debugging and understanding. The authors claimed that, to find the cause of a defect, it is often sufficient to look at only those statements on which the variables in the slicing criterion, derived from the defect observation, are data-dependent. In particular, *all* conditional statements are eliminated. It is possible to argue that, since defects are manifestation of some property violation, one can use thin slices to scale up the property checking techniques. While this idea does bring down the size of the resulting slice, unlike our method it also eliminates some conditional statements that are value-impacting and thus crucial for property checking.

To illustrate this point, consider the program in Figure 1.1(a). As we mentioned earlier, the functions `fn1` and `fn2` are large and complex but without side effects. The backward slice of the program, with the slicing criterion $\langle 15, \{\texttt{u,k}\}\rangle$, is shown in Figure 1.2(a). Clearly, the

```
1  int main() {
2   int i,j,k,st,t,u;
3   t=i=j=k=0;
4   st = fn3();
5   while (i<1000) {
6    i= i+ fn2();
7    t = fn1(i,j);
8
9    if (t>100) {
10    if (st ==1)
11     { j++; k++; }
12    else
13     {j+=2; k+=1;}
14    u = j-k;
15    assert(u==0||u==k);
16   }
17  }
18  return 0;
19 }
20 int fn1(),fn2(),fn3();
```

(a) Backward slice

```
1  int main() {
2   int j,k,st,u;
3   j=k=0;
4   st = fn3();
5   while (*) {
6
7
8
9    if (*) {
10    if (st ==1)
11     { j++; k++; }
12    else
13     {j+=2; k+=1;}
14    u = j-k;
15    assert(u==0||u==k);
16   }
17  }
18  return 0;
19 }
20 int fn3();
21 // fn1 and fn2 removed
```

(b) Value slice

```
1  int main() {
2   int j,k,u;
3   j=k=0;
4
5   while (*) {
6
7
8
9    if (*) {
10    if (*)
11     { j++; k++; }
12    else
13     {j+=2; k+=1;}
14    u = j-k;
15    assert(u==0||u==k);
16   }
17  }
18  return 0;
19 }
20 // fn3 removed
21 // fn1 and fn2 removed
```

(c) Thin slice

Figure 1.2: Usual backward slice, value slice and thin slice

backward slicing will eliminate only line 8, as computation of variable `l` has no influence on values of `u` and `k` at line 15. In particular, in the backward slice, calls to function `fn1` and `fn2`, at lines 7 and 6 respectively, will be retained. This is because line 9 and line 5 both have to be part of backward slice. Line 15 is control dependent on *if* condition at line 9, which subsequently is control dependent on the condition of *while* loop at line 5. Since condition at line 9 has variable `t`, which gets its value from assignment at line 7, line 7 also will be part of the slice. Similarly, variable `i` in condition at line 5 gets its value from assignment at line 6 and so line 6 also will be part of the backward slice. As a result, just backward slicing is not going to be of any use in helping tools like SATABS (version 3.0) [23] to succeed in verifying property of this program. However, observe that the value of `u` does not depend on the values of `i` or `t`. Since these variables merely decide the reachability of line 15 (through conditions

at lines 9 and 5) during an execution, the statements (at lines 6 and 7) computing them are non-value-impacting and thus may be considered irrelevant.

Figure 1.2(b) shows a slice of the program that captures the computation of every value of `u` in the original program. Conditional statements that do not affect the value of `u` are replaced by a symbol $\star$, standing for a randomly chosen boolean value. The resulting slice is much smaller in comparison to the backward slice. SATABS succeeds in showing that the property is indeed satisfied on the sliced program, and, by implication, on the original program.

On the other hand, the thin slice for the same program with respect to same slicing criterion will have no conditions from the program as it retains only those statements that directly or transitively provide values of variables of slicing criterion. It is easy to see that the program shown in Figure 1.2(c) will be the resulting thin slice. The thin slice, although, is smaller in size, it is not useful since the property does not hold on the sliced program. This is because now in some iterations line 11 can be executed while in others line 13 may get executed. Thus, any verifier will produce counterexamples on this slice and that will be spurious on the original program.

Results from our experiments show that both value and thin slice help in scaling up property checking, with thin slice having a small advantage (14%) over value slice. However, compared to the backward slice, the precision drops considerably (29%) in the case of thin slice, while there is only a marginal drop (2%) for value slice. This implies that refinement will be required in many more cases with thin slice as compared to value slice. Therefore, as a slicing technique for increasing the scalability of property checking, value slice represents a sweet spot between backward and thin slice.

1.3.2 Checking quantified properties using loop shrinking

In this section, we present a brief summary of loop shrinking which abstracts a large array processing loop to a loop with a smaller bound that replays a number of non-deterministically chosen sequence of iterations of the original loop. As an example, consider the program of Figure 1.1(b) which we have reproduced in Figure 1.3(a). As illustrated earlier, the loop in the program computes the last value of the longest subsequence $a[i_1]$, $a[i_2]$,..., $a[i_p]$ of the array, such that $a[i_1] = min$, and for any two consecutive elements $a[i_k]$ and $a[i_{k+1}]$ of the

```
#define N 7
main()
{
 int i, m;
 int a[N]={8,4,6,2,11,2,2};
 m = a[0];
 i=0;
 while(i < N)
 {
  if(m >= a[i]-1)
    m = a[i];
  i++;
 }
 assert ∀j ∈ [0..N − 1].(m ≤ a[j]);
}
```

(a) Concrete program

```
#define N 7
main() {
 int i, m, a[N]={2,4,6,8,11,2,2};
 unsigned li, it[2];
 m = a[0]; i=0;
 it[0]=nondet(); it[1]=nondet();
 assume(1 <= it[0] && it[0]<it[1]);
 for (li=0; li < 2 ; li++) {
  i = it[li] - 1;
  if (!(i < N)) break;
  if(m >= a[i]-1) m = a[i];
  i++;
 }
 assume(li==2);
 assert ∀t ∈ it.(m ≤ a[t − 1]);
}
```

(b) Abstract program

Figure 1.3: Loop shrinking abstraction illustration

subsequence, $\texttt{a}[i_{k+1}] \leq \texttt{a}[i_k]+1$. Notice that for ease of exposition, we have used a universal quantification in the assert expression to express the property; in the actual program, a loop will be used instead.

Observe that in this program, the assertion will hold if and only if, after the last index containing the minimum value min, no other index in a contains the value $min + 1$. This can be conservatively checked by examining for each pair of array indices, say k and $k + j$, $j > 0$, whether $\texttt{a}[k + j] = \texttt{a}[k] + 1$. The computation is effected by selecting a pair of indices non-deterministically and executing in sequence the loop body with the loop index i first instantiated to k and then to $k + j$. The resulting value of m can then be checked for the condition $\texttt{m} \leq \texttt{a}[k] \wedge \texttt{m} \leq \texttt{a}[k + j]$. As we shall see later, it is helpful to think in terms of iteration numbers instead of array indices; the correspondence between the two for the present example is that the value at index i of the array is accessed at iteration number $i + 1$.

In other words, we compute m for every pair of iterations of the loop, and check if m satisfies the property for the chosen iterations. For example, the value of m computed for the iterations numbered 2 and 3 of the loop is 4, and the property restricted to these two iterations,

$\mathtt{m} \leq \mathtt{a}[1] \wedge \mathtt{m} \leq \mathtt{a}[2]$, is satisfied. On the other hand, if we change the last two elements to 3 and 5 then the property fails for the original program. However, we can now find a pair of iterations, namely 4 and 6, such that value of m calculated on the basis of just these two iterations will be 3, and it will not satisfy the corresponding property $\mathtt{m} \leq \mathtt{a}[3] \wedge \mathtt{m} \leq \mathtt{a}[5]$, since $\mathtt{a}[3]$ is 2. In summary, if executing the loop for every sequence of two iterations $[i_1, i_2]$, $i_2 > i_1$, establishes the property restricted to these iterations, then the property will also hold for the entire loop. Read contrapositively, if the given program does not satisfy the assertion, then there must be a sequence of two iterations for which the property will not hold. This is true irrespective of the size or the contents of the array in the program. Loops that exhibit this feature for iteration sequences of length k (k is 2 in this example) will be called *shrinkable loops with a shrink-factor* k.

We create a second program, shown in Figure 1.3(b), that over-approximates the behaviour of the original with respect to the property being checked. The `while` loop is substituted with a loop that executes the non-deterministically chosen iteration sequence stored in the two-element array `it`. The `while` loop in the original program, schematically denoted as `while` (C) B, is replaced by a `for` loop that is equivalent to the unrolled program fragment `i=it[0]-1;if(`C`)`$\{B$`;i=it[1]-1;if(`C`)` $B\}$. We call this `for` loop (or its unrolled equivalent) the *residual loop* for the iteration sequence `it`. The `break` statement ensures that the chosen iteration numbers do not result in an out-of-bounds access of the array, and the `assume` statement ensures that exactly two iterations are chosen. Similarly, the given property is also substituted by a *residual property* quantified over array indexes corresponding to the same chosen iteration sequence. CBMC is able to verify the property on this transformed program as the original loop, even with a changed bound of 1000, is now reduced to only two iterations. We call this method *property checking by loop shrinking*. Needless to say, the method can only be applied to a program if its loops are shrinkable with a known shrink-factor. We develop a method to determine both using a BMC (bounded model checker).

We have implemented the approach in a tool called *VeriAbs* [19]. Like other abstraction based approaches, the loop shrinking approach is also sound but not complete. Therefore, for a verification problem, the implementation produces one of the three verdicts: (a) *program is safe*, (b) *program is unsafe*, and (c) *unknown* (can not say whether the program is safe or unsafe). The tool VeriAbs competed in the SV-COMP 2017 verification competition [8], conducted by

```
1  #define N 100000
2  int main()
3  {
4   int a[N];
5   int i, min1, min2;
6   min1 =a[0];
7   for (i=1; i< N; i++)
8     if (min1 > a[i])
9       min1 = a[i];
10  min2 = a[0];
11  for (i=1; i< N; i++)
12      a[i-1] = a[i];
13  for (i=0; i< N-1; i++)
14    if (min2 > a[i])
15      min2 = a[i];
16  assert (min1==min2);
17 }
```

(a) Concrete program

```
1  #define N 100000
2  int main()
3  {
4   int a[N];
5   int i, min1, min2;
6   min1 =a[0];
7   for (i=1,j=0; i< N,j<1; i++,j++)
8     if (min1 > a[i])
9       min1 = a[i];
10  min2 = a[0];
11  for (i=1,j=0; i< N,j<1; i++,j++)
12      a[i-1] = a[i];
13  for (i=0,j=0; i< N-1,j<1; i++,j++)
14    if (min2 > a[i])
15      min2 = a[i];
16  assert (min1==min2);
17 }
```

(b) Abstract program

Figure 1.4: Loop pruning abstraction illustration

ETAPS, where it was ranked third amongst the 17 participating tools in the *ArraysReach* category. Out of total 135 programs, 91 programs were found to have shrinkable loops and among these 65 were safe programs. Our tool could verify 64 of these safe programs. Only for one safe program it could not say that it is indeed safe, and gave an *unknown* verdict. In the SV-COMP 2018 verification competition [9], our tool ranked first in the same category. Out of total 167 programs, 117 programs were found having shrinkable loops and among these 86 were safe programs. Our tool could verify 81 of these safe programs. Only for five programs it gave an *unknown* verdict.

Thus, the central idea demonstrated through this approach is that over-approximation using shrinkability is an effective technique to verify properties of programs that iterate over arrays of large or unknown size.

1.3.3 Checking quantified properties using loop pruning

In this section, we give an introduction to the idea of loop pruning which abstracts large array processing loops with loops having smaller bounds. While the iterations in the smaller loop were chosen non-deterministically in the previous method, in loop pruning we choose iterations that replay the first few iterations of the original loop. As an example, consider Figure 1.4(a). This program is adapted from a benchmark program of SV-COMP 2018 verification competition [9]. In this program, the minimum of an array is computed in the first loop, and in second loop the elements are shifted to left by one position and again a new minimum is computed for the modified array. Let the values of array elements after the shifting loop be `a′[0]`, `a′[1]`, ..., `a′[N-1]`. Obviously, for $0 \leq i < \texttt{N} - 1$, `a′`$[i]$ = `a[i+1]`. The computed value of `min1` is `min(a[0],a[1],...a[N-1])` and that of `min2` is `min(a[0], a′[0], a′[1], ...,a′[N-2])`. It readily follows that `min1 = min2` and therefore the property checked at the end is correct. As observed in the previous section, for this program also, the verifiers (e.g. CBMC) fail to verify the property when the array is of large size.

However, a closer examination of the program suggests a different insight. Suppose we transform this program such that, the computation of minimum in the loops and shifting to left is limited to only one iteration. We claim that for every run of the original program, there will be a run in transformed program in which value of `min1` and `min2` will be same as that in the given run of original program. This is because in the transformed program `min1` = `min(a[0],a[1])`, and `min2` = `min(a[0],a′[0])` = `min(a[0],a[1])`. Since values of arrays are non deterministic, we can have a run of transformed program with values of `a[0]` and `a[1]` such that `min1` and `min2` have values as that in the run of original program. And therefore, if the property holds in transformed program, it will hold in original program too.

A transformed version of the program is shown in Figure 1.4(b). In this program each loop is iterated only once. As a result `a[1]` is accessed in the first loop, `a[0]` and `a[1]` are accessed in the second loop. Element `a[1]` is accessed in third loop as `a′[0]` because value of `a[1]` is transferred to `a[0]` in second loop which gets accessed in third loop as `a′[0]`. Now, `min1` is `min(a[0],a[1])` and `min2` is also `min(a[0],a[1])` because `a′[0]` = `a[1]`. Hence, the property holds in transformed program.

CBMC is able to verify the property on this transformed program successfully as the original loops, having bound of 100000, are now reduced to only one iteration. We call this method *property checking by loop pruning*. In general, for the method to be applicable to a program, it is important to determine under what circumstances would this transformation be a safe approximation. In other words, we identify conditions under which the following holds: if a run of the original program produces certain values of the variables in the property checking assertion, we can produce the same values in some run of transformed program. We will present a criterion for the same and define the transformation. We give a proof that for a program satisfying the criterion, the transformed program is a safe approximation. We will also present two methods: (1) to check if a given program satisfies the criterion, and (2) to transform the program according to the transformation defined when the program satisfies the criterion.

We have implemented this approach also in the latest version of the tool *VeriAbs* that competed in the SV-COMP 2018 verification competition [9] in the *ArraysReach* category, where it was ranked first amongst the 13 participating tools. Out of total 167 programs, 48 programs were such that the loop pruning approach could be applied, and the tool could verify all of them correctly. Out of these 48 programs, 23 programs did not have shrinkable loops, and so only loop pruning approach could verify them.

Thus, the central idea demonstrated through this approach is that, for certain kind of programs, over-approximation using loop pruning is an effective technique to verify properties of programs that iterate over arrays of large size. Like the approach of shrinkable loops, this technique is also impervious to the size of loop bounds—increasing the loop bound does not cause an otherwise verifiable program to timeout.

1.3.4 Choice of base verifiers

Based on the robustness of available verifiers, we zeroed on two of them: SATABS [23] and CBMC [22]. While SATABS is based on predicate abstraction, CBMC uses bounded model checking. Since CBMC uses bounded model checking, its applicability is limited in presence of loops with large bounds, because then the soundness can not be guaranteed for property verification. However, when loops have small bounds then we observed that CBMC is much more effective in comparison to SATABS. This is because CBMC can afford to unroll the loops

up to their complete bound and guarantee soundness in verification for such programs.

In case of scalability using value slice, we are not abstracting the loops, and therefore loops in the sliced program appear as they appear in the original program. Since these loops may have large bounds, we use SATABS as base verifiers on value slices. On the other hand, to address the issues arising due to loops with large bound, we abstract the loops with the ones having a small bound. Since CBMC is very effective for programs having loops with small bounds, we use CBMC as base verifier on the resulting abstract programs.

1.3.5 A summary of the contributions

The contributions of this thesis are as follows.

1. We define a new notion of slicing called value slice and propose a worklist based algorithm for its computation. The algorithm is shown to be correct by construction.

2. We describe the results of experiments on property checking based on the three comparable slicing methods—backward, value and thin slices. We show that on both criteria, scalability and precision, value slice based property checking yields results that are close to the best among the three slicing methods.

3. We introduce and formalize a concept called *shrinkability* for loops that process arrays. We show formally that a shrinkable loop, with shrink-factor k, can be over-approximated by a loop that executes only k non-deterministically chosen iterations.

4. We provide an algorithm that uses a BMC to find the shrink-factor k for which the loop is shrinkable.

5. We describe an implementation of the proposed abstraction and report experimental results showing the effectiveness of the technique on SV-COMP 2017 [8] benchmarks in the *ArraysReach* category.

6. We present a criterion and a transformation which executes only first few iterations of the loops and formally show that programs satisfying the criterion can be safely approximated using the defined transformation.

7. We present a method to find the prune factor to be used in the transformation.

8. We describe an implementation of the proposed abstraction and report experimental results showing the effectiveness of the technique on SV-COMP 2018 [9] benchmarks in the *ArraysReach* category.

Our contributions are reported in proceedings of TACAS-2015 [56] for scalability through value slicing and in proceedings of TACAS-2018 [57] for handling quantified property checking in programs manipulating large arrays. The two techniques: loop shrinking and loop pruning, have been implemented in the tool VeriAbs that participated in the two editions of the SV-COMP competition: SV-COMP 2017 [8] and SV-COMP 2018 [9]. The details of tool VeriAbs are reported as competition contribution for SV-COMP 2017 [19] and SV-COMP 2018 [30].

1.4 Related work

Backward slicing was first introduced for imperative programs by Mark Weiser [71] in 1981. Later, Ferrante et al. introduced a representation of the program called *Program Dependence Graph* (PDG) [36], and backward slicing was modeled as a reachability problem over the PDG. To address the issues of feasible paths arising due to procedure calls, Reps et al. introduced a representation of programs called a system dependence graph (SDG) [46] that spans over more than one functions. A survey of the significant variants of backward slicing proposed since its introduction by Weiser can be found in the survey work of Silva et al. [67]. Notable among these variants are forward slicing [7], chopping [48], and assertion based slicing [16, 24, 6]. The backward slice and its variants have been used for program understanding, debugging, testing, maintenance, software quality assurance and reverse engineering. A survey of the applications of program slicing appears in the survey work of Binkley et al. [12].

Comuzzi and Hart [24] introduce a shift from syntax based slicing to program semantics based slicing by defining a form of program slicing, called *p-slices*. Given a predicate which they consider as a slicing criterion, they use Dijkstra's weakest precondition (wp) to compute the slice. More specifically, for a given program P and a predicate ψ, a subprogram S of P is a *p-slice* if $wp(P, \psi) \equiv wp(S, \psi)$. The extension by Barros et al. to *specification based slicing* [6] generalises the earlier method by taking both the precondition and the post condition together as the slicing criteria. However, computing such slices can turn out to be as hard as

the verification problem itself. Therefore, their usefulness in easing the program verification problem is limited.

All these techniques produce slices with behaviour equivalent to the original program with respect to the slicing criterion. However, to the best of our knowledge, the idea of producing slices which are not exactly equivalent to original program for the selected slicing criterion has not been explored at depth. Our interest in the slicing is more to reduce the program size so that property checking can scale up, as irrelevant portion of the code can be eliminated. At the same time, we want to retain the code which affects the property. *Thin slicing* [68], proposed by Sridharan, Fink and Bodik, with the aim to help debugging, is the first approach that produces a slice whose behaviour may differ from the original program with respect to the slicing criterion. A thin slice retains only those statements that the variables in the slicing criterion are data dependent on and abstracts out *all* the predicates. This approach comes closest to our method as it is based on static analysis and produces an aggressive slice. While a thin slice is indeed much smaller in size compared to the backward slice, our experiments show that the thin slice is too imprecise to be useful for property checking. In contrast, although a value slice is a little larger than the thin slice, it is still much smaller than the backward slice and, more importantly, comes with only a slight loss in precision when compared to the backward slice. We present value slice as a good compromise between precision and scalability. To substantiate the claim, we compared value slice with backward slice and thin slice. This comparison is in no way intended to undermine the usefulness of thin slice in debugging and program understanding, for which it was designed in first place.

The various approaches to handle arrays have their roots in the types of static analyses used for property verification, namely: abstract interpretation, predicate abstraction, bounded model checking and theorem proving. In abstract interpretation, arrays are handled using *array smashing*, *array expansion* and *array slicing*. In array smashing, all the elements of an array are clubbed as a single anonymous element, with writes to the array elements treated as weak updates. As a result, the abstraction becomes too coarse and imprecise. It cannot be used, for example, to verify the motivating example in Figure 1.1(b). In array expansion, the array elements are explicated as a collection of scalar variables, and the resulting programs have fewer number of weak updates than array smashing. However, it works well only for small-sized arrays. In array slicing [41, 42, 28], the idea is to track partitions of arrays based upon some

criteria inferred from programs. Each partition is treated as an independent smashed element. Dillig et al. [33] further refined the approach by introducing the notion of *fluid updates*, where a write operation may result in a strong update of one partition of the array and weak update of other partitions. In contrast to these approaches, our abstraction is based not only on the program elements but also on the associated property. By declaring an array-processing loop as k-shrinkable, we guarantee that an erroneous behaviour of the program with respect to the property can indeed be replayed on some k elements of the array.

There has been considerable work in over-approximation or under-approximation of the loops for verification or bug finding. Daniel et al. [55] under approximate loops using acceleration to find deep bugs in the programs. Darke et al. [29], abstract loops and apply acceleration to verify the property. However, these efforts have been restricted to loops computing only scalar variables. For example, Darke et al. in their work on loop abstraction [29], consider only those loops which do not contain operations on arrays.

Methods based on predicate abstraction go through several rounds of refinement where, in each round, a suitable invariant is searched based on the counterexample using *Craig interpolants* [59]. Tools like SATABS [23] and CPAchecker [10] are based on this technique. To handle arrays, the approach relies on finding appropriate quantified loop invariants. However generating interpolants for scalar programs is by itself a hard problem. With the inclusion of arrays, which require universally quantified interpolants, the problem becomes even harder [51, 60]. Our method, in contrast, does not rely on the ability to find invariant. Instead, we find a bound on the number of loop iterations and, in turn, the number of array elements that have to be accessed in a run of the abstract program.

Theorem proving based methods (e.g. Vampire [45]) generate a set of constraints, typically as Horn clauses. The clauses relate invariant at various program points and the invariants are predicates over arrays. The constraints are then fed to a solver in order to find a model. However, these methods also face the same difficulty of synthesizing quantified invariants over arrays. A technique called *k-distinguished cell abstraction* [61] addresses this problem by abstracting the array to only k elements. A 1-distinguished cell abstraction, for example, abstracts a predicate $P(a)$ involving an array a by $P'(i, a_i)$, where i and a_i are scalars. The relation between the two predicates is that $P'(i, a_i)$ holds whenever $P(a)$ holds and the value of $a[i]$ is a_i. The resulting constraints are easier to solve using a back end solver such as Z3 [32]. However,

there are cases when our method of loop shrinking is able to verify the property but a tool, VAPHOR, based on *k-distinguished cell abstraction* either times out or wrongly says that program is incorrect.

As far as we know, there is no work on just pruning the loop bound without abstracting the loop body, like we do in loop pruning. As mentioned earlier, all of the work dealing with loops and arrays abstract the loop body in one form or the other.

1.5 Organization

In this thesis, we present two pieces of work to tackle the scalability issue of property checking. The rest of the thesis is in three parts. Part I describes value slicing, and Part II describes the two approaches to handle programs processing large arrays. The two parts are independent of each other and can be read in any order. In order to keep them independent, some background concepts are included in both parts. The concluding remarks are presented in Part III, and they are applicable to both the parts appearing earlier in the thesis.

Part I which deals with the scaling up property checking using value slice is organised as follows. In Chapter 2, we describe the program model that falls within the scope of our approach and basic concepts related to program slicing like data and control dependence. We discuss the new concept of value slice and the algorithm to compute it in Chapter 3. Its implementation and experimental results are presented in Chapter 4. In Chapter 5, we discuss related work and compare them with our solution.

In part II, we present two approaches to tackle issues arising due to processing of large arrays and checking quantified properties over these arrays. This part is organised as follows. Chapters 6 to 8 cover the first approach, *loop shrinking*, and Chapter 9 covers the second approach, *loop pruning*. Within this part, Chapter 6 describes the kinds of programs and properties that are in the scope of the method, as well as background concepts such as loop acceleration and the use of Hoare triples as a means of representing the property checking problem. We present the idea of loop shrinking in Chapter 7. In the same chapter, we describe how to determine if a loop is shrinkable, and how to make use of a shrinkable loop to scale up property checking of array processing programs. In Chapter 8, we present our implementation and experimental results. Chapter 9 covers the entire topic of loop pruning, its implementation, and reports ex-

perimental results. In Chapter 10, we describe existing approaches to solve this problem and how our approach compares with them.

Part III has a single chapter with concluding remarks and suggested directions for further research.

Part I

Scaling up Property Checking Using Value Slice

Chapter 2

Background

In this chapter, we describe the basic concepts and terminology required to present our idea on scaling up property checking using value slice. We shall present our ideas in the context of imperative programs consisting of assignments, conditional statements, and *while* loops. Further, we assume that the expressions in these programs are side-effect free. We allow *break* and *continue* statements in the loops. However, we restrict ourselves to single procedure and goto-less programs with single-entry loops and two-way branching conditional statements; it makes for an easier formal treatment of our method without losing expressibility. For the same purpose, we assume that there are no dynamic allocations in the program. Please note that these restrictions are only to keep the presentation of our ideas simple. In actual implementation, as explained later in chapter 4, we allow the full fledged C language programs.

2.1 Control flow graph

Our analysis will be based on a model of the program called the control flow graph (CFG) [1]. A CFG is a pair $\langle N, E\rangle$, where N is a set of nodes representing atomic statements, i.e. assignment statements and conditions (also called predicates) of the program[1], and, E is set of edges such that, $(n_1, n_2) \in E$ if and only if there is a possible flow of control from n_1 to n_2 without any intervening node. We use $n_1 \rightarrow n_2$ and $n_1 \xrightarrow{b} n_2$ to denote unconditional and conditional edges, respectively, where $b \in \{true, false\}$ indicates the branch outcome. Each statement (or node) is associated with a unique label l that represents the program point just before the statement.

[1]For the rest of current part (part I), a statement will mean an atomic statement .

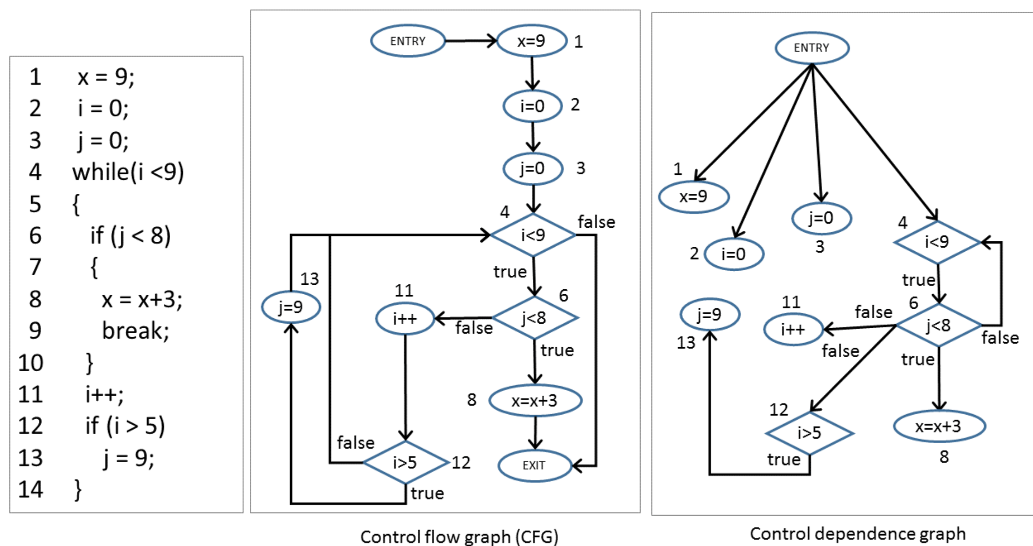


Figure 2.1: Example illustrating CFG and control dependence

Often, we shall refer to a node by its label. In addition, each CFG is assumed to have two distinguished nodes with labels $ENTRY$ and $EXIT$. Except for $ENTRY$ and $EXIT$, there is a one-to-one correspondence between the nodes of the CFG and the statements of the program. Thus we shall use the terms statement and node interchangeably. In the Figure 2.1, we have shown a program and its corresponding CFG. In the CFG, we have labeled the nodes with line number of the corresponding statements.

2.2 Program states and traces

Let Var be the set of variables in a program P and Val be the set of possible values which the variables in Var can take. A *program state* is a map $\sigma : Var \to Val$ such that $\sigma(v)$ denotes the value of v in the program state σ. Since we restrict ourselves to single procedure programs, we do not consider stack variables as part of the program state. Given $X \subseteq Var$, an X*-restriction* of σ, denoted as $\lfloor\sigma\rfloor_X$, is a map $X \to Val$ such that $\forall x \in X. \lfloor\sigma\rfloor_X(x) = \sigma(x)$. Finally, an *execution state* is a pair (l, σ), where σ is a program state and l is the label of a CFG node. The execution of a program is a sequence of execution states starting with $(ENTRY, \sigma_0)$, where σ_0 is the initial program state. We assume that the next state is given by a function $\mathcal{T}$, i.e. for each execution state (l, σ), the next state is $\mathcal{T}((l, \sigma))$.

A *trace* is a (possibly infinite) sequence of execution states $(l_i, \sigma_i),\ i \geq 0$, resulting from an execution of a program. Here, $l_0 =$ *ENTRY*, and σ_0 is the initial program state. Execution state $(l_{i+1}, \sigma_{i+1}) = \mathcal{T}((l_i, \sigma_i))$ for all $i \geq 0$. When a finite trace sequence ends with an execution state (EXIT, σ), it is called a *terminating trace*. We will represent a sequence of execution states (of a terminating trace), $[(l_0, \sigma_0), (l_1, \sigma_1), ..., (l_k, \sigma_k)]$, as $[(l_i, \sigma_i)],\ 0 \leq i \leq k$. We shall only consider terminating traces in the rest of the narrative in the current part (part I) of the thesis.

2.3 Subprograms

A *subprogram* of P is a program formed by deleting some statements from P while retaining its structure. This means if a statement enclosed by a predicate c in P is included in the subprogram, then the predicate c is also part of the subprogram. The deletion of statements is governed by some criterion which depends on the purpose of creating the subprogram. Given a program P and a program location l, an *augmented program* is obtained by inserting a SKIP (do nothing) statement at l. Clearly, an augmented program has the same behavior as the original program. In the sequel, we shall assume that our programs are augmented for a program location l, known from the context. Finally, we shall assume that the program point of the same statement in the original program and the subprogram are represented by the same label.

2.4 Backward slice

Backward slice [72] is a subprogram, created on the basis of a *slicing criterion* defined as a pair $\Upsilon = \langle l, V \rangle$, where l is a statement label and $V \subseteq \mathit{Var}$ is a set of variables. The slicing criterion represents our interest in the values of the variables in V just before the execution of the statement at l. Let $REF(s)$ denote the set of variables referred in a node s. Given a statement s with label l', we shall use $LV(s)$ to denote the slicing criterion $\langle l', REF(s) \rangle$.

Assume for the rest of this section that the slicing criterion is $\Upsilon = \langle l, V \rangle$. Given a program P, we define execution states corresponding to slicing criterion, denoted as *SC-execution states*, to be the execution states of P with label l. For a subprogram to be called a backward slice, there should be a relation between the traces of the program and the subprogram on the same

input when we restrict the traces to their SC-execution states. We call a trace thus restricted as a *sub-trace*. So a sub-trace is a sequence of SC-execution states. We say that the two sub-traces $[(l, \sigma_i)],\ 1 \leq i \leq k$, and $[(l, \sigma'_i)],\ 1 \leq i \leq k'$, are *SC-equivalent* with respect to Υ, if $k = k'$, and, for each i between 1 and k, $\lfloor \sigma_i \rfloor_V = \lfloor \sigma'_i \rfloor_V$.

Let $Tr(P, I, \Upsilon)$ denote the sub-trace of a program P on input I for the slicing criterion Υ. We now define a subprogram P' to be a *backward slice* of P with respect to Υ, if for all inputs I, $Tr(P, I, \Upsilon)$ and $Tr(P', I, \Upsilon)$ are SC-equivalent. As we shall see later, for the same input the sub-traces of a program and its subprogram may not be of the same length. We therefore need a weaker notion of SC-equivalence. We say that a pair of sub-traces $[(l, \sigma_i)],\ 1 \leq i \leq k$ and $[(l, \sigma'_i)],\ 1 \leq i \leq k'$ are *weak-SC-equivalent* with respect to Υ, if for each i between 1 and $min(k, k')$, $\lfloor \sigma_i \rfloor_V = \lfloor \sigma'_i \rfloor_V$. The value $min(k, k')$ is called the *trace observation window* for the pair of sub-traces.

2.5 Data and control dependence

A definition d of a variable v in a node n is said to be a *reaching definition* [1] for a label l, if there is a control flow path from n to l devoid of any other definition of v. A variable x at label l is said to be *data dependent* on a definition d of x, if d is a *reaching definition* for l. Given a set of variables X and a label l, the set of definitions that the variables in X are dependent on is denoted by

Backward slicing algorithms are implemented efficiently using post-dominance and control dependence [36, 46]. A node n_2 *post-dominates* a node n_1 if every path from n_1 to *EXIT* contains n_2. If, in addition, $n_1 \neq n_2$, then n_2 is said to *strictly post-dominate* n_1. A node n is *control dependent* on a conditional edge $c \xrightarrow{b} n'$, denoted $c \overset{b}{\rightarrowtail} n$, if n *post-dominates* n', and n does not strictly *post-dominate* c. If the label b is not important in a context, it is elided. In the Figure 2.1, we have shown the control dependence graph for the program listed there[2].

When there is a chain of control dependence starting with an edge b of a predicate c and ending with a node n, then we say that the node n is *transitively control dependent* on the edge b of the predicate c and denote it as $c \overset{b}{\leadsto} n$. Obviously, if a node n is control dependent on an edge b of a predicate c ($c \overset{b}{\rightarrowtail} n$) then n is *transitively control dependent* also on the edge

[2]By convention, we treat every node that *post-dominates ENTRY* to be control dependent on *ENTRY*

b of the predicate c ($c \overset{b}{\rightsquigarrow} n$). Note that, although it is not possible to have $c \overset{b}{\rightarrowtail} n$ as well as $c \overset{b'}{\rightarrowtail} n$, where $b \neq b'$, but because of `return` and `break` statements, it is possible to have both $c \overset{b}{\rightsquigarrow} n$ and $c \overset{b'}{\rightsquigarrow} n$, where $b \neq b'$. For example, in the Figure 2.1, the node 8 is directly control dependent on the *true* edge of the predicate node 6. But the node 6 itself is control dependent on the *true* edge of the predicate node 4, which in turn is control dependent on the *false* edge of the predicate node 6. Therefore, $6 \overset{true}{\rightsquigarrow} 8$ as well as $6 \overset{false}{\rightsquigarrow} 8$.

Chapter 3

Value slice : A new slicing concept

3.1 Concept of value slice

Given a slicing criterion $\langle l, V\rangle$, a value slice is the answer to the question: "Which statements can possibly influence the values of the variables in V observed at l"?

The answer to this question for the program $P1$ in Figure 3.1 for the slicing criterion $\langle 14, \{\mathtt{y}\}\rangle$ is arrived as follows. The variable y at line 14 gets its value from x through the assignment at line 12. Variable x, in turn, gets its value from the definitions at lines 11 and 5, as both of these can reach at line 12. Thus statements at lines 5, 11 and 12 are in the value slice. The predicate c2 at line 10 is also in the value slice, since, out of the two values generated for x at lines 11 and 5, the value that actually reaches at line 12 is decided by c2. Finally, line 7, where c2 itself is computed, is also in the value slice. The resulting value slice is shown in the figure, as the program $P2$.

Although the program $P2$ contains all the statements required to answer the question posed earlier for the slicing criterion $\langle 14, \{\mathtt{y}\}\rangle$, it is not suitable for property checking. For example, taking the program $P2$ as is, the statement at line 10 will be treated unconditional while in the original program $P1$, the same is conditional. The reason for the same is that apart from the statements that decide the values of variables at the slicing criterion, we also need to explicate the CFG paths along which the computations of these values take place. Therefore, if a statement in the slice is control dependent on a predicate that, by itself, does not influence values of the variables in the slicing criterion, the predicate is retained in the slice in an abstract form.

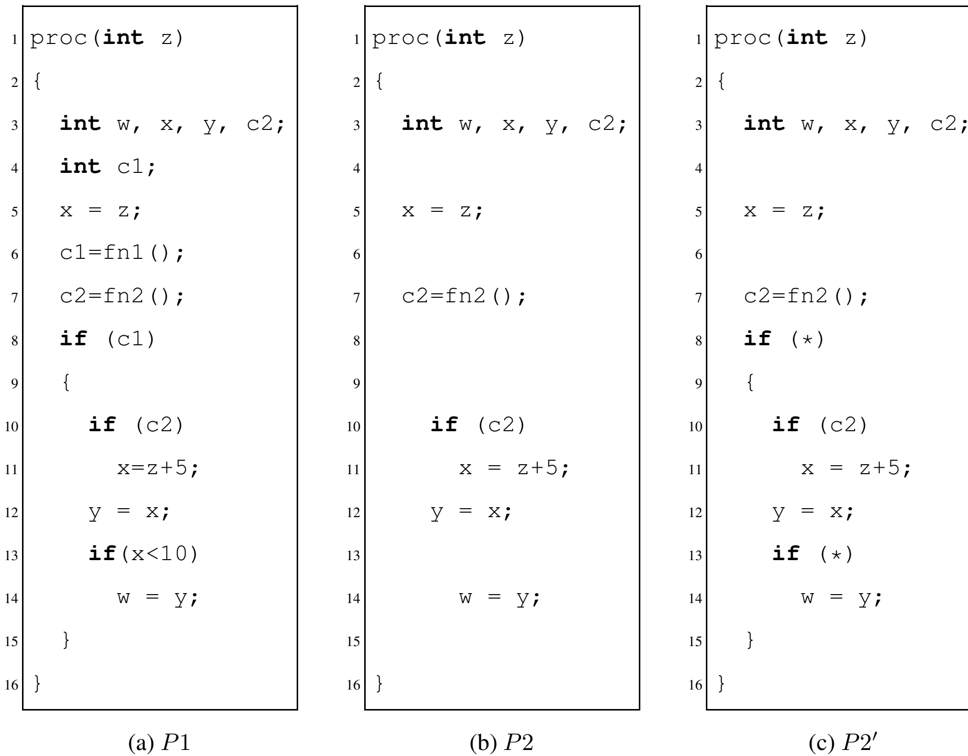
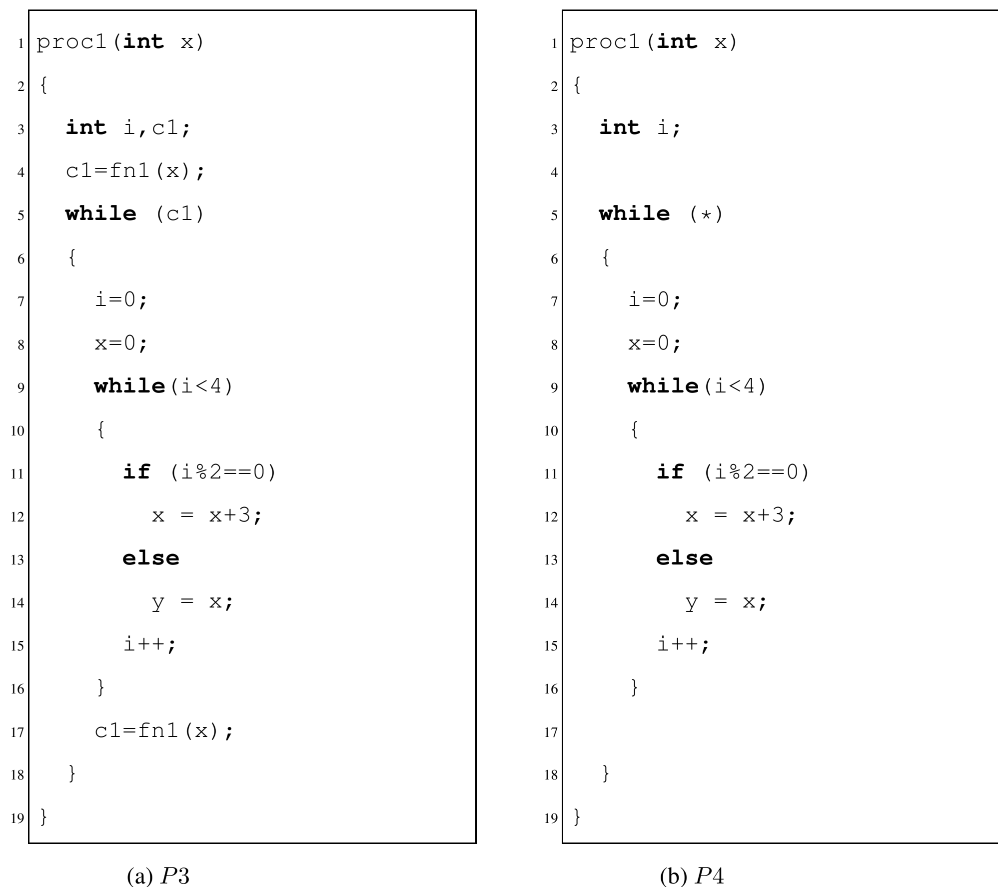


Figure 3.1: Illustration of value slice

This brings the predicates at lines 13 and 8 into the value slice but replaced by '*' indicating a non-deterministic branch. We call such predicates *abstract predicates*. An *abstract predicate* can evaluate to *true* or *false*, non-deterministically in an execution of the program. However, note that if none of the statements that are transitively control dependent on a predicate are included in the slice, the predicate itself can be eliminated. The resulting program, which explicates the CFG paths along which the computations of the values of interest take place, is shown as the program $P2'$ in the same figure, where we include the predicates at lines 8 and 13 as abstract predicates. Note that since an abstract predicate can evaluate to *true* or *false*, non-deterministically in an execution of a program, there can be multiple traces of a program having abstract predicates, for a given input.

In the context of property checking, the inclusion of the predicate c2 in a concrete form at line 10 is a crucial difference between value slice and thin-slice[1]. As an example, assume that when $P1$ is executed with v as the initial value of z, the predicate c2 evaluates to $false$. As a

[1]For comparison in the context of property checking, predicate c2, which would have been eliminated in the thin-slice, is retained in an abstract form.



(a) $P3$ (b) $P4$

Figure 3.2: Generalisation of value slice

result, value assigned to `y` at line 12 will be v. For the same initial value v of `z`, the value slice $P2'$ will also assign the same value v to `y` at line 12. However, if we abstract `c2` also as '*', as in the case of thin slice, the resulting program may produce a trace which assigns the value $v+5$ to y at line 12. To avoid such spurious counterexamples, we retain the predicate `c2` at line 10 in a concrete form.

To generalize this point, consider the program $P3$ in Figure 3.2. Suppose in an execution of the program for a given input, the outer loop executes twice. Obviously, for this execution, the sub-trace generated for $\langle 14, \{\mathrm{x}\}\rangle$ is $[(\mathit{14}, \mathit{3}), (\mathit{14}, \mathit{6}), (\mathit{14}, \mathit{3}), (\mathit{14}, \mathit{6})]$[2]. Suppose we abstract the program $P3$ by abstracting the predicates of both the `while` loops. Consider an execution of the abstracted program for the same input, in which outer loop executes twice, and for every execution of the outer loop, the inner loop executes twice. The sub-trace generated for this execution is $[(\mathit{14}, \mathit{3}), (\mathit{14}, \mathit{3})]$. The two sub-traces do not match in that they are not even weak-

[2]We show only value of `x` in the program state for brevity.

SC-equivalent, because in the second execution state of the first sub-trace, value of x is 6, while in the second sub-trace, it is 3. On the other hand, the program $P4$ in the figure, in which only the outer loop predicate is abstracted, produces sub-traces having zero or more repetitions of the sequence $[(14, 3), (14, 6)]$, because now in every iteration of the outer loop, inner loop will have exactly four iterations. These sub-traces are of course weak-SC-equivalent to the sub-traces produced by $P3$ for the same input. We therefore include the predicate `i<4` in the value slice for the slicing criterion $\langle 14, \{\mathtt{x}\}\rangle$. The predicate `i%2==0` is also in the value slice by a similar argument. In summary, for the same input, the sub-traces of a value-slice and the original program are required to be weak-SC-equivalent. Based on these considerations, we now specify the conditions to be satisfied by a value slice.

Definition 3.1 (Value-slice) *A value slice P^V of a program P for a slicing criterion $\langle l, V\rangle$ satisfies the following conditions:*

1. *P^V is a subprogram of P with zero or more predicates in abstract form.*
2. *If P terminates with trace τ on an input, then there should exist a trace τ' of P^V on the same input such that sub-traces of τ and τ' are* SC-equivalent.
3. *If P terminates with trace τ on an input, then for every trace τ' of P^V on the same input, the sub-traces of τ and τ' should be* weak-SC-equivalent.

Note that, if we do not keep condition (3) then all sub programs, which differ from the original program only in terms of some predicates being abstract, will satisfy the conditions (1) and (2). Which can lead to highly abstracted slices. By including condition (3), we eliminate such gross over-approximation.

3.2 Value-impacting statements

While the trace-based definition is good from a semantic point of view, we present a definition that will enable us to statically identify the set of statements that should necessarily be in the value slice in concrete form. We call such statements *value-impacting* and define the term shortly. As mentioned in the background section, we shall use the term "node" to also mean atomic statements.

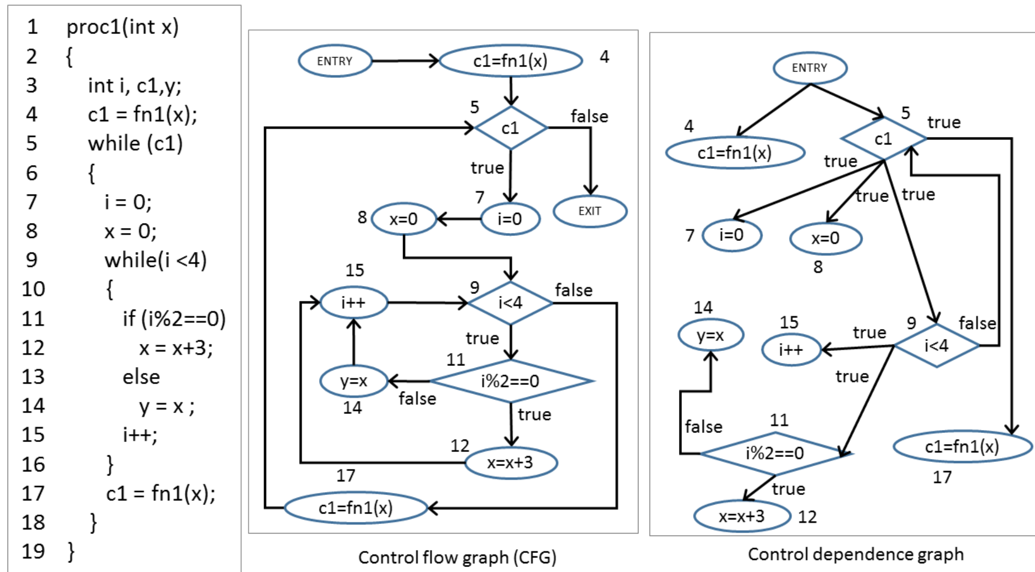


Figure 3.3: Example illustrating value impacting condition

Definition 3.2 (Value-impacting node) *A node* s value-impacts $\Upsilon = \langle l, V\rangle$*, if any of the following conditions hold:*

1. *s is an assignment statement in $DU(\Upsilon)$.*

2. *s is an assignment statement, and there exists a node t such that t* value-impacts *Υ and s is in $DU(LV(t))$.*

3. *s is a predicate from which there exist paths π_1 and π_2, in the CFG, starting with the different out-edges of s and ending at the first occurrence of l. Further, there exists a node $t \neq s$ such that t* value-impacts *Υ, and:*

   (a) t is the first value-impacting node along π_1

   (b) t is not the first value-impacting node along π_2.

A triplet $\langle \pi_1, \pi_2, t\rangle$ due to which a predicate c satisfies rule (3) will be called a *witness* for c being value-impacting. As an illustration, consider the program and its CFG given in Figure 3.3. It is the same program as $P3$ of Figure 3.2. Let the slicing criterion be $\langle 14, \{x\}\rangle$. Statements 12 and 8 are value-impacting because of rules 1 and 2. From the CFG it is easy to see that from the predicate `(i%2==0)` at line 11, there are two paths to the statement `y=x` at

line 14. First is the obvious one namely $11 \xrightarrow{f} 14$, but the second one is more subtle, namely $11 \xrightarrow{t} 12 \rightarrow 15 \rightarrow 9 \xrightarrow{t} 11 \xrightarrow{f} 14$. While the first path has no value impacting statement, the second one has statement 12. Therefore, because of rule 3, the witness $\langle \pi_1\colon 11 \xrightarrow{t} 12 \rightarrow 15 \rightarrow 9 \xrightarrow{t} 11 \xrightarrow{f} 14,\ \pi_2\colon 11 \xrightarrow{f} 14, 12\rangle$ makes the predicate at line 11, value impacting. Similarly, from the predicate `(i<4)` at line 9, there are two paths to the statement `y=x` at line 14, namely:(1) $9 \xrightarrow{t} 11 \xrightarrow{f} 14$ and (2) $9 \xrightarrow{f} 17 \rightarrow 5 \xrightarrow{t} 7 \rightarrow 8 \rightarrow 9 \xrightarrow{t} 11 \xrightarrow{f} 14$. While the first path has the predicate at line 11 as the first value impacting node, the second one has the statement at line 8 as the first value impacting node. Therefore, because of rule 3, the witness $\langle \pi_1\colon 9 \xrightarrow{t} 11 \xrightarrow{f} 14,\ \pi_2\colon 9 \xrightarrow{f} 17 \rightarrow 5 \xrightarrow{t} 7 \rightarrow 8 \rightarrow 9 \xrightarrow{t} 11 \xrightarrow{f} 14, 11\rangle$, makes the predicate at line 9, value impacting. Note that from the predicate `c1` at line 5, there is no path to the statement `y=x` at line 14, starting from the false out-edge of the predicate. In fact the path from false-out-edge goes directly to $EXIT$. Therefore, rule 3 is not applicable for this predicate and as a result, the predicate `c1` at line 5 will not be included as a value impacting predicate for the slicing criterion $\langle 14, \{\texttt{x}\}\rangle$.

Clearly, if a node s *value-impacts* Υ then there is a path from s to l.

3.3 Value slice from value impacting statements

Let $VI(\Upsilon)$ be the set of nodes value-impacting Υ. Let the set of abstract predicates $AP(\Upsilon)$ consist of predicates that are not by themselves *value-impacting*, but on which other value-impacting nodes are transitively control dependent. We construct a subprogram P^{VS} of P by choosing the statements in $VI(\Upsilon) \cup AP(\Upsilon)$ along with $SKIP$ and *ENTRY*. The predicates in $AP(\Upsilon)$ appear in P^{VS} in abstract form.

We claim that P^{VS} is a *value slice*. Clearly condition 1 of Definition 3.1 is satisfied. To show that P^{VS} satisfies conditions 2 and 3, we shall first prove a lemma which shows that if the traces of the original program and the subprogram P^{VS} on the same input are restricted to execution states involving value-impacting statements, then restricted trace of original program is prefix of the restricted trace of the subprogram or vice versa. In the lemmas below, AVI denotes the set of concrete statements in P^{VS}. Further, for $s \in AVI$, $AREF(s)$ denotes $REF(s)$ when $s \in VI(\Upsilon)$, V when s is $SKIP$ and $\emptyset$ when s is *ENTRY*.

Lemma 3.3 *Let τ and τ' be traces of the programs P and P^{VS} for an input I. Assume that $\tau_s = [(l_i, \sigma_i)]$, $i \geq 1$ and $\tau'_s = [(l'_j, \sigma'_j)]$, $j \geq 1$ are restrictions of τ and τ' to the statements in AVI. Let k be the minimum of the number of elements in τ_s and τ'_s. Then for all $i \leq k$, $l_i = l'_i$ and $\lfloor \sigma_i \rfloor_{Z_i} = \lfloor \sigma'_i \rfloor_{Z_i}$, where $Z_i = AREF(l_i)$.*

Proof We shall prove the lemma by induction on the common label index i of the two traces. Obviously $i \leq k$, else the lemma is vacuously true.

Base step : $i = 1$. The lemma holds trivially as $l_1 = l'_1 = ENTRY$ and $\sigma_1 = \sigma'_1 = I$.

Induction step: Let the hypothesis be true for i. Since $\lfloor \sigma_i \rfloor_{Z_i} = \lfloor \sigma'_i \rfloor_{Z_i}$, the edges followed from l_i and l'_i in τ and τ' are the same. Assume $l_{i+1} \neq l'_{i+1}$. This is only possible if (a) there is a predicate c in the original program which has been abstracted in the value slice, (b) the path from l_i to l_{i+1} goes through one of the out-edges b_1 of c, and (c) the path from l'_i to l'_{i+1} goes through the other out-edge b_2 of c. Obviously, there are paths π_1 and π_2 from c to l through b_1 and c to l through b_2, and l_{i+1} and l'_{i+1} are the first value-impacting statements on π_1 and π_2 respectively. Therefore, the predicate c is value-impacting and cannot be abstracted in the value-slice, a contradiction. Therefore, $l_{i+1} = l'_{i+1}$.

Now suppose that for some variable $x \in Z_{i+1}$, $\sigma_{i+1}(x) \neq \sigma'_{i+1}(x)$. Then there must be a statement d which provides the value of x at l_{i+1}; x does not get its value from the input I. This implies d is a value-impacting statement. Clearly, d occurs before l_{i+1} and thus it either also occurs before l_i or is l_i itself. By induction hypothesis, d must also be there in τ' and therefore $\sigma_{i+1}(x) = \sigma'_{i+1}(x)$. ∎

The following lemma helps to prove that the condition 2 of Definition 3.1 holds for P^{VS}.

Lemma 3.4 *Let τ be a finite trace for program P for an input I. Let $\tau' = [(l_i, \sigma_i)]$, $1 \leq i \leq k$, be the sub-sequence of τ restricted to the nodes in P^{VS}. Then for every prefix of τ' of length k', $1 \leq k' \leq k$, there is a prefix $\tau'' = [(l'_i, \sigma'_i)]$, of length k', of some trace of P^{VS} for the same input I, such that for all i, $1 \leq i \leq k'$, (a) $l_i = l'_i$, (b) if l_i is in AVI(Υ), then $\lfloor \sigma_i \rfloor_{Z_i} = \lfloor \sigma'_i \rfloor_{Z_i}$, where $Z_i = AREF(l_i)$.*

Proof Consider a sub-sequence τ' of the given trace τ, restricted to the nodes in P^{VS}. Let the length of the sub-sequence be k. Let τ'_i be the prefix of τ' with length i. The proof is by induction on the length i of the prefix τ'_i.

Base step: $i = 1$ The lemma holds trivially as $[(ENTRY, I)]$ is the only prefix of length 1 for any trace of P as well as P^{VS}.

Induction step: Assume that the statement of the lemma holds for prefixes of τ' of length up to i. Consider a prefix τ'_{i+1} of length $i + 1 \leq k$. By induction hypothesis, there exists a trace of P^{VS}, which has a prefix τ''_i of length i and for which statement of the lemma holds with respect to the prefix τ'_i of τ'_{i+1}. If the node l_i in τ'_{i+1} (and in τ'_i) is an abstract predicate in $AP(\Upsilon)$, then program control reaching the predicate can take either branch. Otherwise $l_i \in AVI(\Upsilon)$, and $\lfloor\sigma_i\rfloor_{Z_i} = \lfloor\sigma'_i\rfloor_{Z_i}$ by the induction hypothesis. Thus for any edge taken out of l_i in τ', l'_i in τ''_i can be made to take the same edge out. Assume this edge extends τ''_i to τ''_{i+1} by taking l'_i to l'_{i+1}.

We claim that there exists a trace of P^{VS} having τ''_{i+1} as its prefix, such that $l_{i+1} = l'_{i+1}$. If not, the divergence must be because of some condition c after l_i and before l_{i+1} in τ'. But then $c \rightsquigarrow l_{i+1}$ and therefore $c \in P^{VS}$. This means that there is a trace of P^{VS} such that $l_{i+1} = l'_{i+1}$. Further, by Lemma 3.3, if $l_{i+1} \in AVI(\Upsilon)$, $\lfloor\sigma_{i+1}\rfloor_{Z_{i+1}} = \lfloor\sigma'_{i+1}\rfloor_{Z_{i+1}}$. ∎

Using Lemma 3.3 and 3.4, it is easy to show that P^{VS} satisfies the second and third conditions of Definition 3.1. Following theorem just does that.

Theorem 3.5 *The abstract subprogram P^{VS} is a* value slice.

Proof Let τ and τ' be traces of the programs P and P^{VS} for an input I. Assume that $\tau_c = [(l_i, \sigma_i)]$, $i \geq 1$ and $\tau'_c = [(l'_j, \sigma'_j)]$, $j \geq 1$ are restrictions of τ and τ' to the statements in AVI. Let k be the minimum of the number of elements in τ_c and τ'_c. Then as per Lemma 3.3, for all $i \leq k$, $l_i = l'_i$ and $\lfloor\sigma_i\rfloor_{Z_i} = \lfloor\sigma'_i\rfloor_{Z_i}$, where $Z_i = AREF(l_i)$. Let τ_1 and τ'_1, be the *sub-traces* of τ and τ', respectively, with respect to $\Upsilon = \langle l, V\rangle$. Since, *SKIP* at l is in AVI, τ_1 and τ'_1 must be embedded in τ_c and τ'_c, respectively. And since, $AREF(l) = V$, τ_1 and τ'_1 must be *weak-SC-equivalent*. Therefore, the subprogram P^{VS} satisfies condition (3) of Definition 3.1.

Let τ be a trace of the program P for an input I. To satisfy the condition (2) of Definition 3.1, we will have to show that there exists a trace τ' of P^{VS} for same input I, such that the sub-traces of τ and τ' are SC-equivalent.

Let $\tau_s = [(l_i, \sigma_i)]$, $i \geq 1$ be a sub-sequence of τ restricted to the nodes of P^{VS}. Let k be the length of τ_s. If we take τ_s itself as a prefix of τ_s, then as per Lemma 3.4, there will be a trace τ_v of P^{VS} for same input I, having a prefix $\tau'_v = [(l'_j, \sigma'_j)]$, $j \geq 1$ of size k such that for all i, $1 \leq i \leq k$, the following holds:

(a) $l_i = l'_i$, (b) if l_i is in $AVI(\Upsilon)$, then $\lfloor \sigma_i \rfloor_{Z_i} = \lfloor \sigma'_i \rfloor_{Z_i}$, where $Z_i = AREF(l_i)$.

If $\tau_v = \tau_s$, then we are done as sub-traces of τ and τ_v must be SC-equivalent.

So assume that $\tau_v \neq \tau_s$. Obviously, the trace τ has no execution state corresponding to some node in P^{VS}, after the occurrence of last element of τ_s. Therefore, the last execution state of τ_s, say (l', σ'), must correspond to an abstract predicate c in $AP(\Upsilon)$ and the trace τ_v diverged from τ by making a different choice for c than what it evaluated to in the state σ' in τ. Now we can have another trace τ_m of P^{VS} for same input I, which imitates τ_v up to its prefix τ'_v but after that (when it should be at abstract predicate c) it makes the same choice for c which c evaluated to in the state σ' in τ. And so τ_m must hit *EXIT* immediately thereafter. So τ_m must be same as τ_s and therefore, the sub-traces of τ and τ_m will be SC-equivalent.

Therefore, the subprogram P^{VS} satisfies condition (2) of Definition 3.1. ∎

3.4 Identifying VI statements using data and control dependence

Value impacting assignment statements can be directly related with data dependence as per the definition 3.2. Computation of data dependence is straight forward using reaching definition analysis. Therefore, the most challenging part of value slice computation is the computation of value-impacting predicates. Given a predicate c, we now identify a necessary and sufficient condition for c to value-impact $\Upsilon = \langle l, V \rangle$. We will relate whether c is value impacting for Υ, with the transitive control dependence relationship between c and l. We observe that, as per the definition of transitive control dependence, only the following three scenarios are possible, for the transitive control dependence relationship between c and l.

1. l is not transitively control dependent on c.

2. l is transitively control dependent on c through exactly one out-edge of c.

3. l is transitively control dependent on c through both out-edges of c.

Obviously, all these three scenarios are mutually exclusive.

Figure 3.4 shows certain situations that we shall refer to in subsequent discussions. In the figure, node c denotes a predicate having two outgoing edges labeled b_1 and b_2, that start the

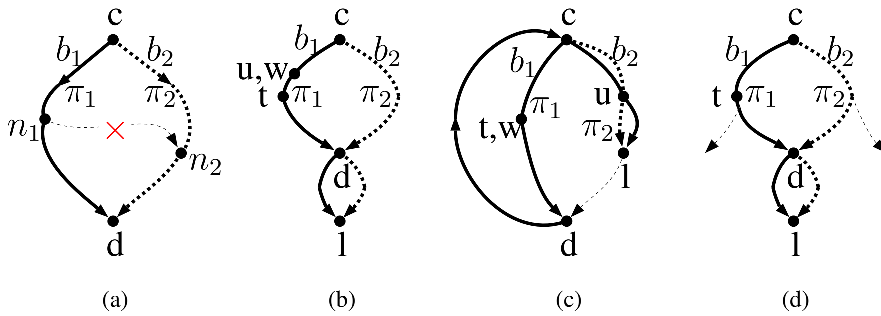


Figure 3.4: (a) A property of CFG paths. (b)-(d) Situations that make a predicate value-impacting. In Fig (c), path π_1 is $c \to t \to d \to c \to u \to l$

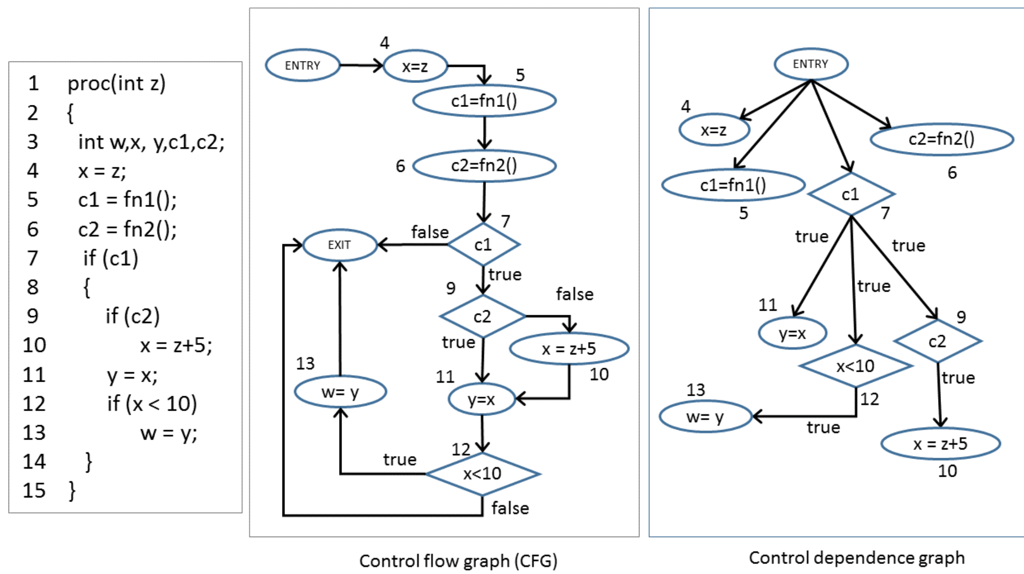


Figure 3.5: Example illustrating value impacting condition of type $cond_1$

paths π_1 (solid line) and π_2 (thick dashed line), respectively. l denotes the node of the slicing criterion. We begin by mentioning a property of the programs under consideration. In figure (a), d is the first node common to π_1 and π_2. As stated earlier, our program model does not allow arbitrary jumps. Therefore, for the programs under consideration, the following property, illustrated in Figure 3.4 (a), holds:

Prop: Let π_1 and π_2 be disjoint paths from a predicate c to a node d, and let n_1 and n_2 be nodes on these paths distinct from d. Then there cannot exist a path from n_1 to n_2 bypassing c.

Consider the program of Figure 3.5 and its CFG and control dependence graph shown in the same figure. It is obvious that the predicate c2 is value-impacting for the slicing criterion

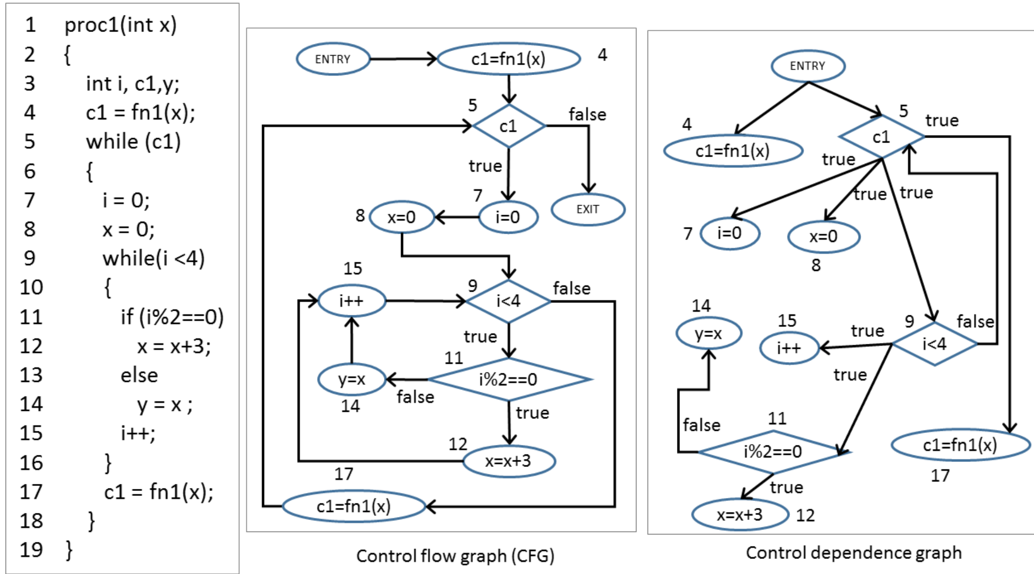


Figure 3.6: Example illustrating value impacting condition of type $cond_2$

$\langle 11, \{\texttt{x}\}\rangle$. Observe in this case that the assignment at line 11 (the criterion) is not control dependent on c2 while a value-impacting statement (assignment at line 10) is control dependent on c2. We generalize this observation, illustrated in Figure 3.4 (b), to claim that the following is a necessary condition for a predicate c to be value-impacting for the slicing criterion $\Upsilon = \langle l, V\rangle$. We will give a proof of this claim later as part of lemma 3.7.

> $cond_1$: If l is not transitively control dependent on c, then there exists a value-impacting node $t \neq c$ such that t is control dependent on c.

Notice that $cond_1$ is also corroborated for the slicing criterion $\langle 13, \{\texttt{y}\}\rangle$, with predicate c2 as c and the assignment at line 10 as t.

Consider the program and its CFG and control dependence graph given in Figure 3.6. It is same as that in Figure 3.3 and reproduced here for ready reference. For this program, as explained in Section 3.2, we had argued that the predicates i<4 and i%2==0 are value-impacting for the criterion $\langle 14, \{\texttt{x}\}\rangle$, as per the definition of value-impacting statements. It is clear that the assignments at lines 8 and 12 are value-impacting for the given criterion. We make some observations about the relationship of these two predicates with the criterion node and with the other value-impacting nodes in terms of control dependence. Note that the criterion node at line 14 is transitively control dependent on the predicate i<4 through the true out-edge but not

through false out-edge. The assignment at line 8 is value-impacting for the criterion $\langle 14, \{\mathtt{x}\}\rangle$. The same assignment is reachable through the false-edge of predicate `i<4`, as both are in a cycle $9 \xrightarrow{f} 17 \rightarrow 5 \xrightarrow{t} 7 \rightarrow 8 \rightarrow 9$. But this assignment is not transitively control dependent on the predicate `i<4` through the true out-edge (in fact the assignment is not transitively control dependent at all on the predicate). So in summary, the criterion node is transitively control dependent on the predicate (`i<4`) through true out-edge only, and a value-impacting statement (assignment at line 8) is in cycle with the predicate but not transitively control dependent on the predicate through true out-edge. Similarly, note that the criterion node is transitively control dependent on the predicate `i%2==0` through false out-edge only, and a value-impacting statement (assignment at line 12) is in cycle with the predicate but not transitively control dependent on the predicate through false out-edge (although it is control dependent on the predicate through true out-edge). From these observations, generalized in Figure 3.4 (c), we claim that the following also is a necessary condition for a predicate c to be value-impacting for the slicing criterion $\Upsilon = \langle l, V\rangle$. We will give a proof of this claim also as part of lemma 3.7.

> $cond_2$: If l is transitively control dependent on c through only one out-edge, say b_2, then there exists a value-impacting node $t \neq c$ such that t is not transitively control dependent on c through b_2, and c and t are in a cycle in the CFG.

There is a third condition $cond_3$ which covers the case when l is transitively control dependent on c through both out-edges, as shown through Figure 3.4 (d). Consider the program and its CFG and control dependence graph of Figure 3.7. As mentioned earlier, transitive control dependence through both out-edges happens when some of the paths starting from out-edges of a predicate do not merge back due to `return` or `break` statements. For example, this is the case with the condition `(j>5)` at line 5. Consider the slicing criterion $\langle 7, \{\mathtt{x}\}\rangle$. Statement at line 10 is value impacting due to rule (1) of the definition 3.2. For the predicate node `(j>5)`, we have a witness: $\langle \pi_1\colon 5 \xrightarrow{t} 10 \rightarrow 11 \rightarrow 3 \xrightarrow{t} 5 \xrightarrow{t} 7, \pi_2\colon 5 \xrightarrow{f} 7,\ 10\rangle$, to make the predicate value impacting. But the statement at 7 is transitively control dependent on predicate `(j>5)` at line 7 through both out-edges, as evident from control dependence graph: $5 \xrightarrow{t} 7$ and $5 \xrightarrow{f} 3 \xrightarrow{t} 5 \xrightarrow{t} 7$. And therefore, it is not covered by $cond_1$ or $cond_2$. However, in this case, we observe that a value impacting statement (assignment at line 10) is transitively control dependent on the predicate `(j>5)` only through one out-edge (*false*). We generalize this observation, illustrated

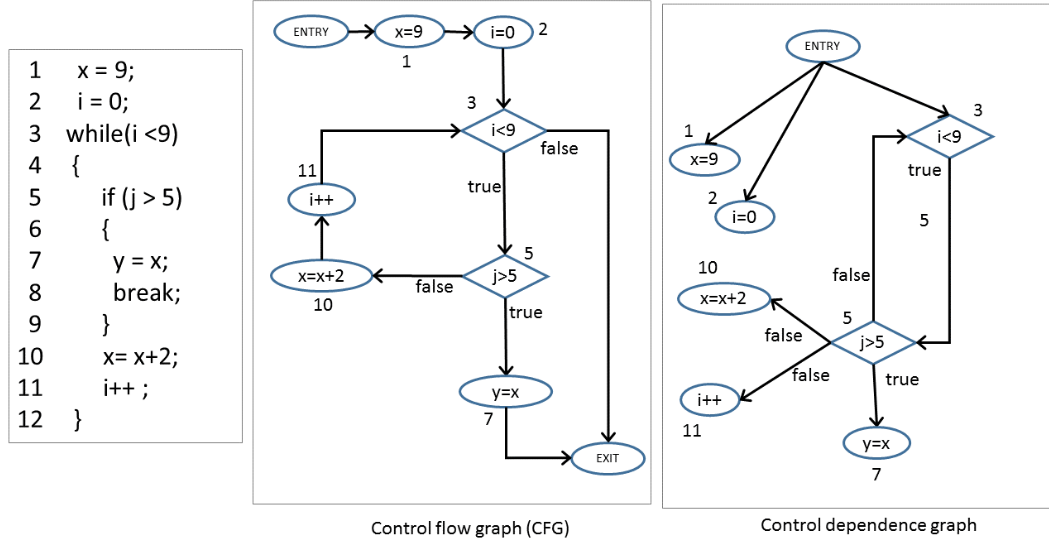


Figure 3.7: Example illustrating value impacting condition of type $cond_3$

in Figure 3.4 (d), to claim that the following is another necessary condition for a predicate c to be value-impacting for the slicing criterion $\Upsilon = \langle l, V \rangle$. We will give a proof of this claim too, as part of lemma 3.7.

> $cond_3$: If l is transitively control dependent on c through both out-edges, then there exists a value-impacting node $t \neq c$ such that t is transitively control dependent on c through exactly one out-edge.

Note that the antecedent of exactly one of the three conditions: $cond_1$, $cond_2$ and $cond_3$, always holds. Therefore, for the conjunction of these conditions to hold, only the consequent of the condition, with true antecedent, needs to hold; the other two conditions will hold vacuously. We will now show that the conjunction of these three conditions: $cond_1$, $cond_2$ and $cond_3$, is a necessary and sufficient condition for a predicate c to be value-impacting for a given slicing criterion $\Upsilon = \langle l, V \rangle$. This result can thus be used for obtaining a sound and precise value slice. But we first prove a property of the witness of a value-impacting predicate.

Lemma 3.6 *Let c be a value-impacting node for the slicing criterion $\langle l, V \rangle$ with a witness $\langle \pi_1, \pi_2, u \rangle$. Then, at least one of π_1 or π_2 must have a value-impacting node before any common node appearing on both π_1 and π_2.*

Proof Let π_1' and π_2' be the prefixes of π_1 and π_2 such that both end with a common node d (possibly l itself), and other than the starting node (c the condition) and the ending node d, rest of the nodes on π_1' are disjoint from the rest of the nodes on π_2'. To prove the lemma, we have to show that at least one of the π_1' and π_2' has a value impacting node before the end node d.

Assume that both π_1' and π_2' have no value-impacting statements before d. Obviously, $u \neq d$ otherwise, contrary to our assumption, c will not be value-impacting. Since u is not the first value-impacting on π_2, π_2 must diverge from π_1 after d but before u. The divergence point will have to be a common predicate node, say c'. Consider the suffixes π_1'' and π_2'' of π_1 and π_2 respectively, starting at the node c'. Obviously, u will be the first value impacting node on π_1'' but not on π_2''. Therefore, there will be a witness $\langle \pi_1'', \pi_2'', u \rangle$ for c' to be a value impacting node. But then c' will be a value impacting node on π_1 before u, which is a contradiction. ∎

We now show that the conjunction of $cond_1$, $cond_2$ and $cond_3$ is a necessary criterion for a predicate c to be value-impacting.

Lemma 3.7 *Given a slicing criterion $\Upsilon = \langle l, V \rangle$ and a value-impacting predicate c, conditions $cond_1$, $cond_2$ and $cond_3$ hold.*

Proof Since the antecedent of exactly one of $cond_1$, $cond_2$ and $cond_3$ will be true, we need to show that the consequent of the one with true antecedent is true.

1. The antecedent of $cond_1$ is true i.e. l is not transitively control dependent on c.

   We need to show that there is a value impacting node $t(\neq c)$ such that t is control dependent on c. Let $\langle \pi_1, \pi_2, u \rangle$ be a witness for c to be a value-impacting statement for l. Let π_1 and π_2 start with the out-edges b_1 and b_2 of c, respectively. Since l is not transitively control dependent on c, l must post-dominate c and the situation must be as depicted in Figure 3.4 (b), where d is the immediate post-dominator of c. By Lemma 3.6, at least one of π_1 or π_2 must have the first value-impacting node w before d.

   First assume that w lies on the segment of π_1. Obviously, $w = u$ and w must post-dominate the out-edge b_1. In addition, by *Prop*, w can not strictly post-dominate the out-edge b_2. Therefore w must be control dependent on c. Thus w is the required t.

Now assume that w lies on the segment of π_2. Obviously, w must post dominate the out-edge b_2 and by *Prop*, w can not strictly post-dominate the out-edge b_1. Therefore w must be control dependent on c. Thus w is the required t.

2. The antecedent of $cond_2$ is true i.e. l is transitively control dependent on c through only one out-edge.

   Without loss of generality, assume that l is transitively dependent on c through out-edge b_2 only. We need to show that there is a value impacting node $t'(\neq c)$ such that t' and c are in a cycle and t' is not transitively control dependent on c through out-edge b_2. Since c is value-impacting, there must be a witness as $\langle \pi_1, \pi_2, w \rangle$ or $\langle \pi_2, \pi_1, u \rangle$ with π_1 and π_2 starting with out-edges b_1 and b_2, respectively. Since l is transitively control dependent on only one out-edge, b_2, by *Prop*, the path π_1 must be such that π_2 is a suffix of π_1, that is c must be in a loop having the out-edge b_1. So the situation resembles Figure 3.4 (c).

   Assume the witness is $\langle \pi_2, \pi_1, u \rangle$. There must exist a value-impacting node $t \neq u$ in the looping segment c to c of π_1, otherwise, u will become the first value impacting node on π_1 also. Obviously, the node t is in a cycle with c and it is not transitively control dependent on out-edge b_2. So t is our desired t'.

   Assume the witness is $\langle \pi_1, \pi_2, w \rangle$. Obviously, the value-impacting node w must be in the c to c looping segment of π_1, and therefore the node w is in a cycle with c and it is not transitively control dependent on out-edge b_2. So w is our desired t'.

3. The antecedent of $cond_3$ is true i.e. l is transitively control dependent on c through both out-edges b_1 and b_2.

   We need to show that there is a value impacting node $t(\neq c)$ such that t is transitively control dependent on c through exactly one out-edge. Since c is value-impacting, there will be a witness with paths π_1 and π_2 as shown in Figure 3.4 (d). By Lemma 3.6, there is a value-impacting statement t on π_1 or π_2 before d. Without loss of generality, we assume that t is on π_1 and it is the first value impacting statement on π_1. By *Prop*, t has to be transitively control dependent on c through b_1 and only through b_1.

∎

We now show that the conjunction of $cond_1$, $cond_2$ and $cond_3$ is a sufficient condition as well for a predicate node c to be value-impacting.

Lemma 3.8 *Given a slicing criterion $\Upsilon = \langle l, V \rangle$ and a predicate c such that the conditions $cond_1$, $cond_2$ and $cond_3$ hold, c is value-impacting for Υ.*

Proof Although all three conditions hold, two of them will hold vacuously and only one of them will hold along with its antecedent. We will consider the three cases of conditions for which its antecedent holds, and in each case, we shall identify a witness for c to be value-impacting for Υ.

1. The antecedent of $cond_1$ is true i.e. l is not transitively control dependent on c. Since $cond_1$ is true, there exists a value impacting node t which is control dependent on c. Without loss of generality, assume that t is control dependent on c through the out-edge b_1. Since there must be a path from t to l, there must be a path, say π_1, from c to t to l starting with out-edge b_1. Clearly, t post-dominates edge b_1. Therefore, since l is not transitively control dependent on c, there must be a path, say π_2 from c to l through out-edge b_2 as well such that π_2 does not pass through t. Consider the first value-impacting statement u between c and t on the path π_1 (u may be the same as t). Then the required witness is $\langle \pi_1, \pi_2, u \rangle$ as shown in Figure 3.4 (b).

2. The antecedent of $cond_2$ is true i.e. l is transitively control dependent on c through only one out-edge. Without loss of generality, assume l is transitively control dependent on c only through one out-edge, say b_2. Since $cond_2$ is true, there exists a node t that is not transitively control dependent on c through b_2 and that c and t are in a cycle. Since t is not transitively control dependent on c through out-edge b_2 the cycle must contain the out-edge b_1 and must not contain out-edge b_2. Consider the first value-impacting statement w on path segment c to t (w may be the same as t). Then the witness is $\langle \pi_1, \pi_2, w \rangle$, as shown in Figure 3.4 (c).

3. Antecedent of $cond_3$ is true i.e. l is transitively control dependent on c through both the out-edges. Since $cond_3$ is true, there exists a value impacting node t which is transitively control dependent on c through only one out-edge. Without loss of generality, assume

that this out-edge is b_1. Then the witness is $\langle \pi_1, \pi_2, t' \rangle$, as shown in Figure 3.4 (d), where t' is the first value impacting node on π_1 and may be same as t.

■

3.5 Value slice computation

Given a program dependence graph (PDG) [36], representing data and control dependence in the program, it is easy to compute value-impacting assignments using Definition 3.2. In addition, Lemmas 3.7 and 3.8 can be used to identify value-impacting predicates. These value-impacting assignments and predicates are augmented with abstract predicates to obtain the value-slice. A minor implementation detail is that a predicate with the reaching definitions of all its variables in VI, is retained in concrete form, even if the predicate itself is not in VI. Abstracting the predicate in this case would not result in a decrease in the size of the slice. Note that the precision of the slice depends on the precision of the PDG; given a precise PDG, the computed value slice exactly matches P^{VS}.

3.5.1 Value slice computing algorithm

Figure 3.8 gives an algorithm to compute $VI(\langle l,\, V \rangle)$ for the given slicing criterion $\Upsilon = \langle l, V \rangle$. Meaning of the variables used in the algorithm are listed in Table 3.1. We use $tcd(t, b)$ to denote $\{c \mid c \overset{b}{\rightsquigarrow} t\}$ and $cd(t)$ to denote $\{c \mid c \rightarrowtail t\}$. For a given t and b, we compute $tcd(t, b)$ from the PDG of the program. Similarly, we use PDG of the program to compute $cd(t)$ for a given t. Further, we use $incycle(t)$ to denote the set of predicates which, along with t, appear in a cycle in CFG. The worklist wl in the algorithm contains value-impacting statements which have not been explored yet, i.e. they have not been used to find other value-impacting statements.

The set vi contains value-impacting statements which have been explored. Given a value-impacting statement t, ic is the set of predicates and $DU(LV(t))$ is the set of assignments that become value-impacting because of t. Any statement which is put in the worklist wl, eventually does get into the set vi and no elements are removed ever from vi. Since we initialise wl with $DU(l, V)$ at line 6, it is obvious that the algorithm satisfies the condition (1) of the definition 3.2. For every value impacting node t, which is put in vi, we add $DU(LV(t))$, (at lines 11-12), to

Symbol	Meaning
$cd(t)$	Set of predicates on which t is control dependent
$tcd(t, b)$	Set of predicates on which t is transitively control dependent through out-edge b
lct	Set of predicates on which the statement l, of the slicing criterion, is transitively control dependent through out-edge $true$ of the predicates
lcf	Set of predicates on which the statement l, of the slicing criterion, is transitively control dependent through out-edge $false$ of the predicates
lc	Union of lct and lcf
lcb	Intersection of lct and lcf
$lcot$	Set of predicates on which the statement l, of the slicing criterion, is transitively control dependent through out-edge $true$, but not through out-edge $false$, of the predicates
$lcof$	Set of predicates on which the statement l, of the slicing criterion, is transitively control dependent through out-edge $false$, but not through out-edge $true$, of the predicates
$incycle(t)$	Set of predicates, which along with the given node t, appear on a cycle in the CFG
vi	Set of value impacting statements, which have been identified as well as explored for identifying further value impacting statements due to them
wl	A worklist of value impacting statements, which have not been explored yet for identifying further value impacting statements due to them
ic	Set of predicates that become value impcating due to a given value impacting statement t (i.e. t participates in a witness triplet)
tct	Set of predicates on which a given statement t is transitively control dependent through out-edge $true$ of the predicates
tcf	Set of predicates on which a given statement t is transitively control dependent through out-edge $false$ of the predicates
tco	Set of predicates on which a given statement t is transitively control dependent through one and only one out-edge of the predicates
dc	Set of predicates on which a given statement t is control dependent
cnd_1	Set of predicates satisfying the condition $cond_1$ for a given value impacting statement t
cnd_2^b	Set of predicates satisfying the condition $cond_2$ for a given value impacting statement t, when l is transtively control dependent on these predicates through out-edge b only
cnd_3	Set of predicates satisfying the condition $cond_3$ for a given value impacting statement t

Table 3.1: Description of variables used in the value slice computing algorithm

wl, which satisfies the condition (2) of the definition 3.2.

We have already shown that a predicate node becomes value-impacting by satisfying condition (3) of the definition 3.2 if and only if it satisfies the conjunction of the three criteria: $cond_1, cond_2$ and $cond_3$. In the main procedure $compVI$, at line 3, the lct represents set of conditions on which l is transitively dependent through $true$ edge. Similarly, lcf provides set of conditions on which l is transitively control dependent through $false$ edge. From these two, we get lc as union of conditions on which l is transitively control dependent, $lcot$ as set of conditions on which l is transitively control dependent through *true* out-edge only, $lcof$ as set of conditions on which l is transitively control dependent through *false* out-edge only and lcb as set of conditions on which l is transitively control dependent through both the out-edges.

The set ic is computed using the function $iConds$ which encodes $cond_1$, $cond_2$ and $cond_3$ in a straightforward manner. In line 6 of the function $iConds$, the set dc is set of conditions on which t is control dependent. So, at line 7 in the function, computation of cnd_1 rightly encodes $cond_1$ by removing lc from dc to provide the value impacting conditions on which l is not transitively dependent and on which t is control dependent.

The set tct computed at line 3, consists of those conditions on which t is transitively control dependent through $true$ edge. Similarly, the set tcf computed at line 4, consists of those conditions on which t is transitively control dependent through $false$ edge. From these two, at line 5, we compute set of conditions, tco, on which t is transitively control dependent through exactly one out-edge. As per $cond_2$, for condition c to be value impacting due to t, all of the following criteria should hold:

(a) t and c should be in a cycle

(b) l should be transitively control dependent on c through only one out-edge so c must belong to $lcot$ or $lcof$.

(c) t is not transitively control dependent on c through the out-edge through which l is transitively control dependent on c.

The criteria (b) and (c) imply that either c is in $lcot$ but not in tct, or c is in $lcof$ but not in tcf. Accordingly, we compute the set of value impacting conditions as per $cond_2$ in two parts, in cnd_2^t and cnd_2^f, at lines 8-9.

For a condition c to be value impacting due to the value impacting node t as per criterion $cond_3$, following should be satisfied:

```
1: function compVI(l, V)
2: begin
3: lct = tcd(l, true); lcf = tcd(l, false)
4: lc = lct ∪ lcf; lcb = lct ∩ lcf
5: lcot = lct \ lcf; lcof = lcf \ lct
6: vi = ∅
7: wl = DU(l, V)
8: while wl is not empty do
9:     choose an element t from wl
10:    ic = iConds(t, lc, lcot, lcof, lcb)
11:    vi = vi ∪ {t}
12:    wl = (wl \ {t}) ∪
13:         ((ic ∪ DU(LV(t))) \ vi)
14: end while
15: return vi
16: end
```

```
1: function iConds(t, lc, lcot, lcof, lcb)
2: begin
3: tct = tcd(t, true)
4: tcf = tcd(t, false)
5: tco = (tct ∪ tcf) \ (tct ∩ tcf)
6: dc = cd(t)
7: cnd_1 = dc \ lc
8: cnd_2^t = (lcot \ tct) ∩ incycle(t)
9: cnd_2^f = (lcof \ tcf) ∩ incycle(t)
10: cnd_3 = lcb ∩ tco
11: return cnd_1 ∪ cnd_2^t ∪ cnd_2^f ∪ cnd_3
12: end
```

Figure 3.8: Algorithm to compute VI

(a) l should be transitively control dependent on c through both out-edges, which means c must belong to lcb

(b) t should be transitively control dependent on c through exactly one out edge, so c must belong to tco.

Accordingly, we compute the set of value impacting conditions as per $cond_3$ in cnd_3 at line 10.

3.5.2 Algorithm complexity

Assume there are E edges and N nodes in the CFG, and out of N nodes, C are predicates. Since a node goes into the worklist at most once, the while loop in $compVI$ iterates at most N times. Further, let there be E_d data dependence and E_c control dependence edges in the PDG, adding to total $E_p = E_d + E_c$ edges in the PDG. The sets lct and lcf can be pre-computed in $O(C)$ time and stored in $O(C)$ space, so that membership of these sets can be checked in constant time. Further, Tarjan's algorithm [69] can be used to find all strongly connected components (SCCs) in a CFG in $O(E + N)$ time, from which we can pre-compute $incycle(t)$. This takes

$O(N \times C)$ time and $O(N \times C)$ space. Thus $c \in incycle(t)$ can also be checked in constant time.

It is clear that each data dependent edge will be traversed at most once during the entire run of $compVI$. Similarly, because of dc and cnd_1, each control dependent edge will also be visited at most once during execution of $compVI$. The computation of tct. tcf, cnd_2 and cnd_3 all require $O(C)$ time. So the overall complexity of the algorithm is $O(E+N)+O(N \times C)+O(E_c+E_d) \approx O(N \times C)+O(E_p)$. Note that backward slice computation has a complexity of $O(E_p)$. Since in the worst case $O(E_p) = O(N \times N)$, the worst case complexity is the same for backward slice and value slice.

Chapter 4

Implementation and measurements

In this chapter we discuss the implementation details of value slice computation and results on experiments conducted using the implementation.

4.1 Implementation

We have built a scalable property checking tool based on value slicing, implemented on top of PRISM, a static analyzer generator developed at TRDDC, June [53, 20]. Our implementation supports full version of C including pointers, structures, arrays, heap allocation and function calls. However, we assume that execution terminates from the *main* function only. For example, we assume that there is no call to *exit()* from functions other than *main*.

Following the conventional approach, the heap is abstracted in terms of allocation points and arrays are summarized to a single abstract element. However, structures are handled in field sensitive manner: `x.a` and `x.b` are treated as separate entities. Although, a flow-insensitive and context-sensitive points-to analysis is generally beneficial, we handle pointers using a flow sensitive but context insensitive points-to analysis, as its implementation was already implemented as part of PRISM.

4.1.1 Interprocedural PDGs

We first construct an intraprocedural PDG for each function. We treat the function entry and exit (return) as the program $ENTRY$ and $EXIT$ points respectively. For data dependence,

we use reaching definition analysis available from PRISM. For control dependence, we have implemented the algorithm of Billardi and Pingali [11] to construct the control dependence graph.

To make the value slice precise, we augment the intraprocedural PDGs with interprocedurally valid data and control dependence as defined by Horowitz et al. [46]. As a first step towards this, we extend the intraprocedural PDGs as follows:

1. A function entry point is made control dependent on its call points.

2. If a definition in a caller passes its computed value to the callee through a formal parameter or as a global, then the entry of the callee is made data dependent on the definition.

3. Similarly, if a definition in a called function passes its computed value to the caller through a global variable or return statement, the calling point is made data dependent on the definition.

Due to these extensions, it is clear that the control dependence due to interprocedural control flow is automatically taken care of.

Over the extended PDGs, we use a method adapted from the approach on interprocedural slicing by Horowitz et al. [46] to compute interprocedurally valid data and control dependence. In the following section, we describe how we take care of the imprecision issues in data dependence, arising due to inter-procedural data flow.

4.1.2 Computing inter-procedural data dependence

A naive reaching definition analysis over an integrated inter-procedural CFG gives rise to imprecision in the data dependence. To illustrate this issue, let us consider the program in Figure 4.1, where both the definitions of x (at lines 18 and 24) will reach to both uses of `x` (at lines 10 and 30) if an integrated interprocedural CFG is used for reaching definition analysis. However, this information is imprecise. For example, the definition at line 18 can never reach line 30. To address the issue, we compute reaching definitions of functions in an intra-procedural manner. We use a summary based approach and do the computation for all the functions by traversing the call graph in a bottom-up manner[1]. To incorporate the inter-procedural data flow

[1]For recursive functions, an iteration-till-saturation process is adopted

from called functions to their callers, we create a function summary for each function. The summary consists of the set of definitions created in the function, called *sum-gen-defs* and the set of variables which get defined unconditionally, called *sum-kill-defs*. While computing the reaching definitions in a caller function the following is how we proceed at a call point:

1. We kill the in-flowing definitions of those variables that belong to *sum-kill-defs* of the called function.

2. We consider that all the variables belonging to *sum-kill-defs* are getting defined at the call point.

3. We generate the definitions of *sum-gen-defs* of the called functions in such a way that they deem to emanate from the call point.

Although the set *sum-gen-defs* will subsume the set *sum-kill-defs*, for our approach to work properly we need both information separately. While constructing the interprocedural PDG, for creating data dependence edges due to inter-procedural data flow, we proceed as follows:

1. The definitions generated at a call point are made data dependent on the corresponding definitions in *sum-gen-defs* of the called function.

2. In a function, when a variable x, used at a point is also live at the function entry, we add a data dependence edge from the use point to an auxiliary definition[2], $x = x$, created just before the call point.

To illustrate this, let us consider the example program in Figure 4.1. The computed summary of each function is shown at the end of the function body as a pair of sets, one for *sum-gen-defs* and another one for *sum-kill-defs*. Function `C` defines the variable `z` unconditionally. Therefore, in the summary of `C`, shown just after the end of the function body at line 15, *sum-gen-defs* will have the definition of `z` at line 14, and *sum-kill-defs* will only have the variable `z`. To elaborate further, while computing the reaching definitions in the function `A1`, after the call point `C()` at line 19, the definition of `x` at line 18, and the definition of `z` deemed to emanate at the call point, will be reaching as per the summary of `C`. Similarly, the definition of `w` emanating

[2]The use of x in the RHS will be considered for data dependence edges subsequently

```
1  int x,z;
2  int w,y;
3  int main()
4  {
5    A1();//{(5, x),(5,w),{5,z}}
6    A2();
7  }
8  void B1()
9  {
10   w = x+z;
11 } // {(10, w)} , {w}
12 void C()
13 {
14   z = 10;
15 } // {(14, z)} {z}
```

```
16 void A1()
17 {
18   x = 10;
19   C();  // {(18, x),(19,z)}
20   B1();
21 } // {(18, x),(20,w),(19,z)}, {x,w,z}
22 void A2()
23 {
24   x = 20;
25   C();  // {(24, x),(25,z)}
26   B2();
27 } // {(24, x),(26,y),(25,z)}, {x,y,z}
28 void B2()
29 {
30   y = x+z;
31 } // {(30, y)}, {y}
```

Figure 4.1: Program to illustrate interprocedural data dependence computation

at the call point `B1()` at line 20 will reach the function exit. Accordingly, the summary of `A1` will be what is shown at the end of its body (in line 21).

To illustrate how inter-procedural data dependence is computed precisely, consider the use of `x` at line 10 and line 30 in functions `B1` and `B2`, respectively. For the use of `x` in line 10, while looking for data dependence, we will go to the calling point of `B1` as there is no definition of `x` reaching this point within the function. At the call point `B1()` in function `A1`, we find that the definition of `x` at line 18 is reaching there. So, we will make the use of `x` at line 10 data dependent on the definition at line 18 only. By a similar argument, the use of `x` at line 24 will be data dependent on the definition of `x` at line 30 only.

4.1.3 Value slice computation

We have implemented the algorithm described in Section 3.5 using the interprocedural PDGs as describe above. We augment the intraprocedural CFGs by connecting calls to procedure entry points, and procedure return nodes to successor of call nodes to get an interprocedural CFG of the program. To implement the check *incycle(c,t)*, we have implemented Tarjan's algorithm [69] of finding strongly connected components to identify cycles in the interprocedural CFG.

4.2 Experiments

We carried out our experiments on 3.0 GHz Intel Core2Duo processor with 2 GB RAM and 32 bit WINDOWS operating system, and we used SATABS (version 3.0) [23] as the verifier for its robustness and scalability.

4.2.1 Source code for experiment

We experimented on one open source application, $icecast$, and one proprietary code base. The proprietary code base is from automotive domain, and it implements an entertainment and navigation system for the automobiles. The entire proprietary system consists of a total of 123 modules (processes) spread over more than 4000 source files, having more than 5 MLOC and 35000 functions. Out of these 123 modules, we picked 60 modules of varying sizes for our experiments. For ease of reporting, we grouped these 60 modules into nine groups: $navi1$ to $navi9$. The average size of individual modules in these nine groups varied from 6 KLOC to 61 KLOC. We present the data indicating size and complexity of the source code chosen for experiments in Table 4.1. This includes the number of functions, their sizes, and, the number of conditional statements and loops within the functions, for icecast as well as the 9 groups of proprietary code. We provide the total code size in KLOC for the group in column (b), the number of functions (excluding uncalled functions) that belong to the group in column (c), the size of the code present in body of these functions in column (d), the average LOC of a function body in column (e), and the maximum LOC of a function body in column (f). Further, we show the total number of conditional statements (if and switch) in these functions in column (g), the average number of conditional statements per function in column (h), and the maximum number of conditional statements in a function in column (i). Similar data is shown for loops (for, while and do-while) in columns (j), (k) and (l), respectively. Table 4.1(b) presents similar data for pointer-dereferences in columns (c), (d) and (e), and for array references in columns (f), (g) and (h).

We checked for the "array index out of bounds" property on these programs. For each array reference chosen for the checking we generated an *assert* to check that the index value is not negative and it is less than the size of the array. We checked the validity of generated asserts using SATABS on three kinds of slices generated for the assert: backward slice, value slice and

thin slice. To construct these asserts, we needed the size of arrays used in array references. Since our implementation can not infer the size of dynamic arrays, we had to exclude array references of dynamic arrays from our checking. To be able to add *assert* statements with ease, we considered only those array references that were part of an expression statement. Further, we excluded the array references having only constant indexes. We report the number of such array references in columns (i), (j) and (k) of Table 4.1(b), in the same manner as we show number of pointer-dereferences.

4.2.2 Conducting the experiment

For $icecast$, there were a total of 65 instances of array references of the kind we considered for our experiments. For the 60 modules of proprietary application, there were a total of 5950 array references of the type we considered, and their number varied from 3 to 548 in different modules. Since running SATABS on all such instances would need a very long time (estimated to be more than 25 days!!), we picked up a maximum of 10 such instances from each module to check the desired property. For a module with 10 or less instances, we picked all the instances, otherwise, we picked 10 instances from the module randomly. In the two tables of Table 4.2, we show the combined size of each group, and number of array references selected for checking "array index out of bound". In all, we checked a total of 647 cases of array references.

For each chosen instance, we computed backward slice, value slice and thin slice. We ran SATABS on each of the three slices with three different timeouts: 1 minute, 2 minutes and 3 minutes. The maximum number of CEGAR loop iterations was set to default (which is 50). Five kinds of outcomes were recorded: property satisfied (Y), property failed (N), timeout, too many CEGAR loop iterations, and tool failure. The possible reasons for the last outcome are refinement failure (inability to find new predicates for further refinements), and SATABS failing due to some bug or a limitation of the tool. The outcome of timeout, too many CEGAR loop iterations, and tool failure were clubbed as no decision ($?$) for reporting purposes.

4.2.3 Experiment observation

The Y answers of all three slices will be correct by construction of the slice. Similarly, an N answer for the backward slice will also be correct[3]. However, in case of a value or thin slice, if an assert with an N answer was recorded as a Y during property checking with the other two slices, it was recorded as being a spurious N (N_S in the table) also. Scalability, given by $(Y + N)/(Y + N+?)$ is the ratio of definite outcomes over all outcomes. Loss of precision is the ratio of outcomes that are known to be spurious over all definite outcomes ($N_S/(Y + N)$).

The data from the experiments are presented in the two tables of Table 4.2. In the Table 4.2(a), we have shown the size and the number of asserts for each program or group. We report the outcomes based on a time out of 3 minutes. Outcome of verification using backward slice is presented in the Table 4.2(a). In the Table 4.2(b), we present the outcome of verification using value and thin slices.

To have a better view of comparison of scalability and loss in precision, we present just this data for all three slices in Table 4.3. We show a graphical view of the results in Figure 4.2.

From the results, it is obvious that both value and thin slices help in scaling up property checking, with thin slice having a small advantage (14%) over value slice. However, compared to the backward slice, the precision drops considerably (30%) in the case of thin slice, while there is only a marginal drop (3%) for value slice. This shows that value slice is a good compromise between backward and thin slices as it provides considerable scalability with only a marginal loss in precision.

An N outcome in case of value and thin slices does not necessarily mean that property does not hold on the original program. Therefore, for such cases an abstraction refinement step will be needed to get an answer which holds for the original program too. From the results observed, it is clear that such an abstraction refinement step will be required in many more cases with thin slice as compared to value slice. We also expect such abstraction refinement cycles to be shorter for value slice compared to thin slice, because of fewer abstractions introduced in the value slice at first place.

It may be noted that this data is quite closely dependent on the verifier used. So in our case, it is with respect to the verifier SATABS. From these results, we can not conclude if a

[3] Assuming the programs are terminating.

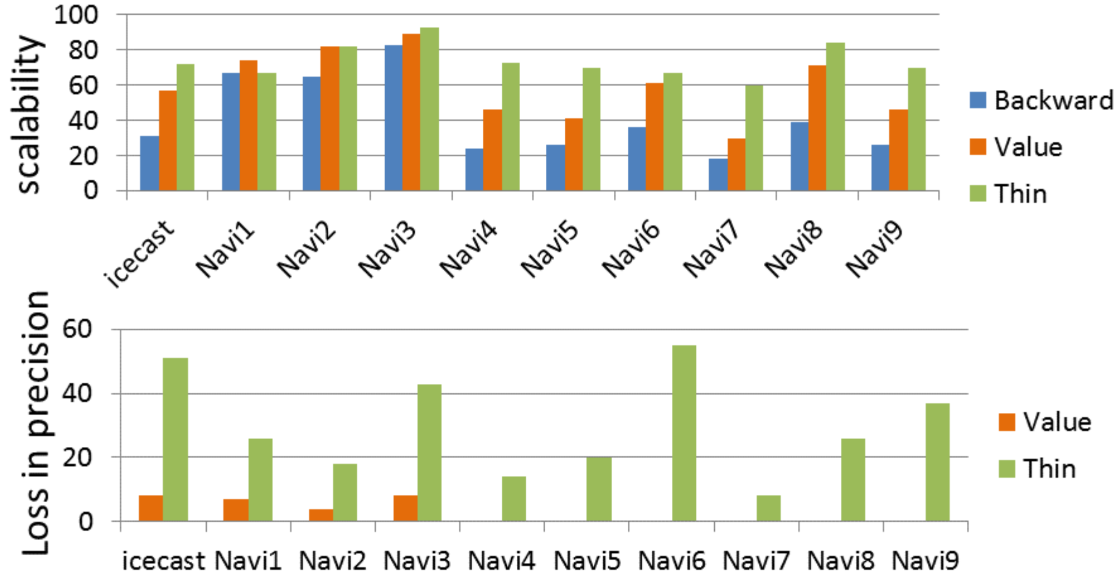


Figure 4.2: A graphical view of scalability and precision of value slice

similar result will be exhibited by use of some other verifier. However, we believe that a similar trend is expected even with other verifiers.

As mentioned earlier, the results reported in the tables of Table 4.2 are with a timeout limit of 3 minutes. Note that we had run the experiments with three different timeouts: 1 minute, 2 minutes and 3 minutes, to see the impact of increase in timeout on the number of cases of different outcomes. We show this data in Table 4.4(a), where we show the number of cases with outcomes of Timeout, Yes(Y), No(N) and any other. In columns (b), (c) and (d), we show the number of cases using backward slice for timeouts of 1 minute, 2 minutes and 3 minutes, respectively. We show similar data for value and thin slices in columns (e), (f) and (g), and columns (h), (i) and (j), respectively. From this data, we observe that value slice benefits more from timeout increase than backward slice.

We also recorded the number of CEGAR loop iterations, number of predicates generated,

and total time taken in running SATABS on individual cases. We show this data in Table 4.4(b). In each row we show the data related to the instances for which outcome using backward and value slice was as given in column (a) and column (b), respectively. In column (c), we show the number of such instances with the given outcome combination. We show the average number of CEGAR loop iterations and predicates, and the average time taken in seconds using backward slice in columns (d), (e) and (f), respectively. Similarly, in columns (g), (h) and (i), we show the corresponding data when value slice is used. Consider the cases where the outcome is "Yes" or "No" using backward and value slices. It can be seen that the use of value slice takes lesser number of CEGAR loop iterations, generates lesser number of predicates, and takes lesser time than backward slice.

4.2.4 Imprecision in property checking with value slice

We observed that there are indeed cases where the verifier could prove the property on a backward slice, but the property did not hold on the corresponding value slice. We analysed and found that there can be a scenario where a condition controlling the property location can also influence the outcome of the property checking. And, if such condition is not identified to be value impacting for the property, as per our definition of value impacting conditions, the condition will be abstracted in the value slice, resulting in a possible violation of the property. To illustrate this, we show an example in Figure 4.3. Clearly, the assertion in the program at line 15 holds true, because `(i < 4)` $\implies$ `(j < 4)` is an invariant after the loop, The backward slice and the value slice for the program, with respect to the assert expression, are shown in Figure 4.3(b) and 4.3(c), respectively.

Since the variable `k` is not at all relevant to the assert at line 15, its computation will not be a part of the backward or value slice. The loop at line 5, computing the value of the variable `j`, will have to be part of these two slices as `j` appears in the assert expression. In addition, since the assert is control dependent on the condition `(i < 4)` at line 14, the condition will be a part of the backward slice. Clearly, the assertion holds in the backward slice as expected, and the same is verified by SATABS. However, as per our definition of value impacting conditions, it is clear that the condition at line 14 is not value impacting for the assert at line 15. Therefore, the condition gets abstracted in the value slice. As a result, the assertion no longer holds in

```
1  main()
2  {
3   int i=0,j=0,k=0;
4   int a[5];
5   while (j < 5)
6   {
7    if (a[j]==0)
8     break;
9    if (a[j] > 100)
10    k++;
11   j++;
12   i++;
13  }
14  if (i< 4)
15   assert(j < 4);
16 }
```

(a) Original program

```
1  main()
2  {
3   int i=0,j=0;
4   int a[5];
5   while (j < 5)
6   {
7    if (a[j]==0)
8     break;
9
10
11   j++;
12   i++;
13  }
14  if (i< 4)
15   assert(j < 4);
16 }
```

(b) Backward slice

```
1  main()
2  {
3   int j=0;
4   int a[5];
5   while (j < 5)
6   {
7    if (a[j]==0)
8     break;
9
10
11   j++;
12
13  }
14  if (*)
15   assert(j < 4);
16 }
```

(c) Value slice

Figure 4.3: Example showing limitation of value slice over backward slice

the value slice. For example, consider an execution of the value slice where the array a gets initialised as $\{1, 2, 3, 4, 0\}$, and the abstract condition at line 14 is taken as true. The variable j will have the value 4 at the assertion point and, as a result, the assertion will be false. Therefore, on this value slice, SATABS will say that the assertion is violated, which is spurious because the assertion holds on the original program.

Table 4.1: Program size and complexity

(a) Number of functions, their size, and conditions and loops within

Prg	Total size in KLOC	Funcs	Size in LOC			Conditionals			Loops		
			Tot.	Avg.	Max.	Tot.	Avg.	Max.	Tot.	Avg.	Max.
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)
icecast	18	182	4019	22	271	570	3.1	45	67	0.4	6
navi1	41	456	11061	24	178	1196	2.6	20	112	0.2	6
navi2	52	385	12181	32	435	1446	3.8	42	120	0.3	9
navi3	50	347	9460	27	148	1229	3.5	16	101	0.3	6
navi4	166	1623	54018	33	445	4979	3.1	31	658	0.4	16
navi5	156	1472	52082	35	579	5418	3.7	45	459	0.3	18
navi6	162	2037	64832	32	579	7772	3.8	61	443	0.2	16
navi7	350	2978	113904	38	579	11966	4.0	66	858	0.3	16
navi8	366	970	41577	43	744	4769	4.9	94	320	0.3	10
navi9	159	1471	49034	33	487	5246	3.6	40	440	0.3	16

(b) Number of pointer dereferences and array references

Prg	Funcs	Ptr Derefs			Array Refs			Array Refs considered[a]		
		Tot.	Avg.	Max.	Tot.	Avg.	Max.	Tot.	Avg.	Max.
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)
icecast	182	1522	8.4	217	134	0.7	21	65	0.3	11
navi1	456	853	1.9	51	488	1.1	55	208	0.5	32
navi2	385	2046	5.3	276	737	1.9	253	192	0.5	10
navi3	347	846	2.4	52	358	1.0	52	235	0.7	52
navi4	1623	9042	5.6	283	2698	1.7	56	684	0.4	34
navi5	1472	8172	5.6	298	3520	2.4	264	729	0.5	22
navi6	2037	8251	4.1	298	2865	1.4	264	929	0.5	34
navi7	2978	16732	5.6	2745	6261	2.1	264	1503	0.5	32
navi8	970	6131	6.3	298	2539	2.6	264	707	0.7	37
navi9	1471	7618	5.2	298	2811	1.9	264	702	0.5	37

[a] Note that as explained in Section 4.2.2 we pick up maximum 10 array references from each module of proprietary code

Table 4.2: Scalability and precision of property checking based on different kinds of slices

(a) Verification with backward slice. Y and N stand for 'yes' and 'no' answers returned by the property checker. ? stands for 'no decision' and N_S stands for a 'no' that is known to be spurious.

Prg	KLOC	Asserts	Backward Slice			
			Y	N	?	$S(\%)$
(a)	(b)	(c)	(d)	(e)	(f)	(g)
$icecast$	18	65	20	0	45	31
$Navi1$	41	58	39	0	19	67
$Navi2$	52	68	44	0	24	65
$Navi3$	50	80	59	7	14	83
$Navi4$	166	70	17	0	53	24
$Navi5$	156	70	16	2	52	26
$Navi6$	162	70	25	0	45	36
$Navi7$	350	60	11	0	49	18
$Navi8$	366	56	20	2	34	39
$Navi9$	159	50	13	0	37	26

(b) Verification with value slice and thin slice. Y and N stand for 'yes' and 'no' answers returned by the property checker. ? stands for 'no decision' and N_S stands for a 'no' that is known to be spurious.

Prg	Asserts	Value slice						Thin slice					
		Y	N	N_S	?	S	L	Y	N	N_S	?	S	L
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)	(n)
$icecast$	65	21	16	3	28	57	8	1	46	24	18	72	51
$Navi1$	58	38	5	3	15	74	7	25	14	10	19	67	26
$Navi2$	68	52	4	2	12	82	4	40	16	10	12	82	18
$Navi3$	80	55	16	6	9	89	8	31	43	32	6	93	43
$Navi4$	70	28	4	0	38	46	0	27	24	7	19	73	14
$Navi5$	70	24	5	0	41	41	0	25	24	10	21	70	20
$Navi6$	70	42	1	0	27	61	0	15	32	26	23	67	55
$Navi7$	60	18	0	0	42	30	0	11	25	3	24	60	8
$Navi8$	56	38	2	0	16	71	0	27	20	12	9	84	26
$Navi9$	50	22	1	0	27	46	0	13	22	13	15	70	37

Prg	KLOC	Asserts	Scale up (%)			Precision loss (%)	
			Backward	Value	Thin	Value	Thin
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
$icecast$	18	65	31	57	72	8	51
$Navi1$	41	58	67	74	67	7	26
$Navi2$	52	68	65	82	82	4	18
$Navi3$	50	80	83	89	93	8	43
$Navi4$	166	70	24	46	73	0	14
$Navi5$	156	70	26	41	70	0	20
$Navi6$	162	70	36	61	67	0	55
$Navi7$	350	60	18	30	60	0	8
$Navi8$	366	56	39	71	84	0	26
$Navi9$	159	50	26	46	70	0	37
Average			41	60	74	3	30

Table 4.3: Comparison of scalability and (%) loss in precision

Table 4.4: Impact of increase in timeout, and change in CEGAR iterations and time taken

(a) Impact of increasing timeout on number of assertions with different outcome. Timeout of 1 minute(1m), 2 minutes(2m) and 3 minutes(3m) is given.

Outcome	Backward slice			Value slice			Thin slice		
	1m	2m	3m	1m	2m	3m	1m	2m	3m
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)
Yes	237	253	264	300	332	338	202	213	215
No	10	11	11	35	52	54	216	252	266
Timeout	319	202	136	234	118	89	181	117	79
Other	81	181	236	78	145	166	48	65	87

(b) Change in CEGAR iterations and time taken (in seconds) from backward to value slice, with a time out of 3 minutes

Outcome		Number of asserts	Average per assert					
			Backward slice			Value slice		
Backward	Value		Iterations	Predicates	Time	Iterations	Predicates	Time
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)
Yes	Yes	221	2.0	4.4	12.5	1.9	3.5	4.9
No	No	11	3.6	14.4	23.8	2.5	5.6	4.9
Yes	No	17	6.4	36.3	31.8	8.7	28.1	30.6
Timeout	Yes	47	1.8	-	-	2.8	11.6	19.7
Timeout	No	21	16.2	-	-	19.4	36.2	43.8

Chapter 5

Related work

Backward slicing was first introduced for imperative programs by Mark Weiser [71] in 1981. A backward slice of a program with respect to a program point l and a set of variables V, consists of all statements and predicates of the program, which may impact value of the variables in V at the program point l, in some execution of the program. The pair $\langle l, V \rangle$ is called a slicing criterion. Later, Ferrante et al. introduced a representation of the program called *Program Dependence Graph* (PDG) [36], and backward slicing was modeled as a reachability problem over the PDG. To address the issues of feasible paths arising due to procedure calls, Reps et al. introduced a representation of programs called a system dependence graph (SDG) [46], that spans over more than one functions. They presented an algorithm over SDG to compute an inter-procedural backward slice. Silva et al. [67] provide a survey of several variants of backward slicing proposed since its introduction by Weiser. Notable among these variants are forward slicing [7], chopping [48], and assertion based slicing [16, 24, 6]. A forward slice of a program with respect to a slicing criterion $\langle l, V \rangle$, consists of all the statements and predicates of the program which may get impacted by value of the variables in V at the program point l, in some execution of the program. Chopping [48], is a generalisation of slicing and encompasses both backward and forward slicing. It has a pair of slicing criteria, $\langle l_f, V_f \rangle$ called *source*, and $\langle l_b, V_b \rangle$ called *sink*, corresponding to the slicing criteria for the forward and the backward slicing, respectively. Chopping produces a set of the statements that may impact value of the variables in V_b at the program point l_b due to value of the variables in V_f at the program point l_f. The backward slice and its variants have been used for program understanding, debugging, testing,

maintenance, software quality assurance and reverse engineering. Binkley et al. [12] provide a survey of the applications of program slicing.

Comuzzi and Hart made a shift from syntax based slicing to program semantics based slicing by defining a program slicing, called *p-slices* [24], using Dijkstra's weakest precondition (wp) for a given predicate as slicing criterion. Given a program P and a predicate ψ, a subprogram S of P is a *p-slice* if $wp(P, \psi) \equiv wp(S, \psi)$. Genardo Canfora et al. extended this idea to define a conditioned program slicing [16] which is basically a backward slice but only for a set of execution paths, where this set is specified by a first order logic formula on the input variables. Barros et al. extended it further and unified the condition based slicing and p-slices, by taking a precondition and post condition as a slicing criteria, and called it *assertion based slicing* or *specification based slicing* [6]. Given a program P, a precondition φ and a postcondition ψ, such that the Hoare triple $\{\varphi\}P\{\psi\}$ is true then a subprogram S of P is a slice with respect to φ and ψ, if the Hoare triple $\{\varphi\}S\{\psi\}$ remains true.

While these slices are static, that is their stated objective is met by all executions of the program, there has been considerable interest in dynamic slices also that are computed with respect to a particular execution of the program. A dynamic slice [54] of a program is set of statements and predicates that impact the value of a set of variables V at a program point l, in an execution of the program for input i. Therefore, the triplet $\langle i, l, V \rangle$ forms a slicing criterion for a dynamic slice. Basically, a dynamic slice matches the behaviour of the original program with respect to value of variables in V at the program point l for a single execution of the program. A dynamic slice is very useful in debugging and understanding program behaviour for specific kind of inputs.

Restricted to the slicing criterion, all these techniques produce slices with behaviour equivalent to the original program. However, to the best of our knowledge, the idea of producing slices which are not exactly equivalent to original program for the selected slicing criterion has not been explored at depth. This is so mainly because the uses to which slicing has been put to, demand that behaviour of slices remain equivalent to original program behaviour. Our interest in slicing is more to reduce the program size and its complexity, so that property checking can scale up by excluding irrelevant portion of the code, and at the same time retaining the code which impacts the property. We want to differentiate between code which *impacts more* and the code which *impacts less*. As a result, we want to be more aggressive in slicing to reduce

the program size as much as we can while retaining the vital part of the code impacting the property.

The semantics based slices [16, 24, 6] have also been claimed to help program verification of large programs. However, at some or the other point during computation of the slices in these techniques, one has to compute weakest precondition for the predicate given as a slicing criterion. Since computing weakest precondition in presence of loops requires loop invariant, computation of such slices itself can become as hard as the original verification problem. So while these slicing techniques may help program verification once such slices are generated, computation of slices themselves becomes as hard a problem as the original verification problem. In contrast, our idea of value slice is entirely based on static analysis and does not require any weakest precondition computation. In addition, once the weakest preconditions have been computed, producing the semantic slices needs an order of complexity of $O(N^2)$ where N is the number of nodes in CFG, as stated by Commuzi and Hart [24]. In contrast, the complexity of the value slice computation is $O(N + E)$ where E is number of edges in the CFG and is typically of the same order as N.

To the best of our knowledge, thin slicing [68] proposed by Sridharan, Fink and Bodik, with an aim to help debugging, is the first approach that produces a slice whose behaviour may differ from the original program with respect to the slicing criterion. A thin slice retains only those statements that the variables in the slicing criterion are data dependent on and abstracts out all the predicates. This variant of slicing comes closest to our method as it is static analysis based and produces an aggressive slice. Although, this slicing was designed to help in debugging, we wanted to explore if the idea can be helpful in scaling up the property checking also. While the slices produced by thin slicing are indeed of smaller size, our experiments show that these slices are too imprecise for property checking. Interestingly, the authors do mention the importance of identifying the predicates that we include in the value slice in a concrete form. However, in their approach such predicates are added manually on demand during debugging. Our central claim about value slice is that it is a good compromise between precision and scalability. To substantiate the claim, on one side we compare value slice with backward slice and on other side we compare it with thin slice. This comparison is in no way intended to undermine the usefulness of thin slice in debugging and program understanding.

Part II

Scaling up Property Checking of Array Programs

Chapter 6

Background

In this chapter, we describe some basic concepts and terminology which we shall be using in discussing our idea on scaling up property checking for array processing programs using loop shrinking.

6.1 Imperative programs and states

We shall present our ideas in the context of imperative programs modeled in terms of assignment statements, conditional statements, while loops, and function calls. We assume that conditional expressions have no side effects. We restrict ourselves to goto-less programs with single-entry single-exit loops; it makes for an easier formal treatment of our method without losing expressibility.

Let Var be the set of all variables in a program P and Val be the set of possible values which the variables in Var can take. A *program state* is a valuation of the variables in Var that is consistent with their types. It is represented by a map $\sigma : \mathit{Var} \rightarrow \mathit{Val}$. For a variable v, $\sigma(v)$ denotes the value of v in the program state σ.

Property-checking will be expressed using the well-known formalism called Hoare logic [44], also known as Floyd-Hoare logic, which is a formal system to reason about correctness of the programs. We use the concept of *Hoare triple* of this system to specify our property checking problem. Property checking will be expressed as Hoare triple and denoted as $\{\Phi\}P\{\psi\}$. Here Φ and ψ are first order logic formulas representing sets of states, and P is a program. Such a Hoare triple is said to be *valid* if starting from an initial state satisfying Φ, if

the execution of P terminates, the final state satisfies ψ. In the context of verifying quantified properties using loop shrinking, we only consider programs that are deterministic and guaranteed to terminate, and thus the issue of termination will not arise in the rest of the discussion. In general, the validity of a Hoare triple $\{\Phi\}P\{\psi\}$ is not equivalent to the invalidity of the Hoare triple $\{\Phi\}P\{\neg\psi\}$. A fact that we shall make use of is that in the special case when Φ represents a single program state σ, the program being terminating and deterministic, its execution starting from Φ, will result into a single unique final state, say σ'. Therefore, the validity of $\{\sigma\}P\{\psi\}$ is equivalent to the invalidity of $\{\sigma\}P\{\neg\psi\}$. As a result, we can represent a Hoare triple $\{\sigma\}P\{\psi\}$ as a proposition p and the Hoare triple $\{\sigma\}P\{\neg\psi\}$ will be the proposition $\neg p$.

6.2 Bounded model checking

Our technique makes good use of bounded model checkers (BMCs). Given a program P and a property ψ, a BMC searches for a counterexample to ψ in executions of P whose length is bounded by some integer n. If it finds a counterexample to ψ within the bound n, then it reports the property as unsafe. However, if it does not find a counterexample and P has executions that are longer than n then the BMC cannot report the program as either safe or unsafe. Bounded model checking of a program means unwinding the loops of the program up to a pre-defined limit and then model checking the resulting program. BMCs are, therefore, very useful in finding bugs (violation of the property) but not very effective in proving properties. However, for a program whose loops can be completely unwound, if a BMC does not find a bug in the program resulting after unwinding of the loops, it means that the original program is correct, i.e. property is proved. In our technique under discussion, we will make use of this fact and will attempt to present only such kind of abstract programs to a BMC for model checking. Industrial strength BMCs exist [22, 62, 35] and are widely used to detect property violations in safety critical software. CBMC [22] developed by Daniel Kroening et al. is the BMC that we have used in our experiments.

6.3 Loop acceleration

Loop acceleration [50, 15, 2] is a technique commonly used for finding loop invariant. Loop acceleration captures the effect of a loop through closed-form expressions that give the value of variables at the beginning of an iteration in terms of the initial state and the iteration count. Variables whose values can be expressed in this manner are called *accelerable*. To illustrate, consider the code snippet of Figure 6.1(a). In this code snippet, the value of variable `i` at the beginning of an iteration number n is expressible as $n-1$. Similarly, the value of `j` and `k` can be expressed as $(n-1)*2$ and 2^{n-1}, respectively. Value of `t` is more complex but still expressible as $\lfloor(3*n-2)/2\rfloor$. However, value of `f` is not expressible as a closed form expression in terms of iteration count and initial state, hence `f` is not *accelerable*. There have been considerable work involving loop acceleration and there are tools and techniques [39, 50, 29, 55] available to identify loop accelerable variables and corresponding accelerated expressions. However, detecting variables like `t` as accelerable is hard and will depend upon sophistication of the underlying technique. We assume that some such tool is available to identify accelerable variables and their corresponding accelerated expressions. While our approach does not require us to identify all accelerable variables, the precision of the result does depend on the identification of as many accelerable variables as possible.

6.4 Programs and properties of interest

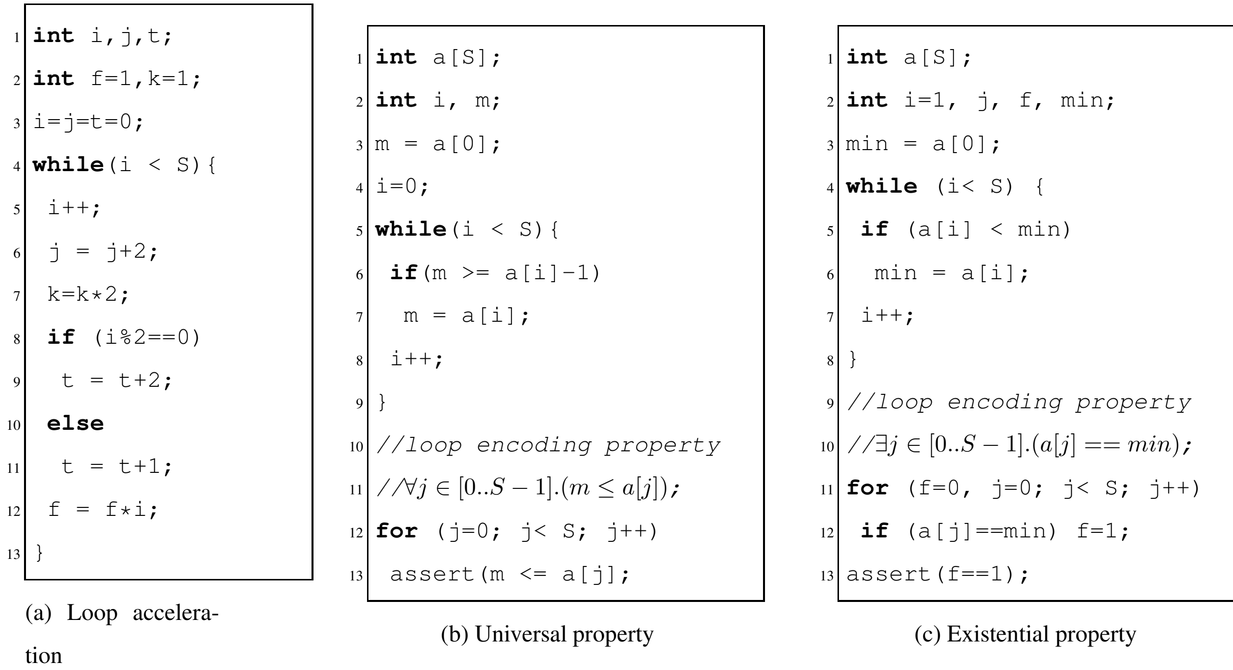
Our focus is on programs that process arrays in loops which we assume to always terminate. The property to be checked is encoded by a fragment of code that follows the array processing loop. If the property is expressed as a loop, we denote it in our discussions as a universally or existentially quantified formula over the elements of the array. As an illustration, the second loop of Figure 6.1(b) encodes the universal property:

$\forall j.0 \leq j < N \implies m \leq a[j]$.

Similarly, the second loop of Figure 6.1(c) encodes the existential property:

$\exists j.0 \leq j < S \wedge min = a[j]$.

In particular, we consider program fragments R ; Q ; ψ, in which R is a simple loop possibly manipulating arrays, Q is a loop free (possibly empty) sequence of statements and



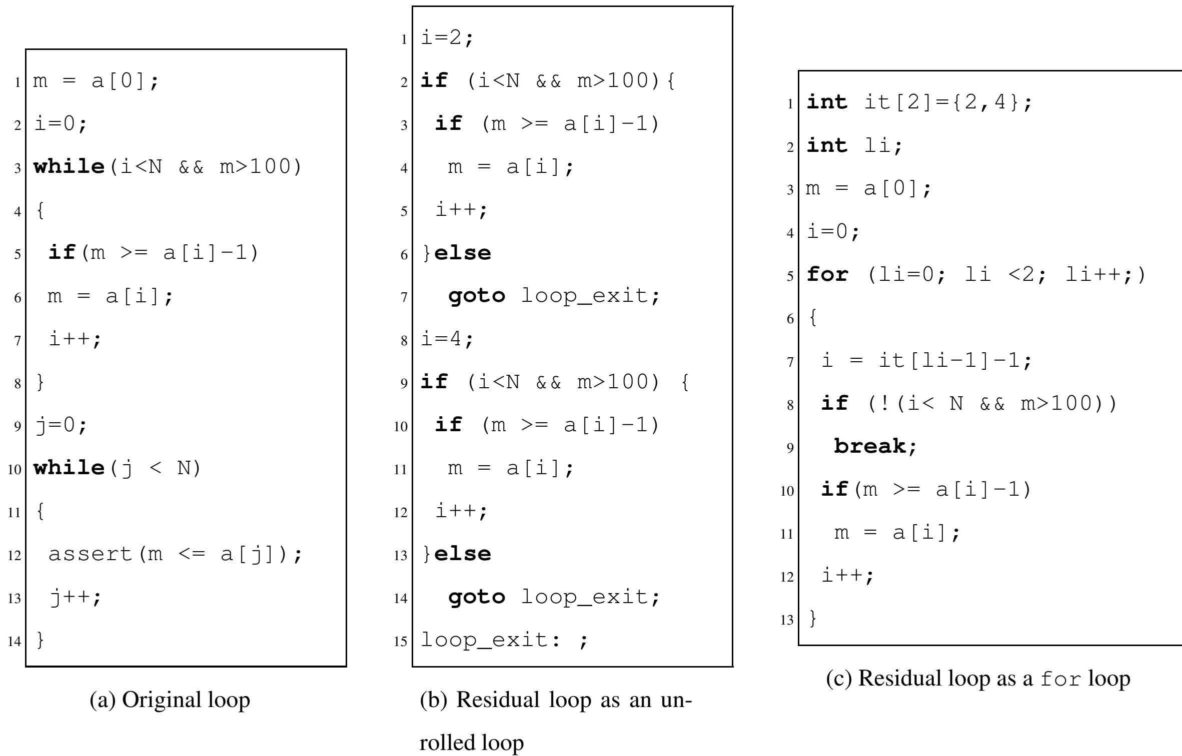
(a) Loop acceleration (b) Universal property (c) Existential property

Figure 6.1: Illustration of loop acceleration and property checking loops

ψ is the property to be checked. We call $R\ ;\ Q$ as an *array processing loop*. In addition, we assume that for the loop R, an upper bound on its number of iterations can be computed through static analysis [31]. The property ψ is assumed to have at most one quantifier. We assume that the array-processing loop and the loop which checks the property have the same number of iterations. Finally, since the quantified variable ranges over a finite domain (iteration counts of a finite loop), it is useful to think of ψ as a set of quantifier-free formulas, connected by conjunction in the case of $\forall$ and disjunction in the case of $\exists$.

6.4.1 Iteration sequence

An *iteration sequence* is a strictly ascending sequence of numbers, representing iteration counts. Iterations of a loop are counted from 1. We shall use $U \sqsubset T$ to mean that U is a strict subsequence of T. For example, $[2, 5] \sqsubset [1, 2, 4, 5, 6]$. The notation $i : T$ will represent a sequence whose first element is i and the suffix of the sequence, excluding the first element, is same as T. Further, we shall write $\mathcal{P}_k(T)$ to denote the set of all k-sized subsequences of a sequence T. For example, if $T = [1, 2, 4, 5]$ then $\mathcal{P}_3(T) = \{[1, 2, 4], [2, 4, 5], [1, 2, 5], [1, 4, 5]\}$.



(a) Original loop

(b) Residual loop as an unrolled loop

(c) Residual loop as a `for` loop

Figure 6.2: Illustration of residual loop for iteration sequence [2,4]

6.4.2 Residual loop and residual property

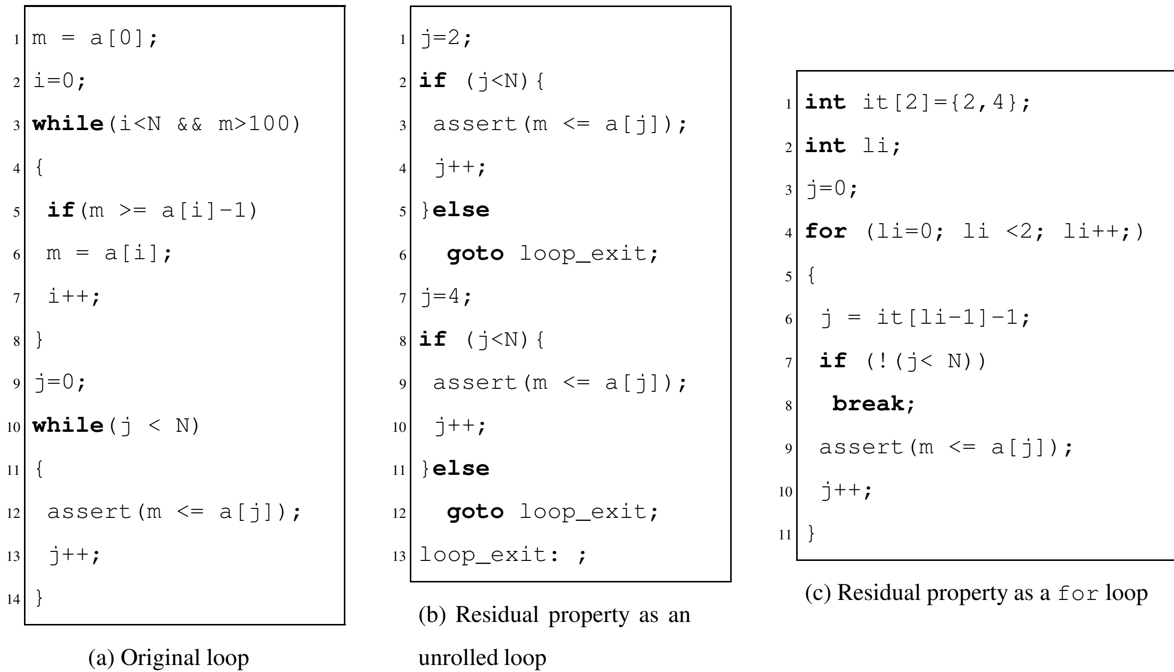
Consider a program P consisting of an array processing loop $L \equiv$ `while` $(C)\ \{B\}Q$ followed by a code fragment that checks the property ψ. Let $T = [j_1, j_2, ..., j_n]$ be an arbitrary iteration sequence of the loop. We define the *residual loop* for the iteration sequence T, denoted as L_T, as the statements $\{S_{j_1}; S_{j_2}; \ldots S_{j_n};$ `loop_exit:` $Q\}$, where each S_{j_r} is

$\{A_{j_r}$; `if` $(C)\ \{B\}$; `else goto loop_exit;`$\}$.

Here A_{j_r} is the sequence of assignment statements, with one assignment per accelerable variable, such that each statement assigns to an accelerable variable its corresponding accelerated expression that defines its value at the beginning of iteration j_r. Obviously, for $T = j : T'$ with T' being nonempty, $L_T = S_j; L_{T'}$.

As an illustration, the code fragment in Figure 6.2(b) is the residual loop for the iteration sequence $[2, 4]$, for the first loop (lines 3-8) of the code snippet given in Figure 6.2(a). In the code snippet of Figure 6.2(c), we show the same residual loop but it is encoded as a `for` loop.

Note that the residual loop, given in Figure 6.2(c), has two iterations. Its first iteration (iteration 1) represents the iteration 2 of the original loop and the second iteration (iteration 2)



(a) Original loop

(b) Residual property as an unrolled loop

(c) Residual property as a `for` loop

Figure 6.3: Illustration of residual property for iteration sequence [2,4]

represents iteration 4 of the original loop. An iteration j of a residual loop which represents iteration j' of the original loop will also be called *representative-of-iteration* j'. So iteration 2 of the residual loop in the example is *representative-of-iteration* 4.

If the loop iterates for a maximum of N times, then $[1, 2, ..., N]$ will be called the *complete iteration sequence* of the loop. It is obvious that, the residual loop $L_{[1,2,...,N]}$ represents an unrolling of L and the two are semantically equivalent. For this residual loop, its every iteration i is *representative-of-iteration* i. Similarly, for the iteration sequence T=$[j_1, j_2, ..., j_m]$ and the property ψ, we define the *residual property* ψ_T as set of clauses $\{\psi_{j_1}, \psi_{j_2}, \ldots, \psi_{j_m}\}$.

As an illustration, the code fragment in Figure 6.3(b) encodes the residual property for the iteration sequence $[2, 4]$ for the property encoded in the second loop (lines 10-14) of the code snippet given in Figure 6.3(a). In the code snippet of Figure 6.3(c), we show the same residual property though encoded as a `for` loop.

6.5 State approximation for residual loops

Let the set of initial states at the beginning of a loop L be φ. Obviously, the set of states at the beginning of the iteration numbered 1 would be φ. The set of states at the beginning of

an iteration numbered i, where $i > 1$, would be given by $sp((S_1; S_2; ...; S_{i-1}), \varphi)$, the strongest post-condition of $S_1; S_2...S_{i-1}$ with respect to φ. We can define the set of states at the beginning of the iteration j for a residual loop also in a similar manner. Obviously, the states at the beginning of a residual loop also will be same as the states at the beginning of the original loop, i.e. φ. However, for a residual loop, say for an iteration sequence $[j_1, j_2, ..., j_n]$, the states at the beginning of some iteration numbered $i > 1$ will depend upon the iteration numbers $j_1, j_2, ..., j_i$ of the iteration sequence. Obviously, the iteration i of the residual loop is representative-of-iteration j_i.

Consider an iteration sequence $U = [1, 3, 5]$ and its residual loop L_U. The second and third iterations of the residual loop are *representative-of-iterations* 3 and 5, respectively. Note that the states at the beginning of second iteration will be $sp(S_1, \varphi)$ and that at the beginning of third iteration of the residual loop will be $sp((S_1; S_3), \varphi)$. Consider another iteration sequence $U' = [2, 4, 5, 7]$ and its residual loop $L_{U'}$. For this residual loop, the states at the beginning of the third iteration, which is *representative-of-iteration* 5, will be $sp((S_2; S_4), \varphi)$. This is different from $sp((S_1; S_3), \varphi)$, the states at the beginning of third iteration (which is also *representative-of-iteration* 5) of the residual loop L_U. Therefore, for a given i, in the residual loops of different iteration sequences containing i, the states at the beginning of the iteration which is *representative-of-iteration* i may be different and will depend upon the composition of the iteration sequence of the residual loop. However, we sometimes have to estimate these set of states in the context of an arbitrary iteration sequence T which contains the iteration i, and in which the sequence of iteration numbers preceding i is not exactly known. Therefore, instead of the earlier exact calculation, we over-approximate the set of states at the beginning of the iteration which is *representative-of-iteration* i, denoted φ_i, through the recurrences:

$\varphi_1 = \varphi$, and $\varphi_i = sp(S_{i-1}, \varphi_{i-1}) \cup \varphi_{i-1}$.

The additional term φ_{i-1} in the union accounts for the possibility that the iteration $i - 1$ may not precede i in T, and therefore the set of states at the beginning of *representative-of-iteration* i should also include the states at the beginning of the *representative-of-iteration* $i-1$. For a loop having N iterations, it is obvious that following holds:

$\forall j, j' \in [1..N] \,.\, j \leq j' \implies \varphi_j \subseteq \varphi_{j'}$.

Chapter 7

Loop shrinkability

We now characterize the conditions under which the behavior of an array-processing loop L, with respect to a property ψ, can be over-approximated by a residual loop L_U in association with the corresponding residual property ψ_U, where U is an iteration sequence. The iteration sequence U consists of fewer iterations than the original program, i.e. $U \sqsubset [1, 2, \ldots, N]$, and the iteration numbers of U are chosen non-deterministically.

7.1 Definition of shrinkable loops

To be able to characterise the loops which can be over-approximated by a residual loop with respect to a property, we first formally define this over-approximation as follows:

Definition 7.1 *(Shrinkable loops) Consider a program consisting of a loop L and a property ψ to be checked. Let T represent the complete sequence of iterations of the loop. The loop is said to be shrinkable with respect to ψ and with a shrink-factor k, $0 < k < |T|$, if and only if, starting from any state $\sigma \in \varphi$, the loop L satisfies ψ whenever the residuals L_U of each k-length subsequence U of T satisfies the corresponding residual property ψ_U. Formally:*

$$\forall \sigma \in \varphi : ((\forall U \in \mathcal{P}_k(T) : \{\sigma\}L_U\{\psi_U\}) \implies \{\sigma\}L\{\psi\}) \tag{7.1}$$

It will often be useful to read the formal description above in a contrapositive manner. The same is depicted in the Figure 7.1. It reads as: starting from a state σ in φ, if the loop L fails to satisfy ψ, then the failure is also witnessed by a k-length iteration sequence U such

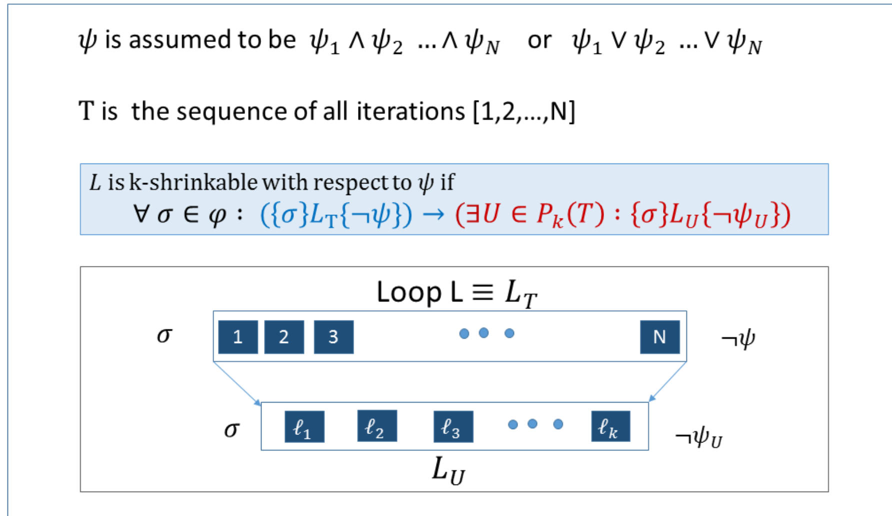
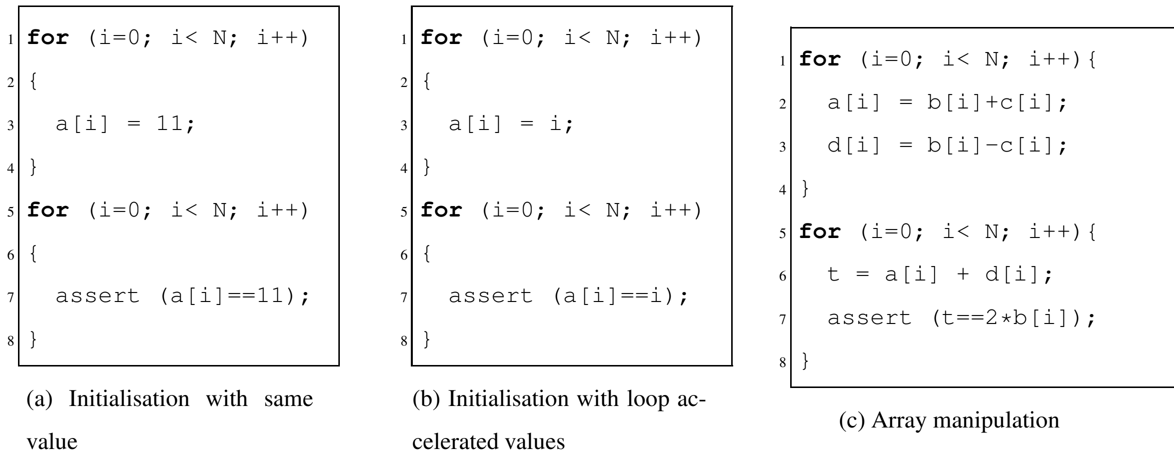


Figure 7.1: Illustration of contrapositive view of shrinkable loops definition

that, starting from the same state σ, the residual loop L_U also fails to satisfy the corresponding residual property ψ_U. Note that executions of both L and L_U begin in the same state σ in φ.

A shrinkable loop with a shrink-factor k will be called k-*shrinkable*. If we know that a loop is k-shrinkable, we can construct an abstract program that non-deterministically chooses an iteration sequence of size k, runs the residual loop, and then checks the corresponding residual property. Assuming k is a small number, we can check the property on this abstract program using a BMC. If the property holds, then shrinkability guarantees the correctness of the original program too. In contrast, the alternative is using a BMC directly on the original program. This alternative will require the BMC to unroll the loop to its complete bound, i.e. enumerate all the iterations, and the same will not be scalable when the loop has a large bound. However, a counterexample in the abstract program does not necessarily imply a violation of the property in the original program, except in situations described below.

Suppose the non-accelerable variables have no loop-carried dependence [3]. In that case the value assigned to such variables in one iteration has no bearing on the value assigned to same variables in another iteration. In addition, consider the case when for all i, all the array elements appearing in the residual property ψ_i get modified (if at all) only in the i_{th} iteration of the array processing loop. In other words, the loop iterations are independent of each-other.



(a) Initialisation with same value

(b) Initialisation with loop accelerated values

(c) Array manipulation

Figure 7.2: Illustration of programs with no loop carried dependence

In this situation, if the property ψ gets violated in the original program P, due to violation of a clause $\psi_{i'}$, say, then in the program consisting of the residual loop $L_{[\,i'\,]}$, constructed on the basis of the only iteration i', the residual clause $\psi_{[\,i'\,]}$ will get violated. Thus, a loop without loop-carried dependence is 1-shrinkable. More significantly, if the property being tested for such programs is universal, the converse is also true. I.e. if in the residual loop corresponding to an iteration sequence consisting of a single iteration, the corresponding residual property gets violated, then the original program will also not satisfy its specified property. The code snippet of Figure 7.2(a), that initialises an array with a single value and then checks that all the elements of the array are indeed initialised with that value, is an example of such programs. In the same figure, we illustrate some other programs of this category.

Note that according to Definition 7.1, if a program P satisfies its property ψ, then the loop constituting the program is k-shrinkable for any shrink-factor $k > 0$. Similarly, a loop with a bound of m iterations is trivially m-shrinkable. Obviously, if the shrink-factor is small, then the abstract program with a smaller length iteration sequence loads the verifier to a lesser extent and thus offers greater chances of verifier returning an answer. Therefore, we are interested in finding shrink-factors that are much smaller than the loop bound.

However, finding out statically whether a loop is shrinkable is difficult as we illustrate through an example. Consider the two code snippets `min2` and `lmin` in Figure 7.3 which have similar structure as well as similar nature of computation. Assume that, for both code snippets, the elements of the array `a` get initialised with non-deterministic values in every execution. In

```
int i, min, a[S];
min = a[0]; i=1;
while (i< S) {
  if (a[i] < min) min = a[i];
  i++;
}
assert ∃j ∈ [0..S − 1].(a[j] == min);
```

(a) Code snippet `min2` : k-shrinkable with $1 \leq k \leq S$

```
int i, m, d, a[S];
m = a[0]; i=0;
while(i < S){
  if(m >= a[i]-1) m = a[i];
  i++;
}
assert ∀j ∈ [0..S − 1].(m ≤ a[j] + d);
```

(b) Code snippet `lmin` : (d+2)-shrinkable

Figure 7.3: Examples illustrating similar loops having different shrinkability

addition, assume that for the code snippet of Figure 7.3(b), the variable `d` is given some non-negative integer value. The code snippet `min2` computes the minimum of the array, and the property being checked at the end is that the computed minimum value in variable `min` is equal to some element of the array. It is obvious that the code snippet `min2` correctly computes the minimum, and therefore, it is correct with respect to the asserted property. Thus the loop in the code snippet is k-shrinkable for all k from 1 to `S`. The second code snippet `lmin` is similar to our motivating example with a property that asserts that the final value of `m` does not exceed any array element by more than a value `d`. Observe that, starting with the second element of the array, if the value of each element exceeds the value of the previous element by 1, then `m` will exceed the first element by $\mathtt{S} - 1$. Therefore, the property does not hold for $\mathtt{d} < \mathtt{S} - 1$. It turns out that the loop in `lmin` is shrinkable with a shrink-factor k, where k is lower of $\mathtt{d} + 2$ and `S`. This illustrates the difficulty of analytically finding whether a given loop is shrinkable, and based on the development in rest of this section, we shall suggest an empirical method in Section 7.4.

7.2 Identifying shrinkable loops

While Definition 7.1 lays down the consequences of a loop being shrinkable, it does not provide a convenient method to decide whether a loop is shrinkable and to find the shrink-factor. To get around this problem, we first extend the notion of shrinkability from loops to arbitrary iteration sequences. We then identify the conditions under which the shrinkability of smaller

iteration sequences (that are checked explicitly) would imply the shrinkability of larger iteration sequences and eventually of the entire loop.

7.2.1 Sequence shrinkability

We adapt the definition of loop shrinkability to define sequence shrinkability as follows:

Definition 7.2 *(Shrinkable iteration sequence) Consider a program consisting of a loop L and a property ψ to be checked. Let T be an iteration sequence, and let j be the first iteration in T. The sequence T is k-shrinkable with respect to ψ, $0 < k < |T|$, if and only if, starting from every state $\sigma \in \varphi_j$, the residual loop L_T satisfies the residual property ψ_T whenever the residual loops L_U of each k-length subsequences U of T satisfies the corresponding residual property ψ_U. Formally:*

$$\forall \sigma \in \varphi_j.((\forall U \in \mathcal{P}_k(T).\{\sigma\}L_U\{\psi_U\}) \implies \{\sigma\}L_T\{\psi_T\}) \quad (7.2)$$

The only difference between the notion of shrinkability of a loop and an iteration sequence is the starting state σ, which, in this case, is from the set φ_j. Recall that φ_j is an approximation of the set of states at the beginning of an iteration, which is the *representative-of-iteration* j, in the residual loop of any iteration sequence that contains j. As in the case of loops, by k-shrinkable sequence we shall mean a shrinkable sequence with shrink-factor k. It is obvious that a loop is k-shrinkable if the sequence consisting of all iterations of the loop is k-shrinkable.

A contrapositive view of the definition is depicted in Figure 7.4. It reads as, starting from a state σ in φ_j, if the residual loop L_T fails to satisfy ψ_T, then the failure is also witnessed by a k-length subsequence $U \sqsubset T$ whose residual loop L_U also fails to satisfy the corresponding residual property ψ_U. Note that, again, executions of both L_T and L_U begin in the same state σ in φ_j.

As an illustration of an iteration sequence that is not shrinkable, consider the code snippet `lmin` in Figure 7.3(b) with `d` $= 0$. Consider the array `a` with its initial two elements as $\{0,1\}$ and the iteration sequence $T = [1, 2]$. The residual loop of T computes `m` $= 1$ for which the residual property ψ_T does not hold (`m` $>$ `a[0]`). However, the residual loop for every 1-length sequence satisfies its residual property, and thus T is not 1-shrinkable. Also notice that, when `d` $= 0$, the code snippet is same as the motivating example of Figure 1.3(a), except for array

T is a sequence $[j = \ell_1, \ell_2, \ell_3, \ldots, \ell_n]$

T is k-shrinkable with respect to ψ if
$\forall\, \sigma \in \varphi_j :\ (\{\sigma\}L_T\{\neg\psi_T\}) \rightarrow (\exists U \in P_k(T) : \{\sigma\}L_U\{\neg\psi_U\})$

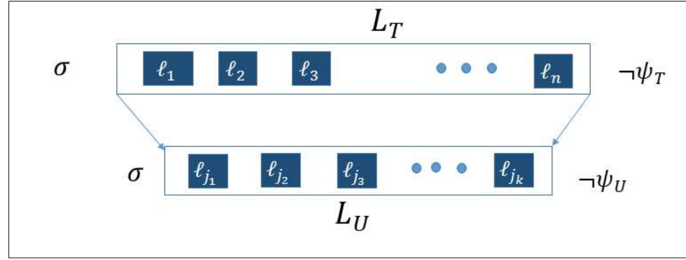


Figure 7.4: Illustration of contrapositive view of sequence shrinkability definition

initialisation. Thus, from the observations in Section 1.3.2, every iteration sequence of length 3 is 2-shrinkable.

7.2.2 Problem with sequence shrinkability definition

We are interested in a method that guarantees that a loop is shrinkable by examining iteration sequences up to a given length. More specifically, we are interested in finding a pair of numbers n and k, such that k-shrinkability of all sequences of length between $k + 1$ and n would imply the k-shrinkability of every sequence longer than n—in particular, the complete set of iterations comprising the loop. If we can identify the condition under which we can find such a pair, then our strategy would be to establish the k-shrinkability of sequences up to n empirically, and the k-shrinkability of all iteration sequences with lengths greater than n will follow.

Since empirical verification of k-shrinkability for all subsequences of length between $k+1$ and n would be costly, we shall consider the case where $n = k+1$, i.e. we shall empirically find a k such that all $k + 1$ length iteration sequences are k-shrinkable. The identified condition will then ensure the k-shrinkability of sequences larger than $k + 1$. Notice that the generalization from $k + 1$ to larger sequences does not happen unconditionally. As an example, consider the

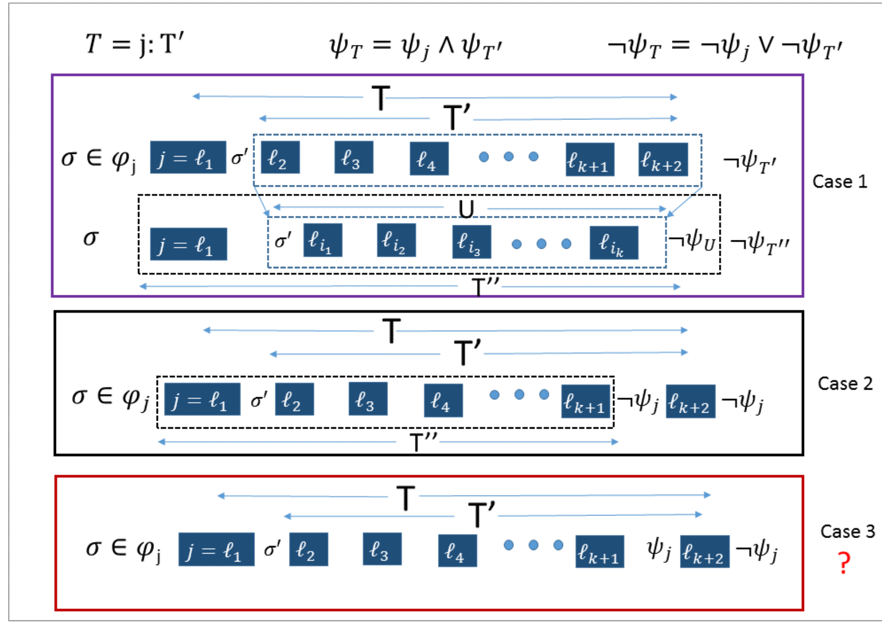


Figure 7.5: Illustration of problem with adapted definition of sequence shrinkability

code snippet `lmin` in Figure 7.3. For `d=2`, all the iteration sequences of size 3 are 2-shrinkable but not all sequences of size 4 are 2-shrinkable.

To derive the required condition, let us examine what it takes to ensure the k-shrinkability of a sequence of length $k+2$, given the k-shrinkability of all sequences of length $k+1$. For simplicity of exposition, first we examine the issues related to conjunctive properties. The treatment for disjunctive properties is very similar, and we shall give final solution later to address the similar issues related to such properties.

Consider an iteration sequence T of size $k+2$. Represent T as $j : T'$. Taking a contrapositive view of the condition for shrinkability, assume that starting from σ, the residual property ψ_T is violated for the program L_T i.e. $\{\sigma\}L_T\{\neg\psi_T\}$ is true. Given that all sequences of length $k+1$ are k-shrinkable, it suffices to find a subsequence $T'' \sqsubset T$ of length $k+1$ such that $\{\sigma\}L_{T''}\{\neg\psi_{T''}\}$ is true. k-shrinkability will then ensure that there is a k-length subsequence U of T'' (and thus also a subsequence of T) such that $\{\sigma\}L_U\{\neg\psi_U\}$. Let the state after the iteration j in the sequence be σ'. Clearly $\{\sigma'\}L_{T'}\{\neg\psi_{[j]} \vee \neg\psi_{T'}\}$ is true. We have shown three possible scenarios in the Figure 7.5.

1. Consider the case when $\psi_{T'}$ is violated, depicted as 'Case 1' in the Figure 7.5. Since

T' is k-shrinkable, it is possible to find a k-length subsequence U within T' such that starting from σ', ψ_U would be violated after L_U. Now consider the iteration sequence $T'' = j : U$. Clearly, starting from σ, $\psi_{T''}$ would be violated after executing $L_{T''}$, and thus the $k+1$-length sequence that we want is T''.

2. Now suppose that ψ_T is violated only because the clause $\psi_{[j]}$ is violated. There are two subcases to be considered. In the first, assume that the violation of $\psi_{[j]}$ also shows up in the state after iteration $k+1$. The scenario is depicted as 'Case 2' in the Figure 7.5. In this case the T'' that we want is the $(k+1)$-length prefix of T.

3. The interesting case is when the violation of ψ_T is solely because of $\psi_{[j]}$, and this violation of $\psi_{[j]}$ does not show up in the state after iteration $k+1$. The scenario is depicted as 'Case 3' in the Figure 7.5. In this case, the definition of shrinkability, in its current form, does not enable us to produce the required sequence T''. To remedy this, notice that for the subsequence T', there is an iteration in the past, namely j, whose clause $\psi_{[j]}$ has been violated. If we revise the definition of k-shrinkability of iteration sequences (Definition 7.2) to ensure that this violation also shows up at the end of some k-length subsequence U' of T', then we are done. The required $k+1$-length subsequence T'' in this case would be $j : U'$ for which $\{\sigma\}L_{T''}\{\neg\psi_{T''}\}$ would be satisfied.

7.2.3 Revised definition of sequence shrinkability

We call this modification, introduced in the previous section, as *past-preservation*. The revised definition of shrinkability that includes past-preservation is presented below.

Definition 7.3 *(Shrinkable iteration sequence, revised for universal properties) Consider a program consisting of a loop L and a universal property ψ to be checked. Let T be an iteration sequence, and let j be the first iteration in T. In addition, let i stand for any iteration before j. The sequence T is k-shrinkable with respect to a property ψ, $0 < k < |T|$, if and only if, starting from every state $\sigma \in \varphi_j$ the residual loop L_T satisfies $\psi_T \wedge \psi_{[i]}$ whenever the residual loops L_U of each k-length subsequences U of T satisfy the corresponding property $\psi_U \wedge \psi_{[i]}$. In other words:*

$$\forall \sigma \in \varphi_j . \forall 0 \leq i < j . ((\forall U \in \mathcal{P}_k(T) . \{\sigma\} L_U \{\psi_U \wedge \psi_{[i]}\}) \implies \{\sigma\} L_T \{\psi_T \wedge \psi_{[i]}\}) \tag{7.3}$$

ψ is assumed to be $\psi_1 \wedge \psi_2 \ldots \wedge \psi_N$

A sequence $T = j : T'$ is k-shrinkable if :

$\forall\, 0 \le i < j, \forall\, \sigma \in \varphi_j : (\{\sigma\} L_T \{\neg\psi_{i:T}\}) \rightarrow (\exists U \in [T]^k : \{\sigma\} L_U \{\neg\psi_{i:U}\})$

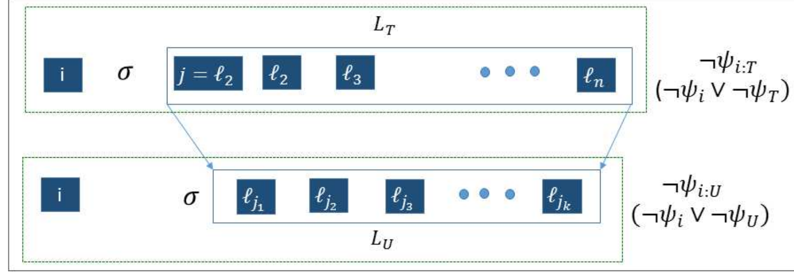


Figure 7.6: Illustration of contrapositive view of revised definition of sequence shrinkability

A contrapositive view of the revised definition is depicted in the Figure 7.6. It reads as: if the execution of L_T with initial state σ results in a violation of its residual property ψ_T or the clause $\psi_{[\,i\,]}$ corresponding to a past iteration i, then there exists a subsequence U of T such that the execution of L_U with the same initial state also results in violation of the residual property ψ_U or the clause $\psi_{[\,i\,]}$. Henceforth, we will consider this definition as the definition of shrinkability of iteration sequences.

As a technical matter, notice that we have included 0 as a value of the past iteration i. Otherwise, any sequence that starts with iteration 1 would have an empty set of past iterations and the condition of k-shrinkability would be vacuously true for the sequence. We, therefore, include 0 as a past-iteration and define $\psi_{[\,0\,]}$ to be *true*. A pleasing consequence of this is when the iteration sequence consists of all the iterations of a loop, the revised definition that includes past-preservation also coincides with the definition of shrinkability of loops (Definition 7.1).

Consider the example `lmin` in Figure 7.3 with the $\mathtt{S} = 5$ and $\mathtt{d} = 1$. Not all sequences of length two are 1-shrinkable by the revised definition. To see this, consider the case of an array `a` as $\{2, 1, 2, 3, 4\}$. Let T be [4,5] and take past iteration i as `1`. Let `m` be 2 in a state σ. Then $\psi_{[\,1\,]}$=$\mathtt{m} \le \mathtt{a[0]} + 1 \equiv \mathtt{m} \le 3$. Clearly, starting from state σ, for the residual loops of size-1 subsequences U, i.e. [4] and [5], the resulting `m` will be 3 and 2 (respectively), and $\psi_{[\,1\,]} \wedge \psi_U$

is satisfied. But starting from the same state σ, the residual loop L_T will produce $\mathtt{m} = 4$, and therefore $\psi_{[1]} \wedge \psi_T$ is not satisfied. On the other hand, it is easy to see that for the same $\mathtt{d}$, all the sequences of size 4 are 3-shrinkable.

To address the issues related to disjunctive properties, the revised definition for sequence shrinkability with respect to existential properties will be as follows:

Definition 7.4 *(Shrinkable iteration sequence, revised for existential properties) Consider a program consisting of a loop L and an existential property ψ to be checked. Let T be an iteration sequence, and let j be the first iteration in T. In addition, let i stand for any iteration before j. The sequence T is k-shrinkable with respect to a property ψ, $0 < k < |T|$, if and only if, starting from every state $\sigma \in \varphi_j$, the residual loop L_T satisfies $\psi_T \vee \psi_{[i]}$ whenever some residual loop L_U of a k-length subsequence U of T satisfies the corresponding property $\psi_U \vee \psi_{[i]}$. In other words:*

$$\forall \sigma \in \varphi_j . \forall 0 \leq i < j . ((\exists U \in \mathcal{P}_k(T) . \{\sigma\} L_U \{\psi_U \vee \psi_{[i]}\}) \implies \{\sigma\} L_T \{\psi_T \vee \psi_{[i]}\}) \qquad (7.4)$$

The revised definition for the case of existential property differs from the one for the case of universal properties, in that now the antecedent needs to hold for only some k-sized subsequence U of T, and not all. On similar lines as in the case of definition with respect to universal property, here also we include 0 as a value of the past iteration i for the same reason. However, for this case, $\psi_{[0]}$ is defined to be *false*. Consequently, when the iteration sequence consists of all the iterations of a loop, this revised definition for existential property implies that the loop is shrinkable as per the Definition 7.1.

7.2.4 From sequence shrinkability to loop shrinkability

We now formally prove the result that we have been working towards: for a loop to be k-shrinkable, it is enough if every iteration-sequence of size $k+1$ is k-shrinkable. Our method of determining the shrink-factor for which a loop is shrinkable will make use of this important result.

Theorem 7.5 *An array processing loop is k-shrinkable with respect to a property ψ, if every iteration-sequence of size $k+1$ is k-shrinkable with respect to ψ.*

Proof To show that the loop is k-shrinkable, it is enough to show that the complete iteration sequence of the loop is k-shrinkable according to Definition 7.3 and 7.4 for universal and existential properties, respectively . However, we shall show a stronger condition that all sequences of size greater than k are k-shrinkable. The proof is by induction on the length n of an iteration sequence T of the loop. For the base case $n = k + 1$, the k-shrinkability of T is a given in the statement of the theorem.

Now, let n be greater than $k + 1$ and assume, as the induction hypothesis, that every sequence of length less than n is k-shrinkable. Let $T = j : T'$. We address the cases of conjunctive and disjunctive properties separately:

Case 1:ψ is conjunctive

As usual, we take a contrapositive view of the shrinkability condition and assume that for some past iteration i of T, starting from a state $\sigma \in \varphi_j$, the property $\psi_{[\,i\,]} \wedge \psi_T$ fails after executing L_T i.e. $\{\sigma\}L_T\{\neg\psi_{[\,i\,]} \vee \neg\psi_T\}$ is true. We show that there exists a k-sized subsequence $U \sqsubset T$ such that $\{\sigma\}L_U\{\neg\psi_{[\,i\,]} \vee \neg\psi_U\}$ is true.

Since $L_T = S_j; L_{T'}$ and $\psi_T = \psi_{[j]} \wedge \psi_{T'}$, we have $\{\sigma\}S_j; L_{T'}\{\neg\psi_{[\,i\,]} \vee \neg\psi_{[j]} \vee \neg\psi_{T'}\}$. Assume that starting with σ, the state reached after executing S_j, the loop body for the iteration j, is σ_1, i.e. $\{\sigma\}S_j\{\sigma_1\}$. We then have $\{\sigma_1\}L_{T'}\{\neg\psi_{[\,i\,]} \vee \neg\psi_{T'}\} \bigvee \{\sigma_1\}L_{T'}\{\neg\psi_{[j]} \vee \neg\psi_{T'}\}$. We show the existence of the desired U by assuming that the first disjunct is true. Since i and j are both past iterations for T', a similar proof works for the case in which only the second disjunct is true. Assume that the first iteration of T' is j'. Since $j + 1 \leq j'$ and $\sigma_1 \in \varphi_{j+1}$, as per the observation made in Section 6.5, $\sigma_1 \in \varphi_{j'}$. Since T' is k-shrinkable, we must have a k-sized subsequence $U' \sqsubset T'$ such that $\{\sigma_1\}L_{U'}\{\neg\psi_{[i]} \vee \neg\psi_{U'}\}$ is true. It follows that $\{\sigma\}S_j; L_{U'}\{\neg\psi_{[i]} \vee \neg\psi_{U'}\}$, and, therefore, $\{\sigma\}S_j; L_{U'}\{\neg\psi_{[i]} \vee \neg\psi_{[j]} \vee \neg\psi_{U'}\}$ are also true. Let T'' be $j : U'$. Obviously, $T'' \sqsubset T$. Since the size of T'' is $k + 1$, T'' is k-shrinkable by the induction hypothesis and thus there exists a k-sized subsequence $U \sqsubset T'' \sqsubset T$ such that $\{\sigma\}L_U\{\neg\psi_{[i]} \vee \neg\psi_U\}$ holds.

Case 2:ψ is disjunctive

We will use direct definition of the shrinkability condition and assume that for some past iteration i of T, starting from a state $\sigma \in \varphi_j$, for some k-sized subsequence $U \sqsubset T$, the property $\psi_{[\,i\,]} \vee \psi_U$ holds after executing L_U i.e. $\{\sigma\}L_U\{\psi_{[\,i\,]} \vee \psi_U\}$ is true. Under this assumption, we show that $\{\sigma\}L_T\{\psi_{[\,i\,]} \vee \psi_T\}$ is true. Assume that starting with σ, the state reached after

executing S_j, the loop body for the iteration j, is σ_1, i.e. $\{\sigma\}S_j\{\sigma_1\}$. There are only two cases: either j is first iteration of U, in that case U can be written as $j : U'$, or U is a subsequence of T'.

Assume $U = j : U'$. Obviously, $U' \sqsubset T'$ and U' is of size $k-1$, and there exists a sequence U'' of size k such that $U' \sqsubset U'' \sqsubset T'$. Since $j : U' \sqsubset j : U''$, by induction hypothesis, $\{\sigma\}L_{j:U''}\{\psi_{[i]} \vee \psi_{j:U''}\}$ is true. Since $L_{j:U''} = S_j; L_{U''}$ and $\psi_{j:U''} = \psi_{[j]} \vee \psi_{U''}$, we can say that $\{\sigma\}S_j; L_{U''}\{\psi_{[i]} \vee \psi_{[j]} \vee \psi_{U''}\}$ is true. Or, $\{\sigma_1\}L_{U''}\{\psi_{[i]} \vee \psi_{U''}\} \bigvee \{\sigma_1\}L_{U''}\{\psi_{[j]} \vee \psi_{U''}\}$ is true. We show the desired proof by assuming that the first disjunct is true. Since i and j are both past iterations for U'', the proof would be similar for the case in which only the second disjunct is true. Assume that the first iteration of T' is j'. As argued in previous case of ψ being conjunctive, it is obvious that $\sigma_1 \in \varphi_{j'}$. Since T' is k-shrinkable, $\{\sigma_1\}L_{T'}\{\psi_{[i]} \vee \psi_{T'}\}$ must be true. This can be extended to $\{\sigma_1\}L_{T'}\{\psi_{[i]} \vee \psi_{[j]} \vee \psi_{T'}\}$. From this, we say $\{\sigma\}L_{j:T'}\{\psi_{[i]} \vee \psi_{j:T'}\}$ or $\{\sigma\}L_T\{\psi_{[i]} \vee \psi_T\}$ is true.

Assume $U \sqsubset T'$. Obviously, $U \sqsubset j : T'$ as well. By induction hypothesis, $\{\sigma\}L_{j:U}\{\psi_{[i]} \vee \psi_{j:U}\}$is true. The rest of the proof is exactly similar to the previous case, with U playing role of U". ∎

7.3 Checking shrinkability of iteration sequences of a size

Recall that according to Theorem 7.5, a loop is k-shrinkable if every iteration sequence of length $k+1$ is k shrinkable. In addition, with our assumption that the loop has a statically computable upper bound of number of iterations, the number of such iteration sequences will be finite. Given a candidate `k`, the procedure `check_loop` in Figure 7.7 non-deterministically chooses an iteration sequence T of length `k+1`, and attempts to verify that T is `k`-shrinkable. This is done in the procedures `check_seq_univ` and `check_seq_exist` for universal and existential properties, respectively, These two procedures encode the criterion for sequence shrinkability, as given by definitions 7.3 and 7.4.

Assume that the given program consists of an array processing loop L of the form `while(`C`)`$\{B\}; Q$ followed by the assertion assert(ψ). Let `X` denote the vector of variables which *may be* modified (by resolving dereferences, if any, using a safe points-to-analysis) in the loop body B. Recall that the implication in the criterion for shrinkability is required to hold for

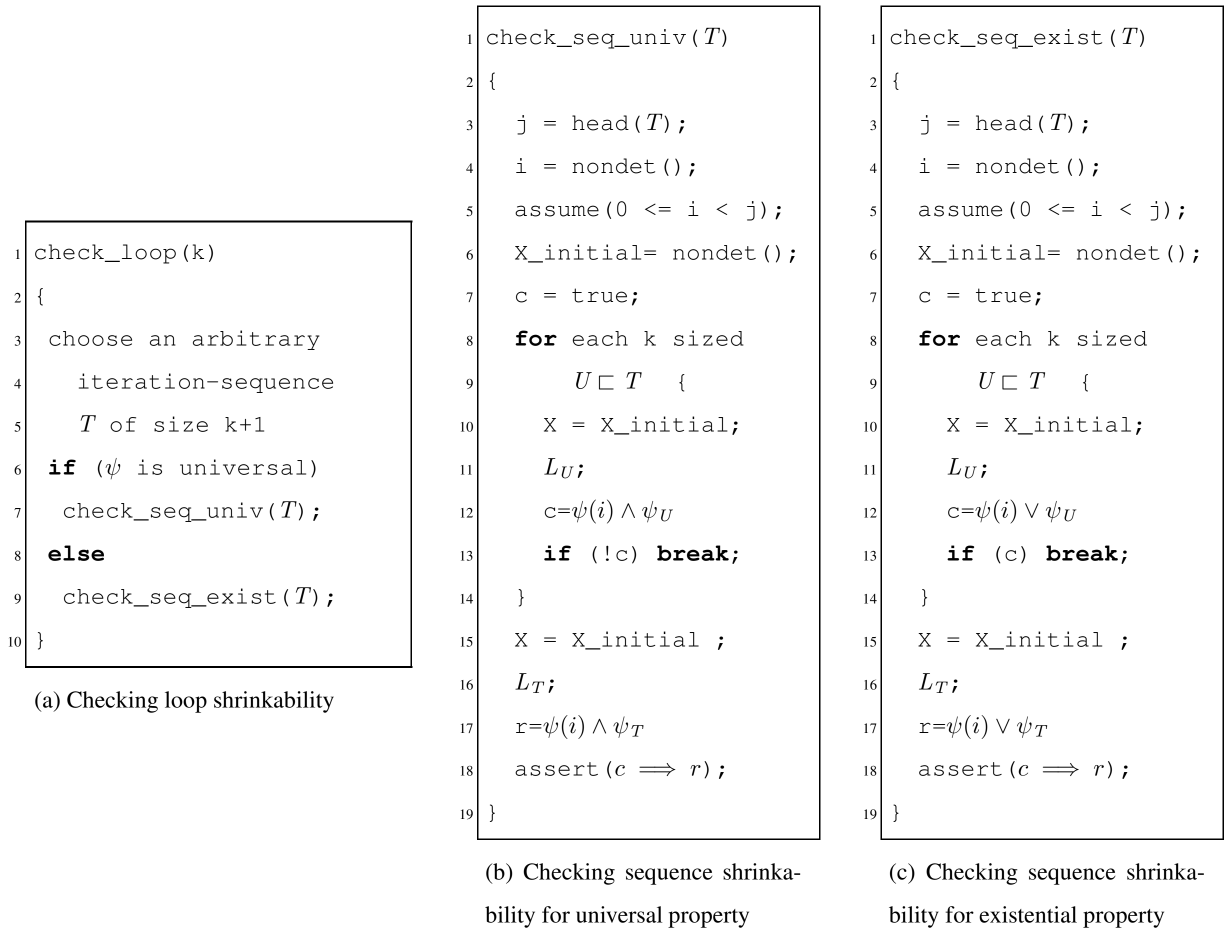


Figure 7.7: Program construction for determining shrinkability. Note that X and X_initial are vectors of variables, and nondet(), accordingly, generates a vector of values.

all states in φ_j, where j is the head of sequence T. The states in φ_j are over-approximated by assigning non-deterministic values to X (through X_initial). Thus, our process of determining shrinkability is conservative, and a future extension to this work would be a static analysis to obtain a better approximation of φ_j.

We discuss the procedure check_seq_univ, given in Figure 7.7(b), which checks the sequence shrinkability for a universal property. The procedure check_seq_exist works on similar lines for existential properties. The loop in lines 8-14 checks the antecedent $(\forall U \in \mathcal{P}_k(T).\{\sigma\}L_U\{\psi_U \wedge \psi_{[i]}\})$ in the implication in the shrinkability condition (Definition 7.3), and stores the result in c. This loop executes a maximum of $k+1$ times, which is the number of subsequences of T of size k. In lines 15-17, the consequent $\{\sigma\}L_T\{\psi_T \wedge \psi_{[i]}\}$ of the same implication is computed, and stored in r. Finally, line 18 checks the condition for shrinkability,

given by the implication c $\Rightarrow$ r. Observe that the residual loop for each subsequence U, and the residual loop for the sequence T, are all evaluated in the same state denoted by the values of the variables in X_initial. It is clear that the program shown in Figure 7.7 can be automatically constructed for any given k, L, and ψ.

The fact that shrinkable loops usually have a low shrink-factor has two consequences for the procedure to determine shrinkability: (i) it allows us to keep the number l till which a program is tested for shrinkability at a low value without the fear of missing out many shrinkable programs, and (ii) since the for loops in lines 8-14 of figures 7.7(b) and 7.7(c) have a bound of $k + 1$, and k is smaller than l, the shrinkability testing procedure is fairly efficient.

7.4 Determining loop shrinkability empirically

We now show how Theorem 7.5 can be used to empirically determine whether a given loop is shrinkable, and to also find the corresponding shrink-factor. To do so, basically we search for a k for which the loop is shrinkable. Since it is an empirical process, we need to limit the search to a predefined limit LIM.

For the given loop L and the property ψ to be checked, we construct a parameterised program check_loop(k) with k as a parameter. Depending on the computing resource availability, we fix the limit LIM to limit our search space. We carry out one of the two processes given in Figure 7.8. While the process search_sf carries out a sequential search starting from k $= 1$ and going up to LIM, the process bsearch_sf does a binary search within 1 to LIM. Each process feeds the parameterised program check_loop(k) for different values of k (the candidate shrink-factor) to a bounded model checker for verification. If the program is verified to be correct for some value of k, then Theorem 7.5 guarantees that the loop in the given program is k-shrinkable. The process stops when it either finds a k for which the loop is shrinkable (success), or when the search space is exhausted (failure).

For both of these processes, it is guaranteed that if they find a k for which loop is shrinkable, then that will be smallest k for which the loop can be found shrinkable as per the Theorem 7.5. When LIM is small, say less than 5, the sequential search search_sf can be used, otherwise the more efficient binary search bsearch_sf is used. We assume that the BMC which we use in the process provides one of the three outcomes: (1) Program is correct, (2)

```
search_sf()
{
 for (k=1; k< LIM; k++)
 {
   Use BMC to check property on
      the program check_prop(k);
   if program is correct then
      Exit with declaration that
          loop is shrinkable
          with shrink factor k;
 }
 Exit with declaration that
     shrikability is unknown;
}
```

(a) Sequential search

```
bsearch_sf()
{
 low = 1; high = LIM;
 checked = {}
 while (high >= low)
 {
   k = (high+low)/2
   if k in checked
      Exit with declaration that
           loop is shrinkable
           with shrink factor k;
   use BMC to check property on
       the program check_prop(k)
   if BMC failed to give a judgement
     high = k-1;
   else if program is correct
     add k to checked
     if (low+1>=high)
        Exit with declaration that
             loop is shrinkable
             with shrink factor k;
     else high = k;
   else low =k+1;
 }
 Exit with declaration that
     loop shrikability is unknown
}
```

(b) Binary search

Figure 7.8: Process to determine loop shrinkability and finding shrink-factor.

Program is incorrect, or (3) Failed due to limit on resources or other reasons. The binary search assumes that if BMC did not fail for some value of k in checking correctness of check_loop, then it will not fail for any lower value of k as well. As we shall see in Section 8.1, the shrink-factor for shrinkable loops are usually small. This is a favourable situation, since programs with a smaller shrink-factors are relatively easier to verify than programs with larger shrink-factors.

However, since our $shrinkability$ finding process is based upon searching for a k within the range $[1, \mathtt{LIM}]$ for which a given program is $shrinkable$ the cases where $shrinkability$

depends upon some parameter appearing in the program can not be addressed by this process. The example of Figure 7.3(b), where the loop is $shrinkable$ for $k = \texttt{d}+2$, is a case in point. For such cases, the process will terminate by saying that $shrinkability$ of the program is unknown.

7.5 Property checking for shrinkable loops

Once we discover that the loop of a program is k-shrinkable, we construct an abstract program that consists of a program fragment to non-deterministically choose a k-sized iteration sequence T, a residual loop L_T, and a residual property ψ_T. The abstract program is submitted to a BMC for verification. The motivating example of Figure 1.3 illustrates the nature of the abstract program, and it is easy to generalize and automate the process of abstraction to arbitrary programs that are within the scope of our method.

Since the quantified property is also encoded as a loop, the residual property can also be constructed as a residual of this loop. Consider a program with a loop L for which the residual has to be constructed with respect to a k-length iteration sequence. Assume that the maximum iteration count of the loop is m. Let $a[e]$ be an arbitrary expression involving an array a of size n. Also assume that the index expression e is accelerable and is of the form $f(i)$, where $i \in [1..m]$ represents a particular loop iteration, and f is the acceleration function. The abstract program non-deterministically chooses a k-length iteration sequence, whose elements are in the range $[1..m]$. The iteration sequence is concretely represented as an array. A loop iterates over all the values of the iteration sequence. The expression $a[e]$ in the loop body is replaced by the corresponding accelerable expression $a[f(i)]$.

To make this clearer, consider the example in Figure 7.9(a). Assume that the size `p` of the array is more than $(\texttt{n}+1)/2$. The loop initializes the array element `a[t]` with the value `2*t`. Assume that the loop is `k`-shrinkable for some property. The maximum iteration count m for the loop is $(\texttt{n}+1)/2$. The code in Figure 7.9(b), written in a C-like notation, is an abstract description of the residual loop. The call to `init` initializes the array `T` with a non-deterministically chosen `k`-length iteration sequence. The C-style comment indicates the constraints on the chosen iteration sequence `T`. The conditions $\texttt{1} \leq \texttt{T[}l\texttt{-1]} < \texttt{T[}l\texttt{]}$ and $\texttt{0} \leq \texttt{2} * \texttt{(T[}l\texttt{]-1)} < \texttt{n}$ together ensure that the iteration sequence consists of increasing values in the range $[1 \ldots m]$, and the condition $\texttt{T[}l\texttt{]-1} < \texttt{p}$ ensures that the chosen values do not cause an out-of-bounds

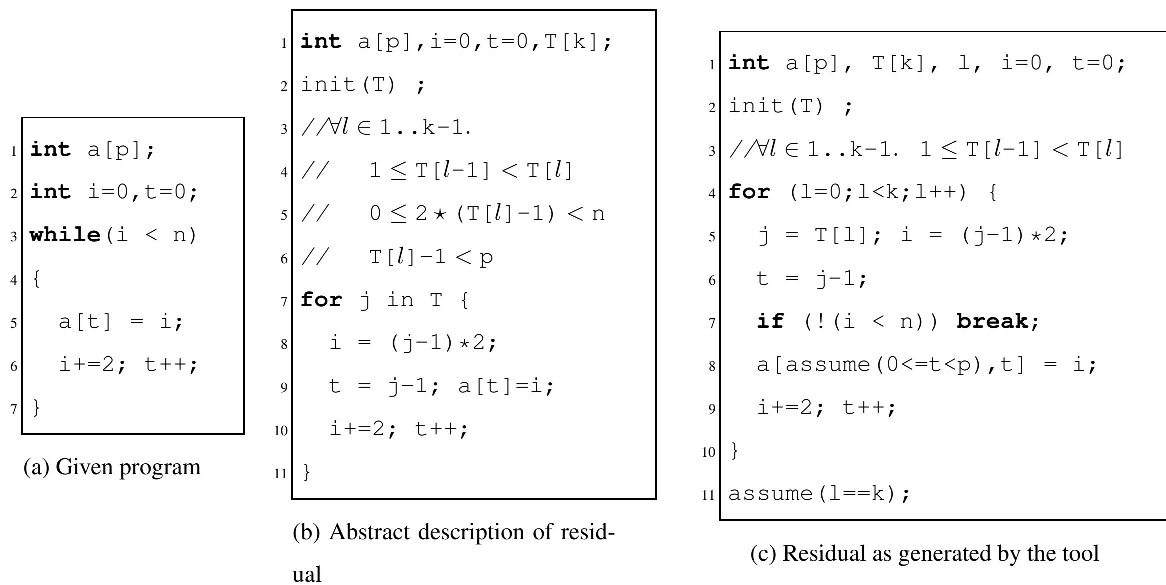


Figure 7.9: Example illustrating the residual of a shrinkable loop. Program in (b) is an abstract description of the residual, presented for ease of explanation.

access of the array. The `for` loop covering lines 7 to 11 iterates over the elements in `T`. Inside the loop body, `i` and `j` are computed through acceleration functions applied to the iteration numbers picked from `T`.

In practice, the constraints on the values in `T` would be enforced programmatically, and this is shown in Figure 7.9(c). Here an increasing sequence of values are chosen, and the constraint that the chosen values are in the range $[1..m]$ is enforced through the conditional `break`. Similarly, the constraint that the index of `a` does not exceed its bound is enforced through the `assume` at line 8. Finally, `assume(l==k)` ensures that the residual indeed iterates `k` times and does not break out of the loop earlier.

7.6 Multiple loops and nested loops

Although the Theorem 7.5 is applicable to a single non-nested array processing loop followed by a property checking code fragment encoded as a non-nested loop, the approach can be used to verify programs having multiple loops or nested loops of certain category.

```
1 sum1=0;
2 for (i=0; i<N; i++)
3   sum1 = sum1+a[i];
4 sum2=0;
5 for (i=0; i<N; i++)
6   sum2 = sum2+a[i];
7 assert(sum1==sum2);
```

(a) Multiple loops

```
1 sum1=0;
2 sum2=0;
3 for (i=0; i<N; i++)
4 {
5   sum1 = sum1+a[i];
6   sum2 = sum2+a[i];
7 }
8 assert(sum1==sum2);
```

(b) Coalesced loop

Figure 7.10: Illustration of multiple loops that can be coalesced

7.6.1 Multiple loops

Our method can be used when the program consists of a cascaded series of simple loops that can be coalesced into one simple loop. To elaborate, let the program be $\{Q_1; R_1; Q_2; R_2; Q_3; R_3\}$, where the Q_is are loop-free statements and the R_is are simple loops of the form `while` (C_i) $\{B_i\}$. Our method can handle such a program if it can be transformed to a semantically equivalent program $Q;$ `while` (C) $\{B_1; B_2; B_3\}$ for some loop-free statements, Q, and a loop condition C.

To illustrate, consider the code snippet in Figure 7.10. The original code processes array `a` by a cascading of two simple loops. We transform the code snippet by coalescing these two loops into one loop as illustrated. Now the method can be used to verify the property on transformed code. The basic criterion to enable coalescing of multiple such loops is that processing in a latter loop should not depend upon values computed by a former loop. Even this simple strategy enabled us to verify 50 of the 81 programs with non-nested multiple loops in the SV-COMP 2017 benchmark suite.

7.6.2 Nested loops

It is theoretically possible to flatten a nested loop to a single non-nested loop. Therefore, any program with a nested loop can be transformed to a flattened loop and then our method can be applied. However, as mentioned earlier, the precision of our method depends upon identifying accelerable variables and their corresponding accelerated expression. Finding accelerable

```
1  for (i=0; i< N; i++)
2   for (j=0;j<N; j++)
3   {
4    t1=a[i][j]+b[i][j];
5    t2=a[i][j]-b[i][j];
6    c[i][j] = t1;
7    d[i][j] = t2;
8   }
9  for (i=0; i< N; i++)
10  for (j=0; j< N; j++)
11  {
12   t1=c[i][j]+d[i][j];
13   assert
14     (t1==2*a[i][j]);
15  }
```

(a) Nested loop

```
1  for (i=0,j=0; i< N; )
2  {
3   t1=a[i][j]+b[i][j];
4   t2=a[i][j]-b[i][j];
5   c[i][j] = t1;
6   d[i][j] = t2;
7   j++;
8   if (j==N)
9     { j=0; i++;}
10 }
11 for (i=0,j=0; i< N; )
12 {
13  t1=c[i][j]+d[i][j];
14  assert
15    (t1==2*a[i][j]);
16  j++;
17  if (j==N)
18    { j=0; i++;}
19 }
```

(b) Flattened loop

```
1  int T[k],ij;
2  init(T);
3  for (l=0; l< k; l++)
4  {
5    ij = T[l];
6    i = ij/N; j = ij%N;
7    if (!(ij < N*N)) break;
8    t1 = a[i][j]+b[i][j];
9    t2 = a[i][j]-b[i][j];
10   c[i][j] = t1;
11   d[i][j] = t2;
12 }
13 for (l=0; l< k; l++)
14 {
15   ij = T[l];
16   i = ij/N; j = ij%N;
17   if (!(ij < N*N)) break;
18   t1 = c[i][j]+d[i][j];
19   assert (t1==2*a[i][j]);
20 }
```

(c) Residual loop with acceleration

Figure 7.11: Illustration of handling nested loops

variables in a loop obtained after flattening a nested loop is much more challenging. In certain kind of nested loops, the accelerable variables can be identified using some template based methods. We illustrate one such example in Figure 7.11. The code snippet of Figure 7.11(a) computes the sum and difference of two matrices `a` and `b`, as matrices `c` and `d`, respectively. Finally, it checks that `c+d=2a`. The code snippet given in Figure 7.11(b) results after flattening of the nested loops in the original code snippet. If we can identify that `i` and `j` are accelerable in the resulting code snippet, as shown in Figure 7.11(c), then we can generate residual loop and residual property which are good enough to verify the property in the given original code snippet.

Chapter 8

Implementation and measurements

The proposed abstraction has been implemented in a tool called *VeriAbs* [19]. Within the scope of our method, i.e. programs with a single loop followed by the property to be checked, the tool supports most C constructs including pointers, structure, arrays, heaps, and non-recursive function calls. However, our implementation does not support nested loops, array accesses through pointer arithmetic, and recursive functions.

8.1 Implementation

We have re-used implementation of LABMC [29] to discover accelerable variables, and we use CBMC 5.8 as the bounded model checker to determine $shrinkability$ of the loop and to check the residual property on the abstracted program. For the static analysis required to produce the abstract program for shrinkability checking and property checking, we use a static analyzer generator PRISM developed at TRDDC, Pune [53, 20]. In case the subject program has multiple loops, we check whether they can be coalesced into one loop. For this, we make use of use-def chains constructed using *reaching definition* [1] analysis to find if processing inside a loop body is dependent on the values computed in some earlier loop. We observe that if a non-accelerable variable's use inside the loop body is dependent on definitions from within the loop body only then that variable can not have loop carried dependency. Using this fact, we make use of the use-def analysis to check if there can not be variables (other than the accelerable variables) having loop-carried dependence in the program. We use the use-def chains implementation available in PRISM, the static analyzer generator mentioned above.

As stated earlier, we have reused the implementation from LABMC [29] to discover accelerable variables and identify their corresponding acceleration expression. However, implementation in LABMC has limited capability for the purpose: it is difficult for LABMC to discover accelerable variables and corresponding acceleration expression from a flattened loop of a nested loop. Therefore, our current implementation does not support nested loops. But if a more sophisticated tool to discover accelerable variables is available, then it can be easily plugged into our implementation to address the nested loops.

Our observation is that majority of shrinkable loops have a very low shrink-factor, typically 1 or 2. Therefore, we use a `LIM` value of 5 while checking if a loop in a given program is shrinkable. If a loop is not found to be shrinkable within a candidate shrink-factor of 5, we report the shrinkability of the loop to be unknown. We used the sequential search algorithm to find shrinkability of the loop. Given a program with a shrinkable loop, if the verification of the corresponding abstract program succeeds on the residual property, the tool declares the original program to be correct with respect to the given property. On the other hand, if the verification of the abstract program, for a universal property, fails, and the array processing loop in the program has no loop-carried dependence for non-accelerable variables, the original program is declared to be incorrect. Otherwise, the tool indicates its inability to decide on the correctness of the program.

8.2 Experiments

An early version of the tool *VeriAbs* competed in the SV-COMP 2017 verification competition [8], where it was ranked third amongst the 17 participating tools in the *ArraysReach* category. In the competition, a time limit of 900 seconds was provided for each program verification task. In the version that participated in the competition, our implementation was not efficient. In particular, in generated abstract programs, the encoding of selecting k iterations non-deterministically was inefficient. As a result the tool timed out on some of the programs.

Subsequently, we modified the implementation to generate the abstract program with efficient encoding of selecting k iterations non-deterministically. We re-run the resulting version on the same benchmark (SV-COMP 2017). We ran the experiment on a machine with two i7-4600U cores @2.70 GHz and 8 GB RAM. We kept the same time limit of 900 seconds per

program verification task. The category *ArraysReach* consisted of 135 programs, of which 95 were correct and the remaining 40 were incorrect with respect to their properties. We have categorized these programs in the Table 8.1(a). Out of a total of 135 programs, 42 were beyond the scope of *VeriAbs* because they either contained nested loops (12 programs), or contained multiple loops which were not collapsible (30 programs). In Table 8.1(a), the first column labeled 'Programs' shows the category of programs, the second column labeled 'True' contains the number of correct programs in the category, and the third column labeled 'False' shows the number of incorrect programs in that category. The last column labeled 'Total' provides the total number of programs in the category. Out of the remaining 93 programs, VeriAbs found 89 programs to be 1-shrinkable, and 2 to be 2-shrinkable. While VeriAbs could not decide the shrinkability of the remaining 2 programs, we found those to be non-shrinkable on manual inspection. Note that, for these two programs, VeriAbs could successfully try out all values of k from 1 to `LIM`, which was 5, in checking of shrinkability of the programs within the time limit of 900 seconds allotted for the verification task.

Table 8.1(b) shows the verification results for the 91 shrinkable programs. All correct programs except one were verified successfully. Moreover, none of the 26 incorrect programs were declared to be correct, demonstrating the soundness of our tool. For all these 26 programs, the underlying BMC found the abstract programs to be incorrect. However, out of the 26 incorrect programs, 23 programs had no loop carried dependency and thus the tool could rightly declare these as being incorrect, as discussed in Section 7.1. Since remaining 3 programs had loop carried dependency, as per the same discussion, VeriAbs could not say if the program is correct or incorrect, so it remained undecided. The timing data shows the average time taken in verifying each program. As expected, bulk of the time is taken in determining shrinkability as the BMC has to verify $O(k^2)$ residual programs to determine that the shrink-factor is k, whereas property checking of the abstract program involves a loop with just k iterations. Given the limits of the configuration in terms of processing power and available RAM of the machine used for the experiments, the timings are reasonable. The tool CBMC, which we use as base verifier in VeriAbs, also had participated in the SV-COMP 2017 competition. However, from the *ArraysReach* category, CBMC could verify only 18 programs. It shows that our technique does have a very significant value add over the base tool CBMC.

An interesting property of *Veriabs* is that while it is limited by its ability to deal only with

Table 8.1: Experimental results for SV-COMP 2017 ArraysReach benchmarks

(a) Programs categories

Programs	True	False	Total
With nested loops	5	7	12
With non-collapsible multiple loops	24	6	30
Shrinkable	65	26	91
Shrinkability unknown	1	1	2
Total	95	40	135

(b) Property verification results

Results on shrinkable programs	#Cases	Average time per program (in seconds)	
		Checking shrinkability	Total
Property declared correct	64	30.60	39.68
Property declared incorrect	23	10.72	19.73
Unable to decide	4	227.26	236.06
Total	91	34.22	43.27

shrinkable loops, once a loop is discovered to be shrinkable, the method is impervious to either the existence or the size of loop bounds—increasing the loop bound does not cause an otherwise verifiable program to timeout. Comparison with the two tools that fared better than *VeriAbs* in the competition, namely *Ceagle* [70] and *Smack* [17], reveals some interesting information. We selected four correct programs, one from each of the following categories of the test suite: array copy, array initialisation, two index copying and finding minimum. We increased the array size considerably (from 100000 to 10000000). While both the tools had succeeded on the programs with the original array sizes, after this increase in array sizes *Smack* timed out, and *Ceagle* either crashed or declared the programs to be incorrect. We surmise that the two tools are based on bounded model checking without any abstraction. In this respect, our tool performs better than these two tools that were placed ahead of ours in the competition.

The tool participated in the 2018 edition of SV-COMP verification competition [9] as well. In this edition of the competition, the time limit per program verification task was again kept at 900 seconds. There were 13 tools competing in the *ArraysReach* category of the competition. Out of a total of 167 programs in the category, 117 programs were found having shrinkable loops and among these 86 were correct programs. Our tool could verify 81 of these correct programs. For the remaining five programs for which the BMC found the corresponding abstract programs to be incorrect, the tool VeriAbs remained undecided. The tool was ranked first among the competing tools in this category. It further reinforces that the shrinkability approach is indeed effective on verifying property on programs processing large arrays through loops. Note that the data presented in the table is for SV-COMP 2017 and version of VeriAbs used in that had the implementation of shrinkability method only.

Chapter 9

Loop pruning

Consider the program of Figure 9.1, a reproduction of the motivating example of Figure 1.4. In Section 1.3.3, we illustrated how it was enough to check the asserted property on a modified program consisting of only one iteration. The modified program, in turn, accessed and processed only the first two elements of the array. This illustrates that there are array processing programs in which a loop can be pruned so that array computations in the pruned loop are restricted to only an initial segment of it. Such programs satisfy the condition that if the asserted property holds for the modified program, it will hold for the original program too. In this chapter, we shall investigate into the kinds of loops that exhibit such behaviour, and also identify the length of the initial array segment that will have to be considered for this purpose.

9.1 Basic idea

Assume that in an execution of the program of Figure 9.1, `min1` is assigned for the last time when the loop counter `i` has value i_1. Observe that the value assigned to `min1` is the *initial value* of $\texttt{a}[i_1]$, and the assignment is under the condition that $\texttt{min1} > \texttt{a}[i_1]$. Suppose that prior to this assignment `min1` was assigned only once when the loop counter `i` had value i_2, and `min1` had its initial value $\texttt{a}[0]$. Obviously, the conditions $\texttt{a}[0] > \texttt{a}[i_2]$ and $\texttt{a}[i_2] > \texttt{a}[i_1]$ would both be true, implying the condition $\texttt{a}[0] > \texttt{a}[i_1]$. In other words, the condition $\texttt{min1} > \texttt{a}[i_1]$ will be true even when we replace `min1` with its *initial value* `a`[0]. Therefore, if we prune the loop to only one iteration, then in an execution of the pruned loop, where value of $\texttt{a}[1]$ is same as that of $\texttt{a}[i_1]$ in the execution of original loop, we would get the same value of `min1` at the

```c
#define N 100000
int main() {
 int a[N], i, min1, min2;
 min1 =a[0];
 for (i=1; i< N; i++)
  if (min1 > a[i]) min1 = a[i];
 min2 = a[0];
 for (i=1; i< N; i++) a[i-1] = a[i];
 for (i=0; i< N-1; i++)
  if (min2 > a[i]) min2 = a[i];
 assert (min1==min2);
}
```

(a) Concrete program

```c
#define N 100000
int main() {
 int a[N], i, min1, min2;
 min1 =a[0];
 for (i=1; i< 2; i++)
   if (min1 > a[i]) min1 = a[i];
 min2 = a[0];
 for (i=1; i< 2; i++) a[i-1] = a[i];
 for (i=0; i< 1; i++)
   if (min2 > a[i]) min2 = a[i];
 assert (min1==min2);
}
```

(b) Pruned program

Figure 9.1: Loop pruning abstraction illustration

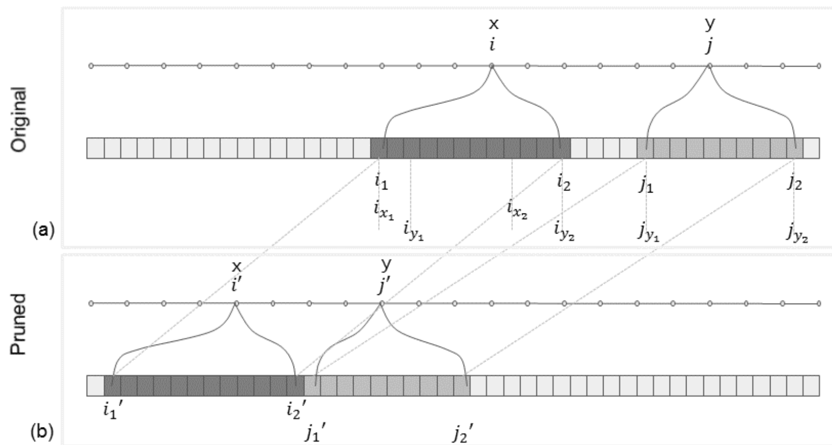


Figure 9.2: Illustration of loop pruning approach

end. This observation will hold no matter how many times `min1` was assigned prior to its last assignment. Observe that while the value of `min1` is not the same as in the original loop, the condition `min1 > a[i]` evaluates to the same value.

As a generalization of this idea, consider two variables of interest (say x and y) are being computed in a loop. Assume that the initial values of the array elements are assigned non-deterministically. This essentially represents a situation in which no assumption can be made

about the initial values of the array elements. For instance, the array could be a parameter to a function being analysed. Also assume that the indices in the array references increase monotonically over successive iterations. In a particular execution, let x and y be assigned for the last time in iterations i and j respectively. Consider the assignments happening in iteration i for variables x and y. The RHS of assignments, and conditions controlling these assignments, if any, will depend upon the initial values of certain scalars and, more importantly, on certain array elements, say those whose indices are in a range $[i_1, i_2]$. For the purpose of illustration, assume that there is a single array being accessed in the loop. We have shown the scenario for a run of original program in Figure 9.2(a). In the diagram, the strip with cells represents the array accessed in the loop. Let us assume that the assignment to x in the iteration i requires references to `a[8]` and `a[10]`. Further, assume that while `a[8]` was unchanged from the initial state, `a[10]` was computed in an earlier iteration in terms of the initial value of, say, `a[5]`. Then the range of indices for the computation of x in the iteration i is $[5, 8]$, and is shown in the figure as $[i_{x_1}, i_{x_2}]$. Similarly, assume y may also have been computed in i (though not for the last time), and the range of indices for this is $[6, 9]$, shown as $[i_{y_1}, i_{y_2}]$. Then the range of array indices $[i_1, i_2]$, whose initial values were used to compute the two variables of interest x and y in iteration i, is $[5, 9]$. Likewise, assume that the range of array indices used in iteration j is $[j_1, j_2]$.

Suppose we have a pruned loop that has an execution as shown in Figure 9.2(b). The computations in the original program's execution in the iterations i and j take place in the iterations i' and j', respectively, in this execution. In addition, though the distance between i' and j' has been reduced[1], the corresponding new ranges $[i'_1, i'_2]$ and $[j'_1, j'_2]$ are still disjoint. A non-deterministic choice of the initial values of the array in these ranges can ensure that the computations at i and j are replayed at i' and j'. We have to additionally ensure that the value of x assigned in the i'th iteration remains the last value assigned. For that, the conditions controlling the assignment to x must remain false after the loop counter value i', as it was the case after the loop counter value i in the execution of the original loop. It may be noted that while the above description of the method is based on a dynamic scenario, i.e. a trace, the actual method is based on static analysis of the program.

[1]Reducing the distance results in a pruned loop with fewer iterations and increases the possibility of successful verification.

$$
\begin{array}{rcl}
\mathsf{PB} & ::= & \mathsf{St} \\
\mathsf{St} & ::= & \mathsf{St}\ ;\ \mathsf{St} \mid \mathsf{AtomSt} \mid \mathbf{if}(\mathsf{BoolE})\ \mathbf{then}\ \mathsf{St} \mid \\
 & & \mathbf{for}\ (\ell := \mathsf{c};\ \ell < \mathsf{c};\ \ell := \ell\text{+z})\ \{\mathsf{LSt}\} \mid \\
 & & \mathbf{for}\ (\ell := \mathsf{c};\ \ell > \mathsf{c};\ \ell := \ell\text{-z})\ \{\mathsf{LSt}\} \\
\mathsf{LSt} & ::= & \mathsf{LSt}\ ;\ \mathsf{LSt} \mid \mathsf{LAtomSt} \mid \mathbf{if}(\mathsf{LBoolE})\ \mathbf{then}\ \mathsf{LSt} \\
\mathsf{AtomSt} & ::= & v := \mathsf{E} \mid a[\mathrm{u}] := \mathsf{E} \mid \mathbf{assert}(\mathsf{BoolE}) \\
\mathsf{LAtomSt} & ::= & v := \mathsf{LE} \mid a[\ell\text{+c}] := \mathsf{LE} \mid \mathbf{assert}(\mathsf{LBoolE}) \\
\mathsf{LE} & ::= & \mathsf{LE}\ \mathrm{op}\ \mathsf{LE} \mid a[\ell\text{+c}] \mid v \mid \mathsf{c} \\
\mathsf{E} & ::= & \mathsf{E}\ \mathrm{op}\ \mathsf{E} \mid a[\mathrm{u}] \mid v \mid \mathsf{c} \\
\mathsf{BoolE} & ::= & \mathsf{BoolE}\ \mathrm{logop}\ \mathsf{BoolE} \mid \mathsf{E}\ \mathrm{relop}\ \mathsf{E} \mid \mathrm{NOT}\ \mathsf{BoolE} \\
\mathsf{LBoolE} & ::= & \mathsf{LBoolE}\ \mathrm{logop}\ \mathsf{LBoolE} \mid \mathsf{LE}\ \mathrm{relop}\ \mathsf{LE} \mid \mathrm{NOT}\ \mathsf{LBoolE}
\end{array}
$$

Figure 9.3: Grammar to describe the programs of interest

The method works for only a limited class of programs. The syntactic and extra-syntactic restrictions to specify this class of programs, are specified in Sections 9.2 and 9.3, respectively. The bounds on the number of iterations in the pruned loop depends on two factors: (1) the range of array indices required to compute the last assignment of variables of interest, and (2) the number of iterations required to compute these variables. These are described in Sections 9.4 and 9.5. Finally, the construction of the pruned program is described in Section 9.5.7.

9.2 Programs of interest

Our focus is on programs that process large arrays, through loops that iterate for a comparable number of iterations, and check universally quantified properties over such arrays. Our idea is to reproduce the values of the variables computed in a loop by a smaller loop, called *pruned loop*, having only a few iterations (let us denote it by K), which may be determined statically. The reproducibility of value of variables through smaller loops will ensure that if the property is violated in the original program then it will also get violated in a transformed program having pruned loops. The basic difference in this approach and loop shrinking is that instead of picking a small set of iterations chosen non-deterministically, here we pick the first K iterations of the

loop. Moreover, the value of K is determined analytically rather than empirically. We observe the following characteristics of the program in our motivating example that made this approach applicable.

1. Inside a loop body, arrays are indexed using a constant offset from the loop iterating variable e.g. `a[i-1]`, and outside a loop body, arrays are indexed with only constants.
2. The loop counter variable is used only for indexing the array elements.
3. There are no nested loops in the program.

9.2.1 Syntactic constraints

We restrict ourselves to programs that are generated by the grammar given in Figure 9.3. In the grammar, c refers to integer constants, u refers to non-negative integer constants, z refers to positive integer constants, v denotes scalar variables, ℓ denotes loop counter variables and a denotes array variables. Note that, the set of scalar variables v is disjoint from the set of loop counter variables ℓ. The grammar captures all the restrictions observed earlier for the motivating example. Following is a summary of restrictions on programs of interest, imposed through the given grammar.

1. Every loop in a program is a **for** loop which initialises a loop counter variable ℓ, with a value, c_{init}, checks it against an upper bound, c_{ub}, and increments or decrements it by a constant t (called step), in every iteration.
2. The **if** statements have no *else* part.
3. Within a loop body, arrays are indexed using a constant offset from the loop counter variable e.g. $\ell, \ell + 2, \ell - 1$, and outside a loop body, arrays are indexed with constants only.
4. The loop counter variable is used only for indexing the array elements, and it is not modified within a loop body.
5. There are no nested loops in the program.

Observe that all loops are terminating, and therefore, the programs under consideration will always terminate. In addition, the execution of such programs is deterministic for a given initial state. We assume that there is no out of bound access of arrays. Following are some additional syntactic constraints, which are not expressed through the grammar.

1. Each loop has its distinct loop counter—the same loop counter variable is not used in more than one loop.
2. There will be only one kind of **for** loop in the program. Either all of them increment their loop counter variable or all of them decrement it. However, the increment/decrement step can be different for different loops.

We assume that, in our programs, all **for** loops increment their loop counter variable. The subsequent discussion can easily be adapted to address programs having **for** loops that decrement their loop counter variable (or, lcv, in short). The lcv in the first and last iterations of a loop will be represented as c_{init} and c_{final} (at most equal to c_{ub}-1) respectively, and it is obvious that these can be computed statically. Since the iterations of a loop and its lcvs will have a one to one relationship[2], henceforth, we will refer to an iteration by its lcv. Every array a within a loop body, is accessed as $a[\ell + c]$, where c is a constant offset (either positive or negative). For example, an array reference $a[\ell + 2]$ refers to the element $a[5]$ in an iteration where ℓ is 3. For notational convenience, we shall often elide the loop counter ℓ from an array reference $a[\ell + c]$, and denote it as a_c. We shall call this a *relative array reference*.

9.2.2 Variables, program state and traces

We consider individual elements of the arrays as separate variables. Let $\mathbb{V}$ be set of all variables in the program. For example, in the program of Figure 9.1, the set $\mathbb{V}$ will comprise of 100000 variables for the elements of the array `a`, the loop counter variable `i` and the scalar variables `min1` and `min2`.

Let the map $\sigma : \mathbb{V} \rightarrow Val$, where Val is an appropriately defined set of values, represent a program state. A *trace* is a sequence of *execution states* ω_1, ω_2, ..., ω_m, where each ω_j is a pair $((l, i), \sigma)$, where l is a program location, i is lcv if l belongs to a loop otherwise it is taken as $\perp$, and σ is the program state at the program location l when lcv is i. We call the pair (l, i) a *trace-point*. In the context of a particular trace τ, we denote a trace-point as $(l, i)_\tau$ and an execution state as $((l, i), \sigma)_\tau$. Given an execution state ω having corresponding program state as σ, and a variable x, we will use both $\omega(x)$ and $\sigma(x)$ to denote the value of x in the execution state. In the subsequent discussion, in the context of a trace, an array reference $a[\ell + c]$ at a trace

[2] i_{th} iteration $(i > 0)$ will have the lcv $c_{init} + (i - 1) * t$, where t is the loop step

point (l, i) will be seen as the variable reference $a[n]$ where $n = i + c$.

9.2.3 Objective

Consider a program P of interest that has a loop having an $\mathbf{assert}$ statement in its body (called a *property checking loop*). For simplicity, we assume that there is only one such property checking loop in the program. Let l_{a} be the location of the $\mathbf{assert}$ statement. Our goal is to construct a pruned program P' in which the counter of each loop reaches only up to a pre-computed loop-specific bound during execution. This bound, in general, is less than the original bound of the loop. Let $\mathcal{V}$ be the set of scalar variables used in the program. We would eventually prove the following claim for the pruned program P'.

Claim 9.1 *Let τ be a trace of the original program having a trace-point $(l_{\mathsf{a}}, i_{\mathsf{a}})$, and let X be the variables used in the* $\mathbf{assert}$ *expression at the trace-point $(l_{\mathsf{a}}, i_{\mathsf{a}})$. Then there exists a trace τ^p of the pruned program P', having a trace-point $(l_{\mathsf{a}}, {i_{\mathsf{a}}}')$, such that the execution states $\omega = ((l_{\mathsf{a}}, i_{\mathsf{a}}), \sigma)_{\tau}$ and $\omega' = ((l_{\mathsf{a}}, {i_{\mathsf{a}}}'), \sigma^p)_{\tau^p}$ satisfy the following:.*

1. *$\forall x \in X \cap \mathcal{V}. \sigma(x) = \sigma^p(x)$ i.e. scalars used in assert expression have the same value at the* $\mathbf{assert}$ *location during corresponding iterations, and*
2. *$\forall a[n] \in X. \sigma(a[n]) = \sigma^p(a[n - i_{\mathsf{a}} + {i_{\mathsf{a}}}'])$ i.e. same array references (but possibly mapping to different array variables $a[n]$ and $a[n - i_{\mathsf{a}} + {i_{\mathsf{a}}}']$) have the same value.*

The claim has a direct bearing on the property checking of the program. Suppose the property is violated in the original program, exhibited by failure of assertion at a trace-point $(l_{\mathsf{a}}, i_{\mathsf{a}})$ in a trace τ. Let $\omega = ((l_{\mathsf{a}}, i_{\mathsf{a}}), \sigma)_{\tau}$ and $\omega' = ((l_{\mathsf{a}}, {i_{\mathsf{a}}}'), \sigma^p)_{\tau^p}$, where, τ^p is a trace of the pruned program, and ${i_{\mathsf{a}}}'$ is the lcv that is a witness to the claim. Without loss of generality, the assert expression can be expressed as a function $f(\sigma(x), \sigma(a[i_{\mathsf{a}} + c]))$ and $f(\sigma^p(x), \sigma^p(a[{i_{\mathsf{a}}}' + c]))$ at the trace-points $(l_{\mathsf{a}}, i_{\mathsf{a}})_{\tau}$ and $(l_{\mathsf{a}}, {i_{\mathsf{a}}}')_{\tau}$ respectively, and if the claim holds, both functions would evaluate to the same value. Therefore, the failure of assertion at the trace point $(l_{\mathsf{a}}, i_{\mathsf{a}})$ in τ will imply failure of the assertion at the trace point $(l_{\mathsf{a}}, {i_{\mathsf{a}}}')$ in τ^p. Alternately, taking a contrapositive view, we can say that the verification of the pruned program as safe implies that the original program is safe too.

9.3 Loop dependence graph and semantic constraints

We introduce a variant of the Program Dependence Graph (PDG) [36] that will help us: (1) justify and formalise the semantic constraints needed for Claim 9.1 to hold, (2) compute the bound of a pruned loop, and (3) prove Claim 9.1. However to avoid introducing new notation, we shall continue to call this variant also as a PDG.

9.3.1 Loop dependence graph

To formalise the semantic constraints, and to develop the solution of our approach, we shall consider every loop as a program in its own right. Variables of this program are the scalar variables used in the loop body, and a variable a_c for each distinct c and array a, for every access $a[\ell + c]$ appearing in the loop body. In particular, the loop counter variable ℓ is not taken as a variable of this program. We assume that *live variable* and *reaching definition* [1] analyses, and *use-def chains* [52] computation is available for each loop, as well as for the given program. Further, we assume that the reaching definition analysis and use-def chains computation for the loop is performed with the following additional semantics. (1) For every variable that is live at $ENTRY$ of the loop, there is a definition assumed to be originating at the $ENTRY$ of the loop. (2) At a use point of a variable a_c, the value assigned by a definition of $a_{c'}$ appearing in the loop body is considered to be used only if: (a) the definition reaches the loop head as well as the use point, and (b) $c' > c$ and $c' - c$ is a multiple of the loop step. This is so because the value to some $a[n]$ can be given in some past iteration through assignment to $a_{c'}$ and $a[n]$ gets accessed as a_c in some latter iteration. For example, in the code snippet of Figure 9.4, the use of $\texttt{d}_{-2}$ at line 11 depends upon the definition of $\texttt{d}_{0}$ at line 12.

We construct the PDG, for each loop, with a node set consisting of: (1) *definition nodes* for each assignment statement and each definition originating at the $ENTRY$ of the loop, and (2) *condition nodes* for every condition expression. In addition, we explicate the dependences of the condition expressions and RHS value of the assignments on the variables used in it and call the nodes representing these variables as *use nodes*. We label the definition node and the use node of a variable v, for a definition and use of v respectively, at a location l by (l, v). We label the condition node for a condition expression by its location l. In Figure 9.4, we illustrate PDG of the loops for the given code snippet. We use line number as program location

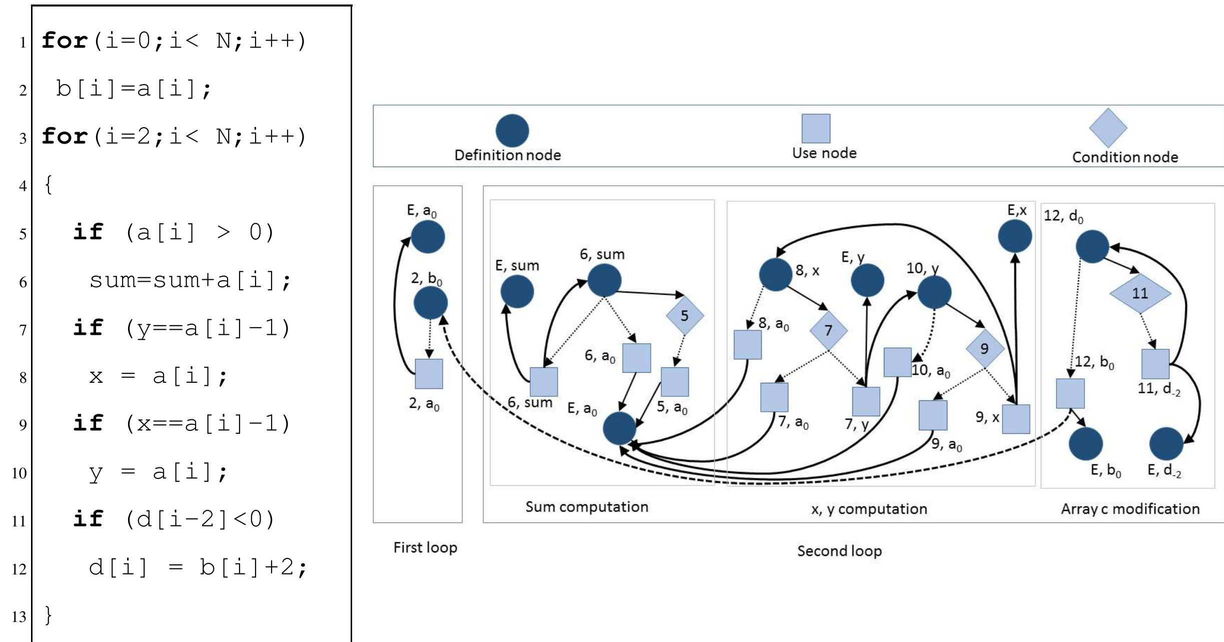


Figure 9.4: Illustration of loop dependence graph and cyclic dependence

to label the nodes[3]. For the nodes representing definitions originating at the $ENTRY$ of the loop, we use letter E as location to label the nodes. The edges in the graph represent usual data and control dependence. An edge from node n_1 to n_2 means n_1 is dependent (data or control) on n_2. The *definition node* for an assignment statement is dependent on the *condition nodes* for the conditions that control the assignment directly or transitively, and the definition node also depends upon the variables used in RHS of the assignment statement. The conditions of condition nodes on which a definition node is dependent are called controlling conditions of the assignment of the definition node. The condition node for a condition expression depends on the variables used in the condition expression. Use of a variable at a use point depends on the definitions of the variable occurring in the use-def chain of the use of the variable at the use point.

The use of a variable a_c in a loop may get its value from the definition of a variable $a_{c'}$ reaching from another loop (identified using use-def chains computed for the entire program). For example, in Figure 9.4, the use node $n_1 \equiv (12, \mathrm{b}_0)$ in the PDG of second loop depends upon the definition node $n_2 \equiv (2, \mathrm{b}_0)$ in the PDG of first loop. To represent this dependence, we put an edge from n_1 to n_2 (dotted edge shown in the diagram) and call it *inter-loop depen-*

[3]We assume that at each line there is at most one statement (assignment, condition or **assert**)

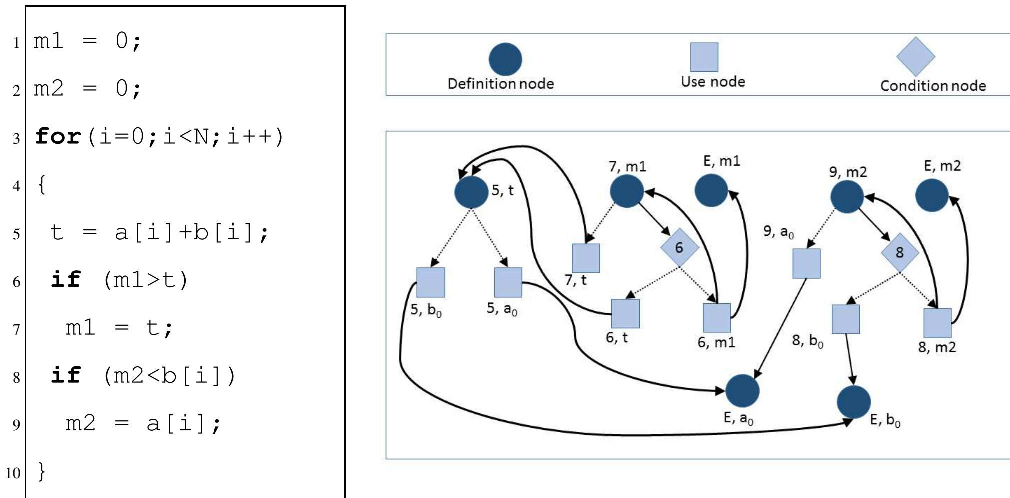


Figure 9.5: Illustration of self control dependence

dence edge.

9.3.2 Constraints on conditions and chain of dependence

When the definition node of a variable, say (l_1, x), has transitive dependence[4] on the definition nodes (l_2, y) and (E, y) of a variable y, then x is said to have *loop-carried dependence*. For the code snippet of Figure 9.4, there is a cycle in the PDG of the second loop, involving definition node and use node of `sum`, implying that the value of `sum` in one iteration data-depends on the value of `sum` computed in some earlier iteration. Similarly, there is a cycle with nodes corresponding to the variables `x` and `y`. The unfolding of these cycles will exhibit a chain of dependence, bounded by only the size of the loop. Therefore, the values of variables `sum`, `x` and `y` can not be reproduced with any reduced bound of the loop. A similar cycle is there involving definition node (11,$\mathtt{d}_{0}$) that is controlled by the condition (10), which makes use of (10,$\mathtt{d}_{-2}$). Therefore, the value of a given element of array `d` can not be reproduced with a reduced bound of the loop. So, we put the constraint that there should be no such cycles in the PDG of the loops.

However, in some specific scenarios, in spite of cycles appearing in the PDG of a loop, we can get a reduced bound for the loop. For example, in the code snippet of Figure 9.5, there is a cycle in the PDG involving the definition node (7, `m1`) and the condition node (6) which uses

[4]There is a path from (l_1, x) to (l_2, y) and (E, y).

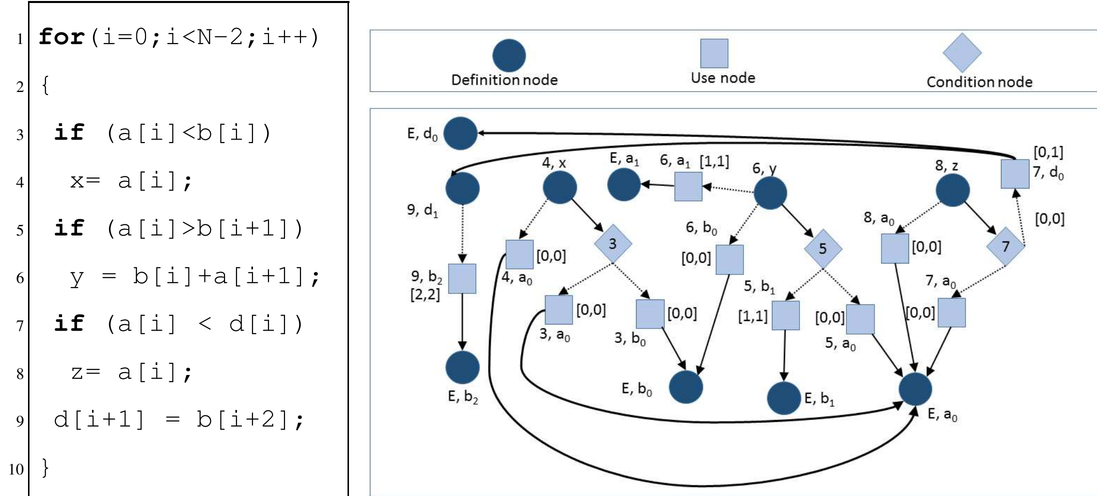


Figure 9.6: Illustration of constraints on conditions

`m1` itself. But, as explained in the discussion of the motivating example, we can reproduce the value of variable `m1` with a bound of 1. We call such cases as *self-control dependence*. Variable `m1` is called *self-controlled variable* and the condition (6), controlling the assignment to `m1`, is called *self-controlling condition*. However, for the variable `m2` involved in a similar cycle, we can not reproduce its value with a reduced bound of the loop. The difference is that the value `a[i]` being assigned to `m2` is not same as the value `b[i]` getting compared with `m2`. Based upon these observations, we define the constraints under which we can obtain a reduced bound of the loops exhibiting cyclic dependence.

Constraint 1: Every cycle in the PDG should have exactly one condition node and one definition node, and the definition node should be for a scalar variable, say x, satisfying the following:

1. There should be no additional definition node for x, except (E, x), in the PDG.
2. The expression of the condition node should be in the form of $x\ rop\ expr$, where the operator $rop \in \{<, \leq, >, \geq\}$, and $expr$ should be semantically equivalent to the RHS of assignment to x.

Consider the PDG of the code snippet of the Figure 9.6. Let `y` and `x` be assigned last in the lcvs 40 and 50, respectively. It means that the condition $\mathtt{a[i]} > \mathtt{b[i+1]}$, controlling the assignment to `y`, must be false for all the lcvs more than 40. To reproduce the same values of `y` and `x`, we need some two lcvs, say 2 and 4 respectively, in an execution of the pruned loop, and ensure that `y` is not reassigned after lcv 2. Let array `a` in the pruned loop be `a'`, To reproduce the last value of `y` and `x`, we must insure that values of `a'[2]`, `a'[3]`, `a'[4]`, `b'[2]`, `b'[3]`

and `b′[4]` are same as `a[40]`, `a[41]`, `a[50]`, `b[40]`, `b[41]` and `b[50]`, respectively. Consider the iteration corresponding to lcv 3. The controlling condition for the assignment to `y` will be `a′[3]` $>$ `b′[4]`, which is equivalent to `a[41]` $>$ `b[50]`, and, contrary to our objective, the same can be true resulting in reassignment of `y` in this iteration. Similar situations will arise no matter what bound or iterations we choose to reproduce the values. Although the array elements required to produce value of y and x were kept disjoint, the controlling condition in between the two iterations ended up comparing one element needed for y and another needed for x. Occurrence of two array elements with different offsets (`a[i]` with 0 and `b[i+1]` with 1) in the controlling condition is the reason for this problem. Such difference in offsets may not be caused by only immediate operands of the condition. Consider the controlling condition of the assignment to `z`. Although `a[i]` is compared with `d[i]`, both having offset 0, since `d[i+1]` gets assigned by `b[i+2]`, effectively `a[i]` is compared with `b[i+1]` having different offsets (0 and 1). To be able to detect such cases, we define *span* of an operand as follows:

Definition 9.2 *(Span) The* span *of a variable operand x of a loop is an interval $[c_l, c_h]$ such that, given an* lcv *i of the loop, the initial values of only those array elements that have indices in the range $[i + c_l, i + c_h]$ may affect the value of the operand x in the* lcv *i of the loop.*

If the interval of a span has a single value in it, we call it a *degenerate span*. Note that, if the variable x is a scalar having loop carried dependence, the span for the same could cover the entire array and such spans will be denoted by ∞. The *span* of the array reference `d[i]` at line 7 will be [0,1], because it is dependent on the initial value of `b[i+1]`, and possibly on the initial value of `d[i]` itself. The dependence on `b[i+1]` arises from assignment of `b[i+2]` to `d[i+1]` in the previous iteration. Later in this section, we shall describe an algorithm to compute the span for every variable operand. We define the constraint for controlling conditions as follows:

Constraint 2: For every definition node, all use nodes (except the ones of a self-controlled variable being assigned in the definition node) belonging to the conditions controlling the definition node must have a degenerate span with same value.

Effectively, the constraint means that a self-controlling condition can only control the assignment to corresponding self-controlled variable, and the span of all the operands (other than the self-controlled variables) of all the conditions controlling the same assignment must

```
annotSpan(Node n) {
 if (n is annotated)
   return span(n);
 e =comp(n);
 annotate n with e ;
 return e ;
}
compUseCond(Node n) {
 e = ∞;
 for all n' such that n → n' {
  e' =annotSpan(n') ;
  if(e' == ∞) return ∞;
  if(e == ∞) e = e';
  else e = (low(e_l, e'_l),high(e_h, e'_h));
 }
 return e;
}
```

```
comp(Node n) {
 annotate n with ∞;
 if (n is defnode(E, x)) return ∞;
 if (n is defnode(E, a_c)) return (c, c);
 if (n is usenode(l, a_c)) e =compArr(n, c);
 else e =compDefCond(n);
 return e;
}
compArr(Node n, c) {
 e = (c, c);
 for all n' : defnode(l, a_c') such that n → n' {
  e' =annotSpan(n') ; if (e' == ∞) return ∞;
  e' = (e'_l + (c − c'), e'_h + (c − c'));
  e = (low(e_l, e'_l),high(e_h, e'_h));
 }
 return e;
}
```

Figure 9.7: Algorithm for span computation

have the same single-value span.

9.3.3 Span computation

In Figure 9.7, we give an algorithm that computes the span for every use node as the smallest interval satisfying the following rules.

1. For an array reference $a[\ell + c]$, its span must contain c.
2. Span of an expression is the join of spans of its variable operands.
3. Span of a scalar use is the join of spans of the definitions on which it depends.
4. Span of an assignment is the join of span of the RHS of the assignment, and, spans of all the condition expressions controlling the assignment.
5. Span of a definition of a scalar variable, originating at the $ENTRY$ of a loop is ∞.
6. If array reference $a[\ell + c]$ depends on an assignment to $a[\ell + c']$ having span $[e_l, e_h]$, then span of $a[\ell + c]$ should include the range $[e_l + c - c', e_h + c - c']$.

As an illustrative example, we have annotated the span for all the use nodes in the PDG of Figure 9.6. The use nodes that have a well defined span, i.e. if it is not ∞, are called *array*

operand.

9.4 Replaying last value computations in the pruned loop

We assume that a given program adheres to the syntactic and semantic constraints given in Section 9.2.1 and 9.3, and is already sliced with respect to the variables used in the assert statement. We now discuss two important issues regarding the pruned loop: (i) number of iterations required to replay the last value computation of the variables of interest in the original loop, and (ii) the desired distance between successive last value computing lcvs. As we shall see later, the second issue has a bearing on the first one, therefore, first we shall take up the discussion of the second issue.

9.4.1 Distance for non-interference

Consider the PDG of a loop in the program. Let the minimum of low values of the spans over all array operands in the PDG be δ_{low}. Similarly, let the maximum of high values of the spans of all array operands in the PDG be δ_{high}. Consider two sets of array elements, say X_1 and X_2, whose initial values are likely to affect the value of the array operands in two different lcvs of the loop, say i_1 and i_2, respectively. Assume that $i_1 < i_2$. It is easy to see that X_1 and X_2 are guaranteed to be disjoint if $i_2 - i_1 > \delta_{\mathsf{high}} - \delta_{\mathsf{low}}$. We define δ as $\delta_{\mathsf{high}} - \delta_{\mathsf{low}}$. For example, consider the second loop of the program of the Figure 9.1. The array operands in the loop are `a[i-1]` and `a[i]`. Therefore, $\delta_{\mathsf{low}} = -1$, $\delta_{\mathsf{high}} = 0$, and $\delta = 1$.

As explained earlier, to replicate the values assigned to the variables of interest in the original program, we select certain iterations that collectively assign values to these variables, and, in the pruned loop, replay the computations carried out in these iterations. For this, we need to assign appropriate initial values to the array elements that decide the values of the variables of interest in the corresponding iterations of the pruned loop. It is important to remember that the objective of pruning is to reduce the size of the loop. Therefore, the distance between value producing lcvs should be made as small as possible in the pruned loop without affecting soundness. However, if this distance is too close, there may be conflicting demands of assignment of initial values to the same array element with respect to different lcvs. But if the lcvs are separated by

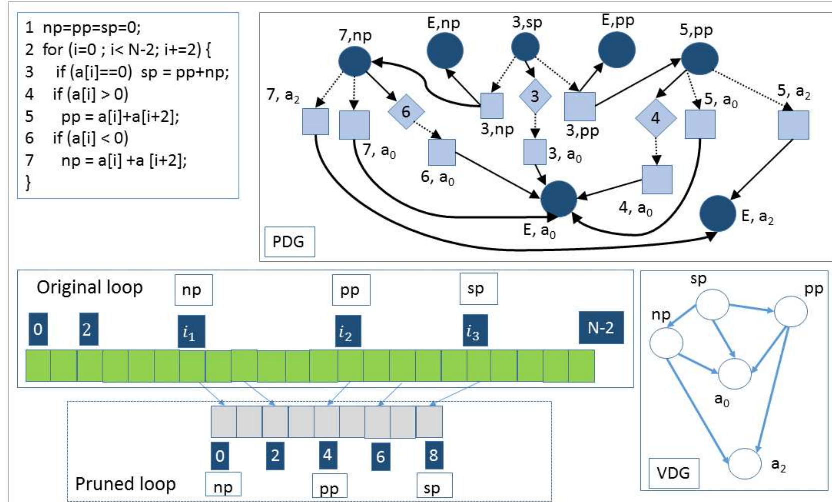


Figure 9.8: Illustration of value reproducibility of variables

more than δ then it is guaranteed that no such conflict will arise. We refer to δ as the *distance for non-interference*. Thus, for the example of Figure 9.1 in the pruned loop, the lcvs of interest should be separated by at least two.

9.4.2 Required number of iterations in the pruned loop

Consider the example given in Figure 9.8. Suppose sp, the only variable of interest, is last assigned at lcv i_3, and the values of the variables np and pp used in the assignment are last assigned at lcvs i_1 and i_2, respectively. Obviously, i_3 would be greater than both i_1 and i_2. Assume, in addition, that np is assigned before pp, and therefore $i_1 < i_2 < i_3$.

The value of np depends on the values of $\mathtt{a}[i_1]$ and $\mathtt{a}[i_1 + 2]$. The pruned loop can be organized so that the computation of np takes places at 0, and the initial values that it depends on can be made available in $\mathtt{a}[0]$ and $\mathtt{a}[2]$ through a non-deterministic choice. To produce the value of pp, we have to choose an lcv at a distance of more than δ from 0. Further, this lcv has to be a multiple of two[5]. Thus, a possible lcv to produce the value of pp is 4, and the required initial values are made available in array elements $\mathtt{a}[4]$ and $\mathtt{a}[6]$. Following a similar argument, sp can be produced at lcv 8 using $\mathtt{a}[8]$. So, a pruned loop with a loop counter bound of 8 (i.e. five iterations) can reproduce the value of sp. This example illustrates the role of inter-variable

[5]Because, the loop step is 2 and c_{init} is 0.

dependence in obtaining a bound on the lcv of the pruned loop. Observe that this bound does not change if pp happens to be computed before np. However, the bound increases if np and pp are also considered variables of interest, as we have to account for the possibility of np and pp being computed after sp. We will generalise this idea in subsequent sections.

9.4.3 Variable dependence

We define a dependence relationship among variables of a loop using the PDG of the loop. For a given loop, a variable v_1 depends on v_2 $(v_2 \neq v_1)$ if the PDG of the loop contains a path from a definition node of v_1 to a node of the form (E, v_2) devoid of any intervening definition node (l, v_3) of a variable v_3 for which the node (E, v_3) exists. For example, consider the loop in Figure 9.8. Variable sp depends on pp, np and a_0, but not on a_2 because of the intervening node (5, pp). Effectively, v_1 depends on v_2 means that the value of v_1 in an iteration may depend on the value of v_2 computed in some previous iteration.

For each loop, we construct a variable dependence graph (VDG), where the nodes are variables, and there is a directed edge from v_1 to v_2 if v_1 depends on v_2. Due to the semantic constraints discussed earlier in Section 9.3, there will be no cycles in the VDG. In Figure 9.8, we have shown the VDG of the loop constructed using its PDG. When there is an inter-PDG edge from a use node of a_c in one PDG to a definition node of $a_{c'}$ in another, we connect the corresponding variable nodes with a directed edge from a_c to $a_{c'}$. Such an edge will be called inter-VDG edge. The same is illustrated in Figure 9.10. The length of the longest path from a variable node to a leaf node will be called *dependence depth* of the variable. For a variable having a dependence depth of d, we will say that it is a d-depth variable. In our example, the dependence depth of sp is 2, and pp and np are 1-depth variables.

9.5 Last value assignments and bound on their numbers

Let P be a program in the scope of our interest. Let $\mathcal{L}$ be the set of the loops in the program, and $L_a \in \mathcal{L}$ be the property loop. Consider a trace τ of P that has a trace-point $(l_{\mathsf{a}}, i_{\mathsf{a}})_\tau$ corresponding to an occurrence of the assert statement. Let $\mathcal{L}_\tau \subseteq \mathcal{L}$, be the set of loops that occur in the trace τ. Obviously, $L_a \in \mathcal{L}_\tau$. Assume that we are given a variable x and a trace-point

in the trace τ. With respect to x and the trace-point, an assignment in the trace is called a dead assignment if the value assigned is never used in computing a value of x that reaches the trace-point. The assignments that are not dead are last value assigning for x at the trace-point. Based upon this observation, we define the notion of *last value assignments* belonging to loops $\mathcal{L}_\tau$. In the rest of this section, our discussion will be in the context of the trace τ and the trace-point $(l_\mathsf{a}, i_\mathsf{a})_\tau$.

9.5.1 Last value assignments

Let $\mathcal{P}\tau$ be a program produced from the sequence of statements corresponding to the trace τ. Note that in the program $\mathcal{P}\tau$, every array reference $a[\ell + c]$ at location (l, i) is replaced with $a[n]$, where $n = i + c$. From the program $\mathcal{P}\tau$, we create another program $\mathcal{P}\tau_s$ by replacing all self-controlling condition checks and all occurrences of the assert statement, except the one at the trace-point $(l_\mathsf{a}, i_\mathsf{a})_\tau$, by a $SKIP$ statement. Assume that a use-def chain analysis, reaching definition analysis and strongly live variable analysis [63] is available for the program $\mathcal{P}\tau_s$. Given a loop $L \in \mathcal{L}_\tau$, let $\bar{l}_L$ and $\hat{l}_L$ denote the locations of the first and last statement in the loop body, respectively. Further, let the trace-point $(\hat{l}_L, \hat{i}_L)$ denote the last trace-point in the trace τ that belongs to the loop L. We will use $live_at(l, i)$ to denote the set of strongly live variables at a trace-point (l, i) in $\mathcal{P}\tau_s$. A condition at a location (l_1, i) belonging to a loop, is said to *control* an assignment to a variable v at location (l_2, i), if the definition (l_2, v) depends on the condition (l_1) in the PDG of the loop.

Consider the set of variables that are strongly live in $\mathcal{P}\tau_s$ at a trace-point (l, i) belonging to the property loop L_a. We are interested in the assignments that belong to loops, and which are not dead in the program $\mathcal{P}\tau_s$ with respect to the value of these variables at the trace-point (l, i). Let this set be denoted as $all_lvas(l, i)$. To define this set, we introduce a notion of *last value assignments* appearing in a loop for the variables that are strongly live at a given trace-point belonging to the loop. Let L be a loop in $\mathcal{L}_\tau$ and let a trace-point (l_L, i_L) belong to the loop.

Definition 9.3 *(Last value assignments in a loop) Given a loop $L \in \mathcal{L}_\tau$ and a trace-point (l_L, i_L), the set S of last value assignments in the loop L for a variable $x \in live_at(l_L, i_L)$, with respect to the trace-point (l_L, i_L), is defined as follows:*

1. *All the definitions (assignments) to x in the program $\mathcal{P}\tau_s$, belonging to the loop L, that*

reach at the trace-point (l_L, i_L) are in S.

2. *The assignments belonging to the loop L, that are in the use-def chain of variables used in RHS of an assignment in S, are in S.*
3. *The assignments of the loop L, that are in the use-def chain of variables used in a condition that* controls *an assignment in S, are in S.*
4. *No other statement is in S.*

We will use $loop_lvas(x, l_L, i_L)$ to denote the set of last value assignments of a variable x at the trace-point (l_L, i_L), as per the definition above. We lift this notation for a set U of variables in a natural manner, and denote it as $loop_lvas(U, l_L, i_L)$. We will use notation $loop_lvas_live(l_L, i_L)$ to represent the set of last value assignments belonging to the loop L, for the set of strongly live variables at (l_L, i_L) with respect to the trace-point (l_L, i_L). In other words, $loop_lvas_live(l_L, i_L) = loop_lvas(live_at(l_L, i_L), l_L, i_L)$. Let the trace-point $(\bar{l}_{L_a}, i_\mathsf{a})$ denote the beginning of the loop iteration corresponding to $(l_\mathsf{a}, i_\mathsf{a})$. The following lemma establishes a relationship between all last value assignments and loop wise last value assignments. It can be easily shown that this holds.

Lemma 9.4 $all_lvas(\bar{l}_{L_a}, i_\mathsf{a}) = loop_lvas_live(\bar{l}_{L_a}, i_\mathsf{a}) \cup (\bigcup_{L \in \mathcal{L}_\mathcal{T} \setminus \{L_a\}} loop_lvas_live(\hat{l}_L, \hat{i}_L))$

9.5.2 Last value assigning lcv

The lcv appearing in the trace-point of a last value assignment is called the *last value assigning* lcv, and we will denote it by LVA. For example, in the code snippet of Figure 9.8 with $\mathtt{N} = 6$, and the array `a` initialised as $\{-1, 10, 0, -5, 20, 0\}$, the last assignment to `sp` (assuming it is used in the `assert` of interest) happens in lcv 5. This assignment uses the values of `pp` and `np` assigned in lcvs 4 and 3, respectively. Therefore, there are three LVA for `sp`, namely 3, 4 and 5. Given a set of last value assignments, say S, let $\|S\|$ represent the number of unique LVA corresponding to S. Since, multiple last value assignments may share the same lcv, it is obvious that $\|S\| \leq |S|$. Let $\#\mathsf{LVA}$ represent $\|all_lvas(l_\mathsf{a}, i_\mathsf{a})\|$. We observe that $\#\mathsf{LVA} = \|all_lvas(l_\mathsf{a}, i_\mathsf{a})\| \leq 1 + \|all_lvas(\bar{l}_{L_a}, i_\mathsf{a})\|$. We shall estimate a bound for $\#\mathsf{LVA}$, which will be independent of any trace.

9.5.3 Last value assignments and variable dependence graph

Consider a loop $L \in \mathcal{L}$. Let $\mathcal{G}_L$ be the VDG of the loop. For a node v of $\mathcal{G}_L$, we introduce a number *lvas-count*, and denote it as $lvas_count(v, \mathcal{G}_L)$. For a subset U of nodes in $\mathcal{G}_L$, let $lvas_count(U, \mathcal{G}_L)$ be the sum of *lvas-count* of all the elements of U. For a node v of $\mathcal{G}_L$, we define $lvas_count(v, \mathcal{G}_L)$ as follows. If v is a leaf node then $lvas_count(v, \mathcal{G}_L)$ is zero, and if v is a non-leaf node with set of children as D, then $lvas_count(v, \mathcal{G}_L) = 1 + lvas_count(D, \mathcal{G}_L)$.

Consider a loop $L \in \mathcal{L}_\tau$, a trace-point (l, i) such that $l \in \{\bar{l}_L, \hat{l}_L\}$, and a variable $x \in live_at(l, i)$. Assuming $\|loop_lvas(x, l, i)\| > 0$, there must be a definition (assignment) of x at a trace-point (l', i') belonging to the loop, and reaching (l, i)[6]. Let x' be the variable (node) in $\mathcal{G}_L$ corresponding to x[7]. Let x' be a k-depth variable. Obviously, for $k = 1$, $\|loop_lvas(x, l, i)\| = 1 = lvas_count(x', \mathcal{G}_L)$. For $k > 1$, let D be the set of variables that are strongly live at $(\bar{l}_L, i')$ due to variables used in RHS of the assignment to x at the trace-point (l', i'). Obviously, $\|loop_lvas(x, l, i)\| \leq 1 + \|loop_lvas(D, \bar{l}_L, i')\|$. Let $D_L \subseteq D$ be the set of variables having corresponding nodes in $\mathcal{G}_L$. Since, $\|loop_lvas(D \setminus D_L, \bar{l}_L, i')\| = 0$, $\|loop_lvas(D, \bar{l}_L, i')\| = \|loop_lvas(D_L, \bar{l}_L, i')\|$. Let D'_L be the set of nodes that correspond to variables in D_L. Obviously, D'_L will have to be a subset of descendants of x' in $\mathcal{G}_L$. Based upon this observation, the following lemma establishes a relationship between the number of last value assignments and the *lvas-count* of nodes.

Lemma 9.5 *Given a loop $L \in \mathcal{L}_\tau$, a trace-point (l, i) such that $l \in \{\bar{l}_L, \hat{l}_L\}$, and a variable $x \in live_at(l, i)$, the following is true:*

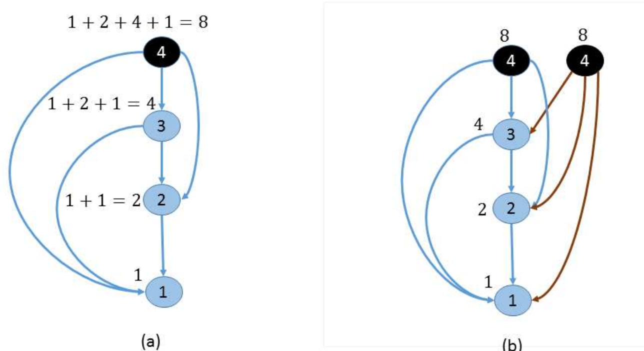
$\|loop_lvas(x, l, i)\| \leq lvas_count(x', \mathcal{G}_L)$ *(where x' is node corresponding to variable x).*

9.5.4 A theoretical bound on #LVA

Let $\mathcal{G}$ be the integrated view of VDGs of the loops considering inter-loop dependence edges. We observe that in presence of array elements that get accessed outside loops, but which may be modified inside the loops, the graph $\mathcal{G}$ may not accurately represent the dependence relations, particularly the ones involving array elements. Therefore, for now, we assume that such array accesses are not there in the program. We will derive a theoretical bound on #LVA for such pro-

[6]There will be only one such assignment.

[7]If x is scalar, x' is same as x, else if x is an array element $a[n]$ then x' is a_c such that $i' + c = n$.



(Number inside a circle denotes degree of dependence and that outside a circle denotes #LVA for the node)

Figure 9.9: Illustration of computing a theoretical bound on $\#$LVA

grams in terms of the number of non-leaf variables, M, and the maximum depth of dependence, k, in $\mathcal{G}$. Obviously, $M \geq k$. Let U be the set of M non-leaf variables. From the discussion of Section 9.5.3, Lemma 9.4, and the observation made in Section 9.5.2, it can be shown that $\#\text{LVA} \leq 1 + \mathit{lvas_count}(U, \mathcal{G})$. Let us consider these two cases:

Case 1 $M = k$: The sum $\mathit{lvas_count}(U, \mathcal{G})$ will be maximal when every variable of depth i is dependent on all $i - 1$ variables of lesser depth, as illustrated in Figure 9.9(a) for $k = 4$. For a 4-depth variable, the maximum possible value of *lvas-count* would be 8. It can be easily shown that for this kind of graph, $\mathit{lvas_count}(U, \mathcal{G})$ is $2^k - 1$. So $\#\text{LVA} \leq 2^k$.

Case 2 $M > k$: It can be shown that $\mathit{lvas_count}(U, \mathcal{G})$ can be maximal when $M - k + 1$ variables have depth k, and all of them have dependence graph as per the Figure 9.9(a) with remaining $k - 1$ variables of lesser depth. The arrangement is illustrated in Figure 9.9(b) for $k = 4$ and $M = 5$, where nodes with k-depth are shown in dark shade. It is obvious that for this kind of graph $\mathit{lvas_count}(U, \mathcal{G})$ will be $(M - k)2^{k-1} + 2^k - 1$, and therefore, upon simplification, $\#\text{LVA} \leq (M - k + 2)2^{k-1}$.

Therefore, in both the cases, we get $\#\text{LVA} \leq (M - k + 2)2^{k-1}$. To illustrate with an example, consider the code snippet given in Figure 9.10. There are a total of 5 non-leaf nodes, i.e. $U = \{\mathtt{m1}, \mathtt{m2}, \mathtt{m3}, \mathtt{a}_0, \mathtt{a}_{-1}\}$, in the integrated graph $\mathcal{G}$, with the maximum dependence depth of 3. So, with $M = 5$ and $k = 3$, we get 16 as theoretical upper bound for $\#$LVA.

9.5.5 Computing a tighter upper bound on $\#$LVA

The bound explained in previous section is a theoretical bound. For the program of Figure 9.10, in a trace there can be at most 3 LVAs for the second loop (one each for assignment to `m1`, `m2` and `m3`) and 3 LVAs for the first loop (one each for assignment to the array elements assigned to `m1`, `m2` and `m3` in the second loop). It is obvious that there will be some traces in which total number of LVAs will indeed be 6. So, a bound of 6 on $\#$LVA of the program is necessary and sufficient. However, the worst case bound was computed as 16. Even by using the Lemma 9.5 over the connected VDGs as a single VDG, we get a higher bound of 10. Can we compute a lower bound which is closer to the actual bound of 6, based upon the VDGs of the individual loops? By simply computing the bound on $\#$LVA of individual VDGs, ignoring any inter-VDG edges, we get a bound of 3 from VDG of the second loop and 1 from VDG of the first loop, and from this we get an unsound bound of 5 as it is less than 6. We observe that in this process we treated the node a_{-1} like a normal variable and accounted it for just one value, while, in reality, we need three values from the first loop as required for computation of `m1`, `m2` and `m3` in the second loop. Therefore, we need to identify the number of array elements (*instances*) that a relative array element node represents in a VDG (a_{-1} in this case).

We still assume that there are no array accesses outside the loops in the given program. We will address the issues involved due to presence of such array accesses later in Section 9.5.6. Before computing the bound using individual VDGs, we identify the number of instances, *instance-count*, of every node in all the VDGs. Initially, we annotate every node with an *instance-count* of 0. In the VDG of every loop, we annotate the nodes for scalar variables that are live at the end of the loop (as per the usual live variable analysis of the program) with an *instance-count* of 1. For the property checking loop, we mark the nodes of all variables that are live at the start of the loop with an *instance-count* of 1. To illustrate this with an example, consider the code snippet of Figure 9.10. Assuming that `m1`, `m2` and `m3` are live at the end of the second loop, the corresponding nodes are annotated with an *instance-count* of 1, as shown in small rectangles in the VDG.

We take a topological ordering of the loops of the program, such that if a loop L_2 can get executed before another loop L_1, then L_1 is placed before L_2 in the ordering. Obviously, the property checking loop will be the first one in this ordering. For each loop in this order, we com-

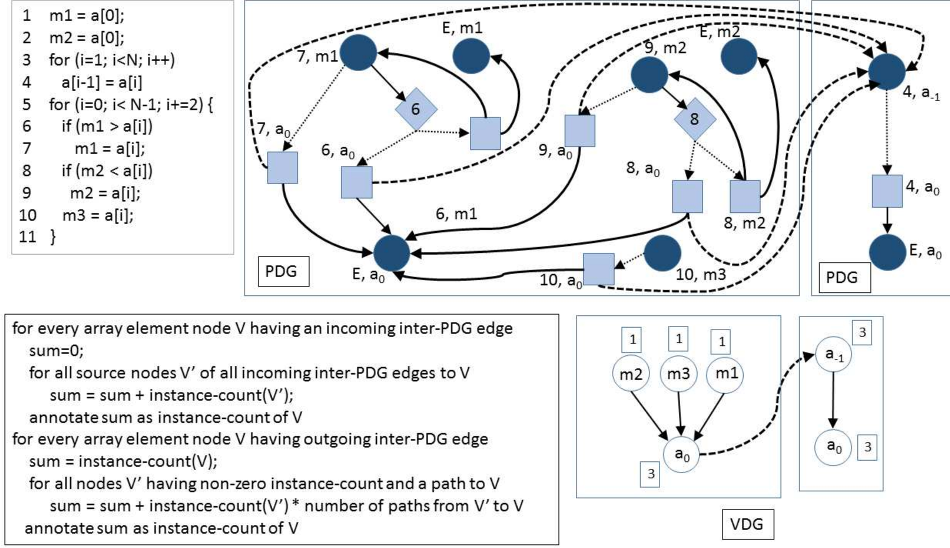


Figure 9.10: Algorithm and illustration for computing *instance-count*

pute and annotate each node with its *instance-count*, using the algorithm given in Figure 9.10. The computed *instance-count* of 3, for the nodes corresponding to a_0 and a_{-1}, is annotated in the VDG of each loop.

For every node in the VDG of every loop, we compute the *lvas-count* of the node. Let $\mathcal{C}_L$ be the sum of product of *lvas-count* and *instance-count* of each node in $\mathcal{G}_L$. For a given integrated graph, let us define $\mathcal{C}$ to be $1 + \sum_{L \in \mathcal{L}} \mathcal{C}_L$. For example, $\mathcal{C}$ would be 7 for the program of Figure 9.10. We claim that $\#\mathsf{LVA} \leq \mathcal{C}$.

Theorem 9.6 *Assuming no array accesses outside the loops, for the trace τ and the trace-point $(l_{\mathsf{a}}, i_{\mathsf{a}})$, the total number of distinct* LVA*s for the variables used in the* **assert** *statement at the trace-point is at most $\mathcal{C}$, i.e. $\#\mathsf{LVA} \leq \mathcal{C}$.*

Proof Let $\mathcal{L}'_\tau = \mathcal{L}_\tau \setminus \{L_a\}$. Note that, for a loop $L \in \mathcal{L}'_\tau$, the number of elements of an array a that occur in $live_at(\hat{l}_L, \hat{i}_L)$, and are modified as a_c in the loop, will not be more than the *instance-count* of the node a_c in $\mathcal{G}_L$. Similarly, the *instance-count* of the scalar variables that occur in $live_at(\hat{l}_L, \hat{i}_L)$, and are modified in the loop, will be 1 in $\mathcal{G}_L$. Using Lemma 9.5, we can show that for a loop $L \in \mathcal{L}'_\tau$, $\|loop_lvas_live(\hat{l}_L, \hat{i}_L)\| \leq \mathcal{C}_L$, and $\|loop_lvas_live(\bar{l}_{L_a}, i_{\mathsf{a}})\| \leq \mathcal{C}_{L_a}$. By Lemma 9.4,

$$\|all_lvas(l_{\mathsf{a}}, i_{\mathsf{a}})\| \leq 1 + \|loop_lvas_live(\bar{l}_{L_a}, i_{\mathsf{a}})\| + \sum_{L \in \mathcal{L}'_\tau} \|loop_lvas_live(\hat{l}_L, \hat{i}_L)\|.$$

Since $\mathcal{L}'_\tau \subseteq \mathcal{L}$ and $L_a \in \mathcal{L}$, the following is obvious:

$$\#\mathsf{LVA} \;\leq\; (1 + \mathcal{C}_{L_a} + \sum_{L \in \mathcal{L}'_\tau} \mathcal{C}_L) \leq (1 + \sum_{L \in \mathcal{L}} \mathcal{C}_L) \;=\; \mathcal{C}. \qquad \blacksquare$$

9.5.6 Issues due to array accesses outside loops

When arrays are accessed outside the loops, as per our constraints on the scope of programs of interest, such accesses will only have constant indices. Due to presence of such accesses, the instance-count, computed in Section 9.5.5 for array elements, may be smaller than necessary. As a result, Theorem 9.6 will not be applicable for such programs. To address this issue, we first find out the largest lcv of iterations in which array elements accessed outside the loops can be modified inside some loop. For illustration, we refer to the program in Figure 9.11. Let β be the largest constant index used in the program, e.g. $\beta = 10$ for this program. For a given loop, let $\mathcal{K}_{const}$ be the largest lcv with which some array element, that is accessed with a constant index outside a loop, can be accessed at an array reference inside the loop. Obviously, $\mathcal{K}_{const}$ will be the smallest lcv of the loop such that $\mathcal{K}_{const} - \delta_{\mathsf{low}} \geq \beta$. Note that, $\mathcal{K}_{const} \geq \mathsf{c}_{init}$ for the loop. For this example, $\mathcal{K}_{const}$ is 14, 13 and 16 for the first, second and third loop, respectively. Let $\mathcal{K}^c$ be the maximum of $\mathcal{K}_{const}$ of all the loops, which is 16 for this example. Let $\overline{\mathsf{c}_{init}}$ be the minimum c_{init} across all the loops. It is guaranteed that the number of distinct lcvs, corresponding to last value assignments arising due to modification of the array elements accessed outside the loops, can not be more than $\mathcal{K}^c - \overline{\mathsf{c}_{init}}$. Therefore, for such cases, $\#\mathsf{LVA} \leq \mathcal{C} + \mathcal{K}^c - \overline{\mathsf{c}_{init}}$.

9.5.7 Pruned program construction

Consider the program of Figure 9.10. Suppose the LVAs for the first loop are 101, 201 and 301, for `a[100]`, `a[200]` and `a[300]`, respectively. We can not produce the value of `a[100]` and `a[200]` in the first pruned loop in lcv 6 and 16, in `a[5]` and `a[15]` respectively. This is because the same can not be accessed by the second pruned loop, which accesses only even-indexed array elements. So, the lcvs in the first pruned loop producing the value of desired array elements must be odd. That is, the difference between corresponding LVAs in the pruned and the original first loop must be a multiple of 2, which is the step count of the second loop. This illustrates that for a loop producing value of some array element, the LVAs in the pruned loop must be such that their distance from the corresponding LVAs in the original loop is divisible by

```
#define N 10000
main(){
 int a[N], i, m1, m2;
 m1 =a[8];
 for (i=0; i< N-4; i+=2)
   if (m1 > a[i+4]) m1 = a[i+4];
 m2 = a[10];
 for (i=10; i< N-10 i+=3)
   if (m2 > a[i-3]) m2 = a[i-3];
 for (i=10; i< N-35; i+=6)
  assert (m1<=a[i] && m2 <= a[i]);
}
```

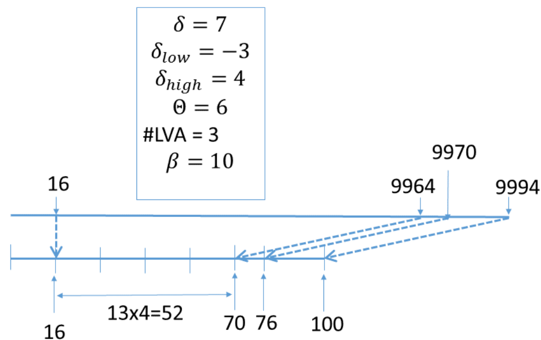


Figure 9.11: Illustration of bound computation

the step counts of all the loops which use the value of the array element.

Suppose for a given trace τ of the program P, we get a trace τ^p of the pruned program P' where the same value of all the desired variables is produced. Let i_1 and i_2, $i_1 < i_2$, be two consecutive LVA in the trace τ, and i'_1 and i'_2 be the corresponding LVA in the trace τ^p. If $i_2 - i_1 > \delta$, then $i'_2 - i'_1$ also must be more than δ to ensure non-interference, as discussed earlier. To be conservative, we compute δ_{low}, δ_{high} and δ from the spans of all array operands of all the loops in the program. For example, for the program of Figure 9.10, $\delta_{\text{low}} = -1$, $\delta_{\text{high}} = 0$ and $\delta = 1$. In addition, as illustrated above, the difference $i_1 - i'_1$ must be divisible by the steps of all the loops that use the value produced by the LVA i_1. Since statically we do not know which all loops may use the value produced, to be conservative, we say that $i_1 - i'_1$ should be divisible by the $\mathbf{lcm}$ of all the steps. We call this $\mathbf{lcm}$ as the synchronising distance, and denote it by Θ. Therefore, $i'_1 = i_1 - m_1\Theta$ for some $m_1 \geq 0$, and by similar argument, $i'_2 = i_2 - m_2\Theta$ for some $m_2 \geq 0$. To have the least possible bound of pruned loops, m_1 and m_2 should be as large as possible. In order to satisfy these constraints, the worst case gap in i'_2 and i'_1 will be $\delta + \Theta$. Therefore, our pruned loop bound should have a provision for a gap of $\delta + \Theta$ between neighboring LVAs.

We illustrate bound computation for the pruned loops through the program in Figure 9.11. The values of δ, Θ and #LVA are shown there. The array elements accessed with constant

indices outside the loops, e.g. `a[8]` and `a[10]`, can be processed inside the loops also. For example, `a[8]` in the first loop in lcv 4, and `a[10]` in lcv 13 in the second loop. Since we need to keep these accesses with same index, the lcvs in which these may get accessed inside loops must remain same in the pruned program also. As discussed in Section 9.5.6, we will need to keep lcvs up to $\mathcal{K}^c$ as is in the pruned program. The c_{final} for the loops are 9994, 9970 and 9964 for the first, second and third loop, respectively. We consider maximum and minimum c_{final}, denoted as $\widehat{\mathbb{N}}$ and $\overline{\mathbb{N}}$, as special LVAs. In this example, $\widehat{\mathbb{N}} = 9994$ and $\overline{\mathbb{N}} = 9964$. We keep the distance between maximum and minimum c_{final} of the pruned program same as it is in the given program. Therefore, to keep provisions for (a) keeping lcvs up to $\mathcal{K}^c$ as is, (b) keeping a gap of $\delta + \Theta$ in between two neighbouring LVAs, and (c) keeping same distance between $\widehat{\mathbb{N}}$ and $\overline{\mathbb{N}}$, we need a bound of at least $\mathcal{K}^c + (\mathcal{C} + 1)(\delta + \Theta) + (\widehat{\mathbb{N}} - \overline{\mathbb{N}})$. We add 1 to $\mathcal{C}$ to keep the required gap before and after the first and last LVA. Let this number be Γ', which is $16+4\times13+(9994-9964) = 98$ for this example. Since we want to map the maximum c_{final} to corresponding maximum c_{final} of the pruned program, the difference between final bound and $\widehat{\mathbb{N}}$ should be divisible by Θ. So, we pick the smallest number $\Gamma \geq \Gamma'$ such that $\widehat{\mathbb{N}} - \Gamma = m\Theta$, for some $m \geq 0$. In this example, Γ will be 100. The number Γ is the bound [8] for the loop with maximum c_{final}. The bound for individual loops is arrived at by relative shift, i.e. the bound c'_{final}, for a loop having its original bound as c_{final}, will be $\Gamma - (\widehat{\mathbb{N}} - \mathsf{c}_{final})$. The bounds in our example come out to be 70,76 and 100 for the third, second and first loop, respectively.

From a given program P, we construct a pruned program P' by replacing every loop $L \equiv$ **for (ℓ=start; $\ell <$ end; ℓ+=t)** with $L' \equiv$ **for (ℓ=start; $\ell <$ end && $\ell \leq \mathsf{c}'_{final}$; ℓ+=t)**. Barring this change, the pruned program P' is exactly the same as P.

We will show that Claim 9.1 holds with Γ as the upper bound of loop counter values for the pruned loops.

9.6 Proof of soundness

For a given trace τ of the program P, and a trace-point $(l_{\mathsf{a}}, i_{\mathsf{a}})_\tau$ for the **assert** statement, we will construct a trace τ^p of the pruned program P', and identify a trace-point $(l_{\mathsf{a}}, {i_{\mathsf{a}}}')_{\tau^p}$ to prove Claim 9.1. Note that in constructing P', $\Gamma \leq \widehat{\mathbb{N}}$. If $\Gamma = \widehat{\mathbb{N}}$ then P' will be equivalent to P, and

[8] If Γ' is more than $\overline{\mathbb{N}}$, then we take $\widehat{\mathbb{N}}$ as our Γ.

Claim 9.1 will hold trivially. So we assume that $\Gamma < \widehat{\mathbb{N}}$. Obviously, with this assumption $\mathcal{K}^c$ will have to be less than $\overline{\mathbb{N}}$. Let X be the set of variables used in the **assert** at the trace-point $(l_a, i_a)_\tau$. We construct the initial state σ_0^p for τ^p, using the initial state σ_0 of τ and an index map χ, such that for an array element $a[n]$, $\sigma_0^p(a[n]) = \sigma_0(a[\chi(n)])$, and for a scalar variable x, $\sigma_0^p(x) = \sigma_0(x)$. We create the map χ by identifying a distinct LVA in τ^p for every LVA identified for X in τ.

9.6.1 Index mapping function

Let the set of the lcvs corresponding to all the last value assignments of X belonging to a loop be called the *loop-LVAs* of the loop. We add the c_{final} of every loop to its *loop-LVAs*. We also add i_a to the *loop-LVAs* of the property checking loop. Let $\mathbb{Q}$ be the union of $\{\mathcal{K}^c\}$ and the *loop-LVAs* of all the loops. Let $\eta = |\mathbb{Q}|$, and $\Pi = [\mathcal{K}_1, \mathcal{K}_2, ..., \mathcal{K}_\eta]$ denote the elements of $\mathbb{Q}$ arranged in ascending order of their value. Obviously, $\mathcal{K}_\eta = \widehat{\mathbb{N}}$, and $\mathcal{K}^c$ and $\overline{\mathbb{N}}$ are in Π. Let w and u be such that $\mathcal{K}_w = \overline{\mathbb{N}}$ and $\mathcal{K}_u = \mathcal{K}^c$. Since $\mathcal{K}^c < \overline{\mathbb{N}}$, $u < w$. And by Theorem 9.6, $w - u \leq \mathcal{C} + 1$. The way Γ is computed, we claim that there exists a sequence of numbers $\Pi' = [\mathcal{K}'_1, \mathcal{K}'_2, ...\mathcal{K}'_\eta]$, arranged in ascending order, satisfying the following constraints:

1. $\mathcal{K}'_\eta = \Gamma$ and $\forall i \in [1, \eta] \,.\, (\mathcal{K}_i \leq \mathcal{K}^c \implies \mathcal{K}_i = \mathcal{K}'_i) \wedge (\exists m \,.\, m \geq 0 \wedge \mathcal{K}'_i = \mathcal{K}_i - m\Theta)$
2. $\forall i \in [1, \eta - 1] \,.\, \mathcal{K}'_{i+1} - \mathcal{K}'_i \leq \delta \implies \mathcal{K}'_{i+1} - \mathcal{K}'_i = \mathcal{K}_{i+1} - \mathcal{K}_i$

We create the index mapping function $\chi : \mathbb{Z} \cup \{\bot\} \to \mathbb{Z} \cup \{\bot\}$ as follows:

$$\chi(i) = \begin{cases} i & \text{if } i = \bot \vee i \in (-\infty, \mathcal{K}'_2 + \delta_{\mathsf{low}} - 1] \\ \mathcal{K}_j + (i - \mathcal{K}'_j) & \text{if } \exists j \in [2, \eta] \text{ such that } i \in [\mathcal{K}'_j + \delta_{\mathsf{low}}, \mathcal{K}'_{j+1} + \delta_{\mathsf{low}} - 1] \\ \mathcal{K}_\eta + (i - \mathcal{K}'_\eta) & \text{if } i \in [\mathcal{K}'_\eta + \delta_{\mathsf{low}}, \infty) \end{cases}$$

Let $im(\chi)$ be the image of mapping function χ. Obviously, the inverse function $\chi^{-1} : im(\chi) \to \mathbb{Z} \cup \{\bot\}$ is well defined, and the function χ satisfies the following property.
$\forall i, j \in \mathbb{Z} \,.\, (i \leq \beta \implies \chi(i) = i) \wedge (\exists m \geq 0 \,.\, \chi(i) - i = m\Theta) \wedge (i < j \implies \chi(i) < \chi(j))$.

9.6.2 Equivalence of the two traces

Let $\mathbb{V}'$ be the set of variables in the pruned program P', excluding the loop counter variables. Let $\lambda : \mathbb{V}' \to \mathbb{V}$ be such that for every $v \in \mathbb{V}'$, if v is $a[n]$ then $\lambda(v) = a[\chi(n)]$, otherwise $\lambda(v) = v$.

For $v \in im(\lambda)$, let v' denote $\lambda^{-1}(v)$. Let the initial state σ_0^p for the trace τ^p be constructed as $\forall v \in \mathbb{V}'.\sigma_0^p(v) = \sigma_0(\lambda(v))$. We will prove Claim 9.1 for τ, i_a, τ^p and $i_a{}' = \chi^{-1}(i_a)$.

Let a trace-point $(l, i)_\tau$ be *relevant* if either $i = \bot$, or i is an LVA of the loop to which l belongs. Note that $(l_a, i_a)_\tau$ is a relevant trace-point. From the trace τ, we create a program $\mathcal{P}^r$ which is the sequence of statements that correspond to the *relevant* trace-points of τ. In addition, the statements corresponding to initialisation, test and increment of the loop counter variables are replaced by a $SKIP$ statement in $\mathcal{P}^r$. Note that in the program $\mathcal{P}^r$, every array reference $a[\ell + c]$ at location (l, i) is replaced with $a[n]$, where $n = i + c$. We create another program $\mathcal{P}_s^r$ from $\mathcal{P}^r$ by replacing all condition checks with $SKIP$ statement. The trace-points of τ will represent the program locations in both $\mathcal{P}^r$ and $\mathcal{P}_s^r$. In the discussion that follows, a trace-point would always mean a relevant trace-point. Let $LV(\mathcal{P}^r, (l, i))$ and $LV(\mathcal{P}_s^r, (l, i))$ be the set of live variables at the location (l, i) in the programs $\mathcal{P}^r$ and $\mathcal{P}_s^r$ respectively, assuming X is live at the location (l_a, i_a). For a location (l, i) belonging to a loop, i.e. $i \neq \bot$, let $\mathbb{V}_{sc}(l)$ denote the set of self-controlled variables of the loop. For a trace-point $(l, i)_\tau$, we define $RLV(l, i)$ as follows: $RLV(l, \bot) = LV(\mathcal{P}^r, (l, \bot))$, and for $i \neq \bot$, $RLV(l, i) = (LV(\mathcal{P}^r, (l, i)) \setminus \mathbb{V}_{sc}(l)) \cup LV(\mathcal{P}_s^r, (l, i))$.

Let $entry$ represent the location of the loop counter variable initialisation of a loop, which is the entry point for the loop. Let $exit$ be the location of the statement just after a loop[9]. Note that the set $RLV(l, i)$ contains no loop counter variables and it is a subset of $\mathbb{V}_{im}$. Let v be a self-controlled variable in a loop. Obviously, for the loop, if $v \in RLV(exit, \bot)$ then $v \in RLV(entry, \bot)$ also. Moreover, v belongs to $RLV(l, i)$, where l belongs to the loop and $i \neq \bot$, only if its value at location (l, i) is used later in some assignment within the loop. We define a restricted notion of equivalence, called *modulo-index equivalence*, between the program states of τ^p and τ, corresponding to a trace-point.

Definition 9.7 (Modulo-index equivalence) *Given a trace-point $(l, i)_\tau$, and the execution states $\omega \equiv ((l, i), \sigma)_\tau$ and $\omega' \equiv ((l, \chi^{-1}(i)), \sigma^p)_{\tau^p}$, the execution state ω' is* modulo-index equivalent *to the execution state ω (written as $\omega' \preceq \omega$) if $\forall x \in RLV(l, i).\sigma^p(x') = \sigma(x)$.*

Note that, since $LV(\mathcal{P}_s^r, (l_a, i_a)) = X$, $RLV(l_a, i_a) = X$, and to prove Claim 9.1, it is enough to show that $((l_a, i_a{}'), \sigma^p)_{\tau^p} \preceq ((l_a, i_a), \sigma)_\tau$. Given an expression e in the trace τ, the corre-

[9]In a limiting case where there is no statement after the loop, $exit$ is same as $EXIT$

sponding expression in the trace τ^p, denoted as e', is equivalent to the expression produced by replacing every variable v of e by v'. Note that, since constant indices used outside the loop map to themselves, this observation holds for expressions within a loop, as well as outside it. Evaluation of e in a state σ is denoted by $[\![e]\!]_\sigma$. Given an expression e at a trace-point $(l,i)_\tau$, if $((l,\chi^{-1}(i)),\sigma^p)_{\tau^p} \preceq ((l,i),\sigma)_\tau$, we observe that if the set of variables used in e, denoted as $vars(e)$, is a subset of $RLV(l,i)$, then $[\![e]\!]_\sigma = [\![e']\!]_{\sigma^p}$.

For ease of exposition, we assume that the $relop$ in self-controlling conditions is the *less-than* ($<$) operator. Due to the constraints (1) and (2) discussed in Section 9.3, without loss of generality, a controlling condition or expression e of a self-controlling condition $v < e$ is expressible as a function $f(a_c)$, where the value of $a[\ell + c]$ is from the initial state. Given an lcv i' of a pruned loop, since $\sigma^p_0(a[i'+c]) = \sigma_0(a[\chi(i')+c])$, the value of every controlling condition and expression e of a self-controlling condition in the lcv i' in the pruned loop will match those in the lcv $\chi(i')$ of the corresponding original loop.

For a loop, let *first* and *last* represent the location of the loop exit condition check and the loop counter increment of the loop, respectively. So, *first* and *last* are the first and last statement in each iteration of the loop. Given a trace $\hat{\tau}$, and the lcv i of a loop, we use $firstst(\hat{\tau},i)$ and $lastst(\hat{\tau},i)$ for the execution states $((first,i),\sigma)_{\hat{\tau}}$ and $((last,i),\sigma)_{\hat{\tau}}$. We observe that the value of a self-controlled variable of a loop will be monotonically increasing in subsequent iterations of the loop. We show that value of a self-controlled variable get synchronised in τ^p and τ in every LVA in which the self-controlled variable is assigned in τ.

Lemma 9.8 *Given a self-controlled variable v of a loop, for all the* lcvs *i' of the pruned loop, if $firstst(\tau^p,i')(v) \leq firstst(\tau,\chi(i'))(v)$ then:*

1. $lastst(\tau^p,i')(v) \ \leq \ lastst(\tau,\chi(i'))(v)$
2. *If v is assigned in the* lcv *$\chi(i')$ of the loop in τ, then,* $lastst(\tau^p,i')(v) = lastst(\tau,\chi(i'))(v)$.

Proof Assume that the self-controlling condition is $v < f(a_c)$, and the assignment to v is $v = f(a_c)$, where the value of a_c comes from the initial state. Let $m'_1 \equiv firstst(\tau^p,i')(v) \leq m_1 \equiv firstst(\tau,\chi(i'))(v)$. Let $m'_2 \equiv lastst(\tau^p,i')(v)$ and $m_2 \equiv lastst(\tau,\chi(i'))(v)$. As we had observed earlier, every controlling condition and $f(a_c)$ will have matching value in the lcv $\chi(i')$ and i' of the loop in τ and τ^p, respectively. Let $m = f(a_c)$ in the lcv i' of the loop in τ^p. If v is not assigned in τ^p then it would not be assigned in τ also. Therefore, $m'_2 = m'_1 \leq m_1 = m_2$

and from contrapositivity, $m_2' = m_2$. In the remaining case, when v is assigned in τ^p but not in τ, $m_2' = m \leq m_1 = m_2$. From this, it is obvious that the lemma holds. ∎

Lemma 9.9 *Given a self-controlled variable v of a loop, if the value of v is same at the beginning of the loop in τ and τ^p then the following holds:*

1. $\forall i \in$ loop-LVAs .$\mathit{firstst}(\tau^p, \chi^{-1}(i))(v) \leq \mathit{firstst}(\tau, i)(v)$
2. $v \in RLV(entry, \bot) \implies \mathit{lastst}(\tau^p, \mathsf{c}'_{\mathit{final}})(v) = \mathit{lastst}(\tau, \mathsf{c}_{\mathit{final}})(v)$

Proof We are given that $\mathit{firstst}(\tau^p, \mathsf{c}'_{\mathit{init}})(v) = \mathit{firstst}(\tau, \mathsf{c}_{\mathit{init}})(v)$. Let the loop step be t. Let i' be an lcv of the pruned loop which will be in the form $\mathsf{c}'_{\mathit{init}} + h't$, where $h' \geq 0$ and $\mathsf{c}_{\mathit{init}} + h't \leq \mathsf{c}'_{\mathit{final}}$. Using induction on h' and Lemma 9.8, we can show following:

$\mathit{firstst}(\tau^p, i')(v) \leq \mathit{firstst}(\tau, \chi(i'))(v)$ and $\mathit{lastst}(\tau^p, i')(v) \leq \mathit{lastst}(\tau, \chi(i'))(v) \qquad (C1)$

For $i \in$ *loop-LVAs*, $\chi^{-1}(i)$ is an lcv in the pruned loop, and using $(C1)$, we get:

$\mathit{firstst}(\tau^p, \chi^{-1}(i))(v) \leq \mathit{firstst}(\tau, i)(v)$.

Assume that $v \in RLV(entry, \bot)$. If v is not assigned in the loop in τ, it would not be assigned in the loop in τ^p also, and $\mathit{lastst}(\tau^p, \mathsf{c}'_{\mathit{final}})(v) = \mathit{lastst}(\tau, \mathsf{c}_{\mathit{final}})(v)$, trivially. Assume that v is assigned in the loop in τ. Obviously, the last assignment to v in the loop must be in some LVA, say i. So, $\mathit{lastst}(\tau, \mathsf{c}_{\mathit{final}})(v) = \mathit{lastst}(\tau, i)(v)$. As per Lemma 9.8, $\mathit{lastst}(\tau, i)(v) = \mathit{lastst}(\tau^p, \chi^{-1}(i))(v) \leq \mathit{lastst}(\tau^p, \mathsf{c}'_{\mathit{final}})(v) \leq \mathit{lastst}(\tau, \mathsf{c}_{\mathit{final}})(v)$. Therefore, $\mathit{lastst}(\tau^p, \mathsf{c}'_{\mathit{final}})(v) = \mathit{lastst}(\tau, \mathsf{c}_{\mathit{final}})(v)$. ∎

For an LVA i of a loop, let the self-controlling condition for a self-controlled variable v be at a trace-point $(l, i)_\tau$. Since the only assignment to v is controlled by the self-controlling condition, the value of v will be same at $(\mathit{first}, i)_\tau$ and $(l, i)_\tau$. Therefore, as per Lemma 9.9:

$((l, \chi^{-1}(i)), \sigma^p)_{\tau^p}(v) \leq ((l, i), \sigma)_\tau(v) \qquad (C2)$

In the following lemma, we show that the state equivalence between τ and τ^p is preserved from one LVA to the next LVA in every loop.

Lemma 9.10 *If i_1 and i_2, $i_1 < i_2$, are the successive* LVA *of a loop, then the following holds:* $\mathit{lastst}(\tau^p, \chi^{-1}(i_1)) \preceq \mathit{lastst}(\tau, i_1) \implies \mathit{firstst}(\tau^p, \chi^{-1}(i_2)) \preceq \mathit{firstst}(\tau, i_2)$

Proof Let $v \in RLV(\mathit{first}, i_2)$. Since i_1 and i_2 are the successive LVA of the loop, v must not be assigned in between $(\mathit{last}, i_1)_\tau$ and $(\mathit{first}, i_2)_\tau$. And therefore, $v \in RLV(\mathit{last}, i_1)$. Assume

that $\mathit{lastst}(\tau^p, \chi^{-1}(i_1)) \preceq \mathit{lastst}(\tau, i_1)$. So, $\mathit{lastst}(\tau^p, \chi^{-1}(i_1))(v') = \mathit{lastst}(\tau, i_1)(v)$. We need to show that $\mathit{firstst}(\tau^p, \chi^{-1}(i_2)) \preceq \mathit{firstst}(\tau, i_2)$, and for that it is sufficient to show that $\mathit{firstst}(\tau^p, \chi^{-1}(i_2))(v') = \mathit{firstst}(\tau, i_2)(v)$.

If there are no assignments to v in the loop then following is obvious.

$$\mathit{firstst}(\tau^p, \chi^{-1}(i_2))(v') = \mathit{lastst}(\tau^p, \chi^{-1}(i_1))(v') = \mathit{lastst}(\tau, i_1)(v) = \mathit{firstst}(\tau, i_2)(v)$$

Let t be the loop step. If $i_2 = i_1 + t$, then it is obvious that $\chi^{-1}(i_2) = \chi^{-1}(i_1) + t$. And therefore, for this case also the above argument holds trivially, because:

$$\mathit{lastst}(\tau^p, \chi^{-1}(i_1))(v') = \mathit{firstst}(\tau^p, \chi^{-1}(i_2))(v') \text{ and } \mathit{lastst}(\tau, i_1)(v) = \mathit{firstst}(\tau, i_2)(v)$$

Assume that there exists some assignment to v in the loop and $i_2 - i_1 > t$. Obviously, all the assignments to v in the loop must be conditional, and the path controlling condition for every assignment to v must be false from $(\mathit{last}, i_1)_\tau$ to $(\mathit{first}, i_2)_\tau$. For the loop, consider an lcv $i' = \chi^{-1}(i_1) + h't$, with $h' > 0$, and $i' \in (\chi^{-1}(i_1), \chi^{-1}(i_2))$. Obviously, $\chi(i') \in (i_1, i_2)$. Consider execution states $\omega'_f \equiv \mathit{firstst}(\tau^p, i'), \omega_f \equiv \mathit{firstst}(\tau, \chi(i')), \omega'_l \equiv \mathit{lastst}(\tau^p, i')$, and $\omega_l \equiv \mathit{lastst}(\tau, \chi(i'))$. Assume that $\omega'_f(v') = \omega_f(v)$. As observed earlier, since all the controlling conditions to all the assignments to v are false in the lcv $\chi(i')$ of the loop in τ, they must be false in the lcv i' of the loop in τ^p too. It means that $\omega'_l(v') = \omega'_f(v') = \omega_f(v) = \omega_l(v)$. In other words, the following holds:

$$\mathit{firstst}(\tau^p, i')(v') = \mathit{firstst}(\tau, \chi^{-1}(i'))(v) \implies \mathit{lastst}(\tau^p, i')(v') = \mathit{lastst}(\tau, \chi^{-1}(i'))(v) \quad (C3)$$

Further, the equality $\mathit{firstst}(\tau^p, \chi^{-1}(i') + t)(v) = \mathit{lastst}(\tau^p, \chi^{-1}(i'))(v)$ is obvious.

Since $\mathit{lastst}(\tau^p, \chi^{-1}(i_1))(v') = \mathit{lastst}(\tau, i_1)(v)$, using the equality mentioned above, and the result $(C3)$, it is easy to show (by using induction) that the following holds:

$$\mathit{firstst}(\tau^p, \chi^{-1}(i_2))(v') = \mathit{firstst}(\tau, i_2)(v)$$ ∎

The following theorem finally shows the *modulo-index equivalence* of the two traces for the relevant trace-points.

Theorem 9.11 *For all locations (l, i) of the program $\mathcal{P}^r$, the execution state $((l, \chi^{-1}(i)), \sigma^p)_{\tau^p}$ exists and is modulo-index equivalent to the execution state $((l, i), \sigma)_\tau$.*

Proof We will use induction on j for the sequence of locations (l_j, i_j) of the program $\mathcal{P}^r$ with $0 \le j \le |\mathcal{P}^r|$, where $|\mathcal{P}^r|$ denotes the number of statements in $\mathcal{P}^r$.

Base step: $j = 0$. Since $l_0 = \mathit{ENTRY}$, $i_0 = \perp$, and $((l_0, \chi^{-1}(i_0)), \sigma^p_0)_{\tau^p} \preceq ((l_0, i_0), \sigma_0)_\tau$, the

statement holds for (l_0, i_0).

Induction step: Let $\forall j \in [0, k]$. $((l_j, \chi^{-1}(i_j)), \sigma^p_j)_{\tau^p} \preceq ((l_j, i_j), \sigma_j)_\tau$ be true for $k < |\mathcal{P}^r|$. Consider the location (l_{k+1}, i_{k+1}). Let execution states for (l_k, i_k) be $\omega_k \equiv ((l_k, i_k), \sigma_k)_\tau$ and $\omega'_k \equiv ((l_k, \chi^{-1}(i_k)), \sigma^p_k)_{\tau^p}$. Let $\omega_{k+1} \equiv ((l_{k+1}, i_{k+1}), \sigma_{k+1})_\tau$. We need to show that the execution state $\omega'_{k+1} \equiv ((l_{k+1}, \chi^{-1}(i_{k+1})), \sigma^p_{k+1})_{\tau^p}$ exists, and $\omega'_{k+1} \preceq \omega_{k+1}$. We consider different possible transitions to the location (l_{k+1}, i_{k+1}) from the location (l_k, i_k), depending on the statement at l_k, which are as follows:

1. Initialisation of a loop counter variable, assert statement, assignment to a variable which is not a loop counter variable, and condition that uses variables only from $RLV(l_k, i_k)$.
2. self-controlling condition $v < e$ with $v \notin RLV(l_k, i_k)$.
3. Loop exit condition check $\ell < \mathsf{c}_{ub}$, and increment of a loop counter variable $\ell = \ell + c$.

As we had observed earlier, for an expression e, if $vars(e) \in RLV(l_k, i_k)$ then $[\![e']\!]_{\omega'_k} = [\![e]\!]_{\omega_k}$. Therefore, the conditions in category (1) will evaluate to same value in ω'_k and ω_k. And, since other statements in category (1) are unconditional, in all the cases of the category (1), ω'_{k+1} will exist and will be next to ω'_k. It is straightforward to show that $\omega'_{k+1} \preceq \omega_{k+1}$, for statements of category (1). So, we illustrate only the case of assignment from this category, and consider the remaining categories individually.

(1) An assignment of the form $v = e$ — The corresponding assignment in τ^p would be $v' = e'$. If $v \notin RLV(l_{k+1}, i_{k+1})$, then $\omega'_{k+1} \preceq \omega_{k+1}$ holds trivially. If $v \in RLV(l_{k+1}, i_{k+1})$, then $vars(e) \subseteq RLV(l_k, i_k)$ and $[\![e']\!]_{\omega'_k} = [\![e]\!]_{\omega_k}$. So, $\omega'_{k+1}(v') = \omega_{k+1}(v)$. Therefore, $\omega'_{k+1} \preceq \omega_{k+1}$.

(2) A self-controlling condition, say $v < e$, with $v \notin RLV(l_k, i_k)$ — Obviously, $vars(e) \subseteq RLV(l_k, i_k)$, and therefore $[\![e]\!]_{\omega_k} = [\![e]\!]_{\omega'_k}$. The corresponding condition in τ^p would be $v < e'$. Obviously, $v \notin RLV(l_{k+1}, i_{k+1})$ and $RLV(l_{k+1}, i_{k+1}) \subseteq RLV(l_k, i_k)$. As per $(C2)$, $\omega'_k(v) \leq \omega_k(v)$. If $[\![v < e]\!]_{\omega_k}$ is true, so would be $[\![v < e']\!]_{\omega'_k}$, and therefore ω'_{k+1} exists and is next to ω'_k in τ^p. Trivially, $\omega'_{k+1} \preceq \omega_{k+1}$. If $[\![v < e]\!]_{\omega_k}$ is false, l_{k+1} must be the immediate post dominator of l_k. So, again, ω'_{k+1} exists, and there would be no assignments to any other variable in between $(l_k, \chi^{-1}(i_k))_{\tau^p}$ and $(l_{k+1}, \chi^{-1}(i_{k+1}))_{\tau^p}$. Therefore, $\omega'_{k+1} \preceq \omega_{k+1}$.

(3) A loop exit check in the form $\ell < \mathsf{c}_{ub}$ — The condition is equivalent to $i_k \leq \mathsf{c}_{final}$ in ω_k and $\chi^{-1}(i_k) \leq \mathsf{c}'_{final}$ in ω'_k. Since $[\![\chi^{-1}(i_k) \leq \mathsf{c}'_{final}]\!]_{\omega'_k} = [\![i_k \leq \mathsf{c}_{final}]\!]_{\omega_k}$, the execution state ω'_{k+1} exists and is next to ω'_k in τ^p. If the condition is true then $\omega'_{k+1} \preceq \omega_{k+1}$, trivially.

If the condition is false, it means that the control exits the loop with l_{k+1} as $exit$ of the loop, $i_{k+1} = \bot$, $\sigma_k = laststs(\tau, \mathsf{c}_{final})$ and $\sigma^p_k = laststs(\tau^p, \mathsf{c}'_{final})$. As per the definition of $RLV(l, i)$, $RLV(l_{k+1}, i_{k+1}) \supseteq RLV(l_k, i_k)$ and $RLV(l_{k+1}, i_{k+1}) \setminus RLV(l_k, i_k) \subseteq \mathbb{V}_{sc}(l_k)$. Obviously, for all $v \in RLV(l_{k+1}, i_{k+1}) \cap RLV(l_k, i_k)$, $\omega'_{k+1}(v') = \omega_{k+1}(v)$. Let $v \in RLV(l_{k+1}, i_{k+1}) \cap \mathbb{V}_{sc}(l_k)$. Obviously, v is a self-controlled variable and therefore, for the trace-point $(entry, \bot)$ of the loop $v \in RLV(entry, \bot)$. We observe that there must exist $j, 0 \leq j \leq k$, such that the trace-point $(entry, \bot)$ is same as (l_j, i_j). Therefore, by induction hypothesis value of v will be same in the beginning of the loop in τ and τ^p. So, as per Lemma 9.9 $laststs(\tau, \mathsf{c}_{final})(v) = laststs(\tau^p, \mathsf{c}'_{final})(v)$. Therefore, $\omega'_{k+1} \preceq \omega_{k+1}$.

(4) Increment of a loop counter variable $\ell = \ell + c$ — So, $i_{k+1} = i_k + hc$, for some $h > 0$, with i_k and i_{k+1} both being successive LVA of the loop. Therefore, ω'_{k+1} exists, and ω_k, ω'_k, ω_{k+1} and ω'_{k+1} must be $laststs(\tau, i_k)$, $laststs(\tau^p, \chi^{-1}(i_k))$, $firststs(\tau, i_{k+1})$ and $firststs(\tau^p, \chi^{-1}(i_{k+1}))$, respectively. And as per Lemma 9.10, $\omega'_{k+1} \preceq \omega_{k+1}$. ∎

Recall that showing $((l_a, i_a'), \sigma^p)_{\tau^p} \preceq ((l_a, i_a), \sigma)_\tau$ is sufficient to prove Claim 9.1. Since (l_a, i_a) is a location in $\mathcal{P}^r$, using Theorem 9.11 the claim is proved.

9.7 Implementation and measurements

We have explained that for a program amenable to loop pruning approach, we can compute a bound on number of iterations required to reproduce the values of modified variables and array elements (*modulo-index equivalence*). Checking if a program is as per the grammar given in Section 9.2.1 is straightforward. To check the semantic constraints, we make use of PDG of the program from where the loop dependence graph for a loop can be inferred. To construct the PDG, we identify data dependence using reaching definition analysis available from PRISM[10], and to identify control dependence, we reuse the implementation of the algorithm of Billardi and Pingali [11], available from the work on *value slice*. While building the PDG for the program, we compute the synchronising distance (Θ) as $\mathbf{lcm}$ of the loop steps. We compute c_{final} for each loop, from which the values of $\widehat{\mathbb{N}}$ and $\overline{\mathbb{N}}$ are computed. We find the *loop counter bound for constant indices*, $\mathcal{K}^c$, also in the process. We annotate the span for the use nodes of

[10] A static analyzer generator developed at TRDDC, Pune [53, 20].

the PDG using the algorithm given in Section 9.3.3. We check the semantic constraints (1) and (2), given in Section 9.3, using the PDG annotated with computed spans. From the computed span, we compute the value of safe distance for non-interference (δ) for the program.

We compute the variable dependence graphs (VDG) using the PDG, and annotate the nodes of VDG with the *instance-count* of variables, as per the approach given in Section 9.5.5. The bound $\mathcal{C}$ on #LVA of the program is computed as per the process given in Section 9.5.7. Using $\mathcal{C}$, we compute $\Gamma' = \mathcal{K}^c + (\mathcal{C}+1) \times (\delta + \Theta) + (\widehat{\mathbb{N}} - \overline{\mathbb{N}})$. Finally, we compute $\Gamma = \Gamma' + (\Theta - (\widehat{\mathbb{N}} - \Gamma')\%\Theta)\%\Theta$. The bound $\mathsf{c}'_{\mathit{final}}$ for individual loops is computed as $\mathsf{c}'_{\mathit{final}} = \Gamma - (\widehat{\mathbb{N}} - \mathsf{c}_{\mathit{final}})$. Using the bound for the individual loops of the program P, we produce a new program P', by replacing all the loops with corresponding pruned loops in which every loop iterates only up to the lcv bounded by $\mathsf{c}'_{\mathit{final}}$ of the loop. Since no change to the loop body is needed, this abstraction is straightforward. We use a bounded model checker to check the property on the pruned program P'.

9.7.1 Implementation

The proposed loop pruning approach has been implemented in the tool *VeriAbs* [19]. Within the scope of the programs for which loop pruning approach is suitable, the tool supports all C constructs including pointers, structure, arrays, heaps and non-recursive function calls. Internally, the tool uses the bounded model checker CBMC 5.4 [22], to check the property on the pruned (abstracted) program. If the verification of the pruned program succeeds, the tool declares the original program to be safe with respect to the given property.

If the programs have no conditions in the loop body then no variable, except the loop counter variables, has a loop-carried dependence. In such cases, since the property checked is universally quantified, if the verification of the abstract program fails then it is guaranteed that original program is also incorrect, and the tool declares the program to be incorrect. Even when there are conditions in the loop body, but only of self-controlling type, then also it is guaranteed that the original program is incorrect when the verification of the abstract program fails. In other situations, if the verification of the abstract program fails, the tool indicates its inability to decide the verification status of the program.

Our implementation allows two additional code patterns where the loop pruning approach

```
1 min1 =a[0]; min2 =a[1];
2 if (min1 > a[1]) { min2 = min1; min1 = a[1]; }
3 for (i=2; i< N; i++) {
4   if (min1 > a[i]) { min2 = min1; min1 = a[i]; }
5   if (min2 > a[i]) if (min1 < a[i]) min2 = a[i];
6 }
7 assert (min1<=min2);
```

(a) Second minimum computation

```
1 sum1=0;
2 for (i=0; i < N; i++)
3   sum1=sum1+a[i];
4 sum2=a[0];
5 for (i=1; i < N; i++)
6   sum2=sum2+a[i];
7 assert(sum1==sum2);
```

(b) Sum computation

Figure 9.12: Additional code patterns where loop pruning works

is guaranteed to work, even though they do not satisfy the constraints given in Section 9.3. These are given in Figure 9.12. In the code snippet of Figure 9.12(a), a self-controlled variable `min2` is dependent on another self-controlled variable `min1`. But the comparison operator and expression being compared and assigned are the same for both the variables. The argument presented earlier for *modulo-index equivalence* can be extended to such patterns also. In the code snippet of Figure 9.12(b), the variables `sum1` and `sum2` have a self cyclic data dependence. But the array on which they depend is not getting modified and no other variable depends on these array elements. In this case also, an initial state can be constructed so that we get same values of the variables `sum1` and `sum2` in the pruned program. However, the construction will not be similar to what we have used in our discussion on the proof of soundness. An extension of our approach to formalise and generalise these cases can be an interesting future work.

9.7.2 Experiments

The tool *VeriAbs*, equipped with implementation of the loop pruning as well as loop shrinking approaches, participated in the *ArraysReach* category of SV-COMP 2018 verification competition [9]. There were a total of 167 programs in the category, 123 of which are correct (safe) and the remaining 44 are incorrect (unsafe) with respect to their properties. Out of the total 167 programs, there were 48 programs that satisfied the constraints defined in sections 9.3 and 9.2.1. The loop pruning approach could verify all these 48 programs. In fact, out of these 48 programs there were 23 programs for which only the loop pruning approach worked, because these programs were not amenable to the loop shrinking approach. This shows that the approach has its

own usefulness and applicability.

In Table 9.1(a), we show different parameters, and theoretical and computed bounds on $\#$LVA of the programs. In Table 9.1(b), we show the results delivered by the tool. To compare the computed $\#$LVA of the program with the theoretical bound, we found the total number of non-leaf variables in the integrated VDG (denoted as $\hat{M}$), and the maximum degree of dependency of these variables (denoted as $\hat{k}$), and computed the theoretical upper bound on $\#$LVA, which is $(\hat{M} - \hat{k} + 2)2^{\hat{k}-1}$. In Table 9.1(a), we have shown the distribution of those 48 programs on the basis of values of δ, $\hat{k}$, $(\hat{M} - \hat{k})$, number of loops that modify some variable relevant to the property, synchronising distance (Θ), the theoretical bound on $\#$LVA, the bound $\mathcal{C}$ on $\#$LVA as explained in Section 9.5.5, and the overall bound Γ. As expected, when values of $\hat{k}$ and $\hat{M} - \hat{k}$ are small, there is no significant difference in the computed $\#$LVA and the theoretical bound on $\#$LVA of the programs. However, when $\hat{k}$ is higher than 2 then the difference is much more, and after a point the theoretical bound becomes quite huge since it is exponential in $\hat{k}$. However, computed bound on $\#$LVA continues to remain a small number. Since the loop step count was 1 in each program, the maximum number of iterations can go only up to Γ in the pruned program. While the original programs had loops of size more than 100000, the data shows that pruned programs had lesser than 15 iterations in every case. In Table 9.1(c), we show the number of programs that were verified to be safe or shown to be incorrect by the two approaches: loop pruning (LP) and loop shrinking (LSH). The column "LP Only" shows the number of programs that could be verified or falsified by only loop pruning and not by loop shrinking. There were 23 such programs. Similarly, the column "LSH Only" shows the number of programs that could be verified or falsified by only loop shrinking and not by loop pruning. The column "LP & LSH Both" shows the number of programs that could be verified or falsified by both loop pruning as well as loop shrinking.

Like the loop shrinking approach, loop pruning also has the interesting property that while it is limited by its ability to deal only with programs which satisfy the constraints imposed, once a program is found to be amenable to the approach, the method is impervious to the size of arrays—increasing the size does not cause an otherwise verifiable program to timeout.

Table 9.1: Experimental results for SV-COMP 2018 ArraysReach benchmarks

(a) Parameter value wise program distribution

δ	$\hat{k}$	$\hat{M} - \hat{k}$	#Computing loops	Θ	Theoretical #LVA	Computed #LVA	Γ	Number of programs		
								Correct	Incorrect	Total
0	2	0	1	1	4	1	3	9	1	10
0	3	1	1	1	12	3	4	1	0	1
1	3	5	4	1	28	4	12	18	0	18
1	2	0	1	1	4	1	6	1	0	1
0	4-10	1	2-5	1	24-1536	3-6	5-8	4	4	8
0	12-20	1	6-10	1	6144 - *	7-11	9-13	5	5	10
Total								38	10	48

(b) Programs categories

Programs	Correct	Incorrect	Total
Not-amenable to loop pruning	85	34	119
Amenable to loop pruning	38	10	48
Total	123	44	167

(c) Property verification results

Programs	LP Only	LSH Only	LP & LSH Both
Declared correct	22	65	16
Declared incorrect	1	17	9
Total	23	82	25

Chapter 10

Related work

The various approaches to handle arrays have their roots in the types of static analyses used for property verification, for example: abstract interpretation, predicate abstraction, bounded model checking and theorem proving.

In abstract interpretation, arrays are handled using *array smashing*, *array expansion* and *array slicing*. In array smashing, all the elements of an array are clubbed as a single anonymous element, with writes to the array elements treated as weak updates. As a result, the abstraction becomes too coarse leading to imprecision. It cannot be used, for example, to verify the motivating example of Figure 1.3(a). In array expansion, array elements are explicated as a collection of scalar variables, and the resulting programs have fewer number of weak updates than array smashing. However, it works well only for small-sized arrays. A mix of smashing and expansion has been used by Blanchet et al. [13, 14] to prove that a program does not perform executions with undefined behaviour such as out-of-bound array accesses. In array slicing, the idea is to track partitions of arrays based upon some criteria inferred from programs [41, 42, 28]. Each partition is treated as an independent smashed element. Dillig et al. [33] further refined this approach to introduce the notion of *fluid updates*, where a write operation may result in a strong update of one partition of the array, and a weak update of the other partitions. In contrast to these approaches, our abstraction is based not only on the program's data elements, but also on the property being verified. By declaring an array-processing loop as k-shrinkable, we guarantee that an erroneous behaviour of the program with respect to the property can indeed be replayed on some k elements of the array.

There has been considerable work in over-approximation and under-approximation of the loops for verification and bug finding, respectively. Daniel et al. [55] under-approximate loops using acceleration to find deep bugs in the programs. Halbwachs et al. [40] and Schrammel et al. [66] also abstract loops using abstract acceleration. Darke et al. [29] abstract loops and apply acceleration to verify property. Based on the number of input, output and I/O variables of the loop, and the number of control paths in the loop body, they compute a bound for the abstract loop which is sound to verify the property. However, these efforts have been restricted to loops that compute only scalar variables. For example, Darke et al., in their work on loop abstraction [29], consider only those loops that do not contain operations on array elements.

Methods based on predicate abstraction go through several rounds of counterexample guided abstraction refinement (CEGAR). In each round a suitable loop invariant is searched based on the counterexample using *Craig interpolants* [59]. Tools like SATABS [23] and CPAchecker [10] are based on this technique. To handle arrays, the approach relies on finding appropriate quantified loop invariants. However, generating interpolants for scalar programs is, by itself, a hard problem. With the inclusion of arrays, which requires universally quantified interpolants, the problem becomes even harder [51, 60]. Our method, in contrast, does not rely on the ability to find loop invariant. Instead, we find a bound on the number of loop iterations and, in turn, the number of array elements that have to be accessed in a run of abstract program.

Qadeer et al. [37] infer higher-level loop invariants using predicate abstraction, for a given set of predicates and some lower-level loop invariants. To address the need of quantified loop invariant in presence of arrays, they allow some special variables, called *skolem constants*, to be used in the given set of predicates and loop invariants. After inferring the loop invariant using their method, authors produce quantified loop invariant by universally quantifying the skolem constants used. Typically, these skolem constants would be used in array index expressions. Although, authors have given some heuristics to automatically identify the required predicates and skolem constants, but their examples show that the efficacy of their approach relies heavily on the set of predicates and skolem constants provided as input.

Jhala et al. [51], in their work on "Array Abstraction from Proofs" infer *range predicates*, rather than inferring the loop invariants. Range predicates describe a property over some unbounded array segments. In this technique, infeasible paths generated by some unrolling of the loops, are used to automatically infer the range predicates. These range predicates are then

used to verify the target property. Using some informal axioms, and an interpolant discovered from infeasible path constraints, the authors infer the range predicates. During this inference process, authors apply rules to shrink/expand and join the inferred range predicates to arrive at a smaller number of desired range predicates. There are two basic challenges: (1) identifying the base property over array elements data values, for which range predicates are to be inferred, and (2) inferring range predicates involving a large number of disjoint array segments, which may require large number of unrolling of the loops, in different refinement cycles. In contrast, our methods do not rely on discovering any predicate or property over the array elements.

In the work of Alberti et al. [60], a given program with arrays is transformed into a program with only scalar variables. The array is abstracted by a few elements with as many index variables, and the loops remain as is in the transformed program. To verify the translated programs, the authors rely on back-end verifiers which may be predicate abstraction based or acceleration based. The basic challenge here is to know the number of elements with which the array should be abstracted such that it is sufficient to prove the property. Authors have not given any clue on this issue. Moreover, since post conversion the loops remain unchanged, the challenges posed by loops carry over to the converted program also. In contrast, our method of loop shrinking finds out the sufficient k automatically, and the loops produced in the abstract program have very small bound.

Theorem proving based methods generate a set of constraints, typically Horn clauses that relate invariants at various program points. The invariants are predicates over arrays. The constraints are then fed to a solver in order to find a model. However, these methods also face the same difficulty of synthesizing quantified loop invariant over arrays. The technique of *k-distinguished cell abstraction* addresses this problem by abstracting the array to only k elements. A 1-distinguished cell abstraction, for example, abstracts a predicate $P(a)$ involving an array a by $P'(i, a_i)$, where i and a_i are scalars. The relation between the two predicates is that $P'(i, a_i)$ holds whenever $P(a)$ holds, and the value of $a[i]$ is a_i. The resulting constraints are easier to solve using a back-end solver such as Z3 [32]. This technique, which is a refinement of work by Alberti et al. [60], appears in the work by Gonnord et al. [61]. We experimented with VAPHOR, a tool based on the work by Gonnord et al. By way of comparison, we present two examples, one with a $\exists$ property and the other with a $\forall$ property. The first program computes the minimum of an array and asserts that the minimum is the same as the value of some element in

the array. The second program copies all but 1 elements from one large array to another. It then asserts that the copied elements are pairwise equal. While our tool could verify both examples, VAPHOR declared the first program to be incorrect with 1-and 2-distinguished cell abstraction and timed out on the second program.

A method that is properly subsumed by our method is by Anushri et al. [49]. This uses only one distinguished element, called a *witness element*, and transforms a program with arrays and loops to a loop free scalar program. This program is then model-checked using a BMC. This approach works well on what authors call *full array processing loops* and such loops are a proper subset of our 1-shrinkable loops.

Regarding our work on loop pruning, to the best of our knowledge, there is no work done on just pruning the loop bound without abstracting the loop body. As mentioned earlier, all of the work dealing with loops and arrays abstract the loop body in one or the other form.

Inductive reasoning also has been used to verify programs where property checked can be sliced and each slice can be related to a generic iteration of the array processing loop. Work by Chakraborty et al. [18] is one example of use of such inductive reasoning. A range of index of array elements that are processed by a single iteration of the loop and which cover the array elements accessed in the property slice corresponding to the iteration, is called a *tile*. The tiles so identified should collectively cover the entire array index range. By an inductive reasoning, authors check that a generic iteration of the loop satisfies the corresponding sliced property and this behaviour remains intact when subsequent iterations of the loop are executed. However, since inductive reasoning requires some loop invariant to work with, they also rely on some loop invariant (may be weak). They obtain these in their implementation using a dynamic analysis tool Daikon [34], which provides likely loop invariant. Identifying the tiles themselves is based on certain heuristics. In contrast, our technique of loop pruning does not require any loop invariant, and requires no relationship between the property checking loop and the computing loop.

Part III

Concluding Remarks

Chapter 11

Conclusion and future directions

There are two common reasons why property checking tools fail on real life programs. Both relate to the complexity of the program being property checked. The first is that the size of the program may be large, and the second is that it may have loops with large bounds. In this thesis, we have presented two approaches for scaling up property checking: (1) by reducing the size of program through a slicing technique called *value slicing*, which is more aggressive than traditional backward slicing without losing much in way of precision, and, (2) by reducing the loop bounds of large array processing programs using two different techniques—loop shrinking and loop pruning. Experimental results have shown that, wherever applicable, the approaches do enable property checking to scale up to a large extent.

11.1 Scaling up property checking through value slice

Slicing is an obvious pre-processing step before submitting a program to a verifier for property checking. Backward slice has been a natural choice for this so far, since the behaviour of the sliced program exactly matches the behaviour of the original program with respect to the property being checked[1]. In our approach, we have suggested a more aggressive form of slicing called value slice. This technique slices out statements that only affect the reachability of the assertion point and retains just those statements which influence the values of the property variables. As a result, the value-sliced program is smaller in size compared to backward slicing. Therefore, property checking with value slice is more scalable than backward slice. On the other

[1] Assuming the program is terminating.

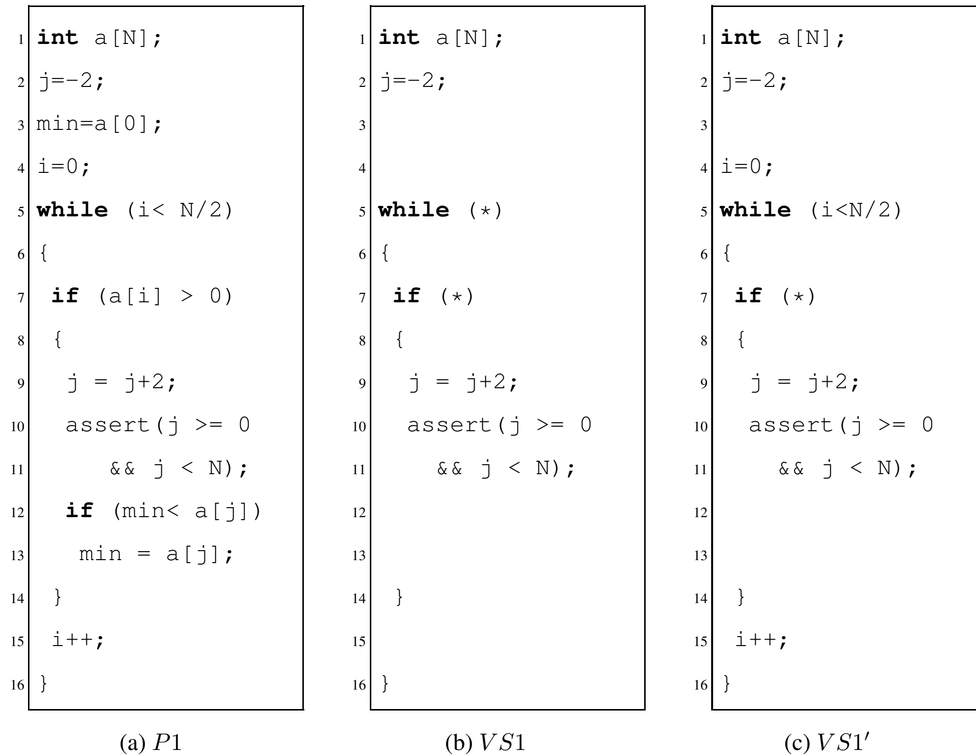
hand, since our method carefully identifies and retains certain predicates, property checking with value slice is more precise than an even more aggressive form of slicing called thin slice. Indeed, our experiments show that on both axes of comparison, scalability and precision, value slice based property checking comes close to the best performer of the three comparable forms of slicing.

In the sequel of this section, we shall take up each of the methods and point out the limitations of our investigation. This will suggest natural directions in which the work in the thesis could be extended.

11.1.1 Minimal value slice and other aggressive slices

Our development of value slicing starts with a trace based specification of the value slice that uses the notion of SC-equivalence of sub-traces (Definition 3.1). Given a program to be sliced, we compute a subprogram P^{VS} based on data and control dependence (Definition 3.2) and show that P^{VS} indeed satisfies the trace based specification of value slice. However, we have not shown that the slice thus computed is always a minimal subprogram satisfying the specification. Therefore, we still have the following open questions: Is the slice computed always minimal? If not, is the minimal value slice computable? If not, is there a method to compute a value slice that is smaller (in general, no larger) than the value slice computed by our method?

There are cases where the value slice is not very effective. Consider the program $P1$ and its value slice $VS1$ in Figure 11.1. Program $P1$ is correct with respect to the property that is being checked. However, since the variable `j` becomes unconstrained in the value slice $VS1$, the property no longer holds and we will get a counterexample for the same. It will be better if a slice $VS1'$ can be produced instead, where retaining the loop exit condition makes the sliced program remain correct. Note that $VS1'$ is still an aggressive slice compared to the backward slice. A different issue arises when the same property is encoded as an error state (bad state) that should be unreachable, as illustrated in program $P2$ of Figure 11.2. Note that the value slice for this program, shown as $VS2$ in the figure, is the same as its thin slice. This is because the assert expression has no variables used and therefore, as per the formulation, there will be no value impacting statements. We can of course transform the original program $P2$ so that it becomes like $P1$ and do the value slicing on transformed program to get a slice like $VS1$. However, in



(a) $P1$ (b) $VS1$ (c) $VS1'$

Figure 11.1: Illustration of limitation of value slice

this case a desired slice would be $VS2'$, where property is still encoded as an unreachable error state and the property also holds in the slice. We believe that if we can get the slice $VS1'$ for $P1$, it would be possible to get $VS2'$ also directly (without any transformation) from the program $P2$. The examples demonstrate that a different kind of slice, where some conditions, but not all, which impact only reachability of the property are also retained, can be more effective than value slice for certain programs. Exploration and formalisation of such slices, and methods to compute them, can be a useful extension of our work.

11.1.2 Effectiveness of the method for different verifiers

As we have discussed in the experimental results, our results are based on a particular verifier, namely SATABS. We believe that similar trends will also show up when other verifiers are used. However, it will be interesting to see the actual trends by experimenting with other verifiers, especially those based on different techniques. One of the question to explore could be: What kind of verification techniques can best exploit value slices.

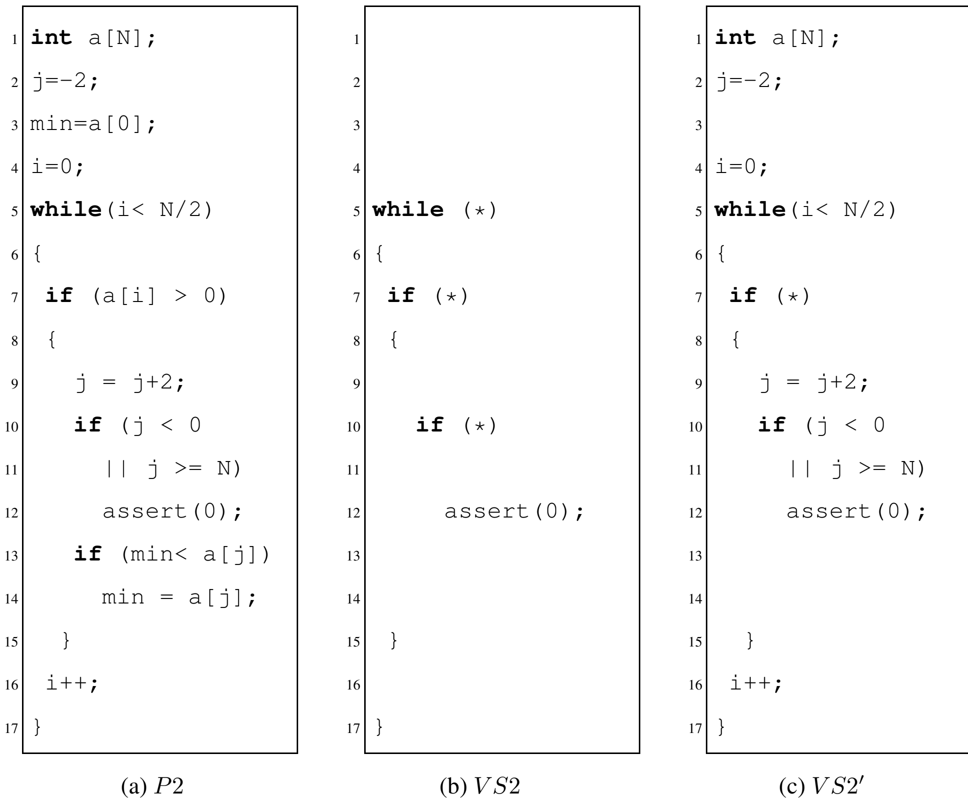


Figure 11.2: Illustration of limitation of value slice when property encoded as unreachable error state

11.1.3 Refinement

Since value slice is a sound but not a precise abstraction of the input program with respect to the property being checked, a verifier may declare that the property does not hold for a correct program and produce a spurious counterexample. This happens because our method abstracts certain conditional expressions of the program. Because of this, the verifier finds a path that leads to the violation of the property, while such a path may not exist in any run of the program in reality. We could use any of the existing counterexample guided techniques to track the abstracted conditional expressions that cause the spurious counterexample, and refine these expressions. In the process, we may face a situation where multiple abstracted conditions are identified as a cause of a spurious counterexample. A prioritisation strategy to choose the condition expressions to refine may be needed to retain the scalability of our technique; refining all the conditions at once may compromise that.

11.1.4 Other uses of value slice

Slicing is very commonly used for program understanding and debugging, mainly due to the reduction in size of the resulting sliced program. The reduced size helps programmers to focus only on relevant part of the code for a given purpose. In fact, the primary use of thin slice is debugging.

As our experiments show, SATABS timed out for 50% less programs when used to property check a value slice than it did when used on a backward slice. On the other hand property checking using value slice produced 10 times lesser number of incorrect results (declaring a program incorrect when it is actually correct) than the number of incorrect results produced using thin slice. It shows that value slice reduces the program size and complexity much more than the backward slicing, and at the same time, retains much of the useful information with respect to the slicing criterion than thin slice. Therefore, it is worthwhile to use value slice for program understanding as well as debugging.

11.2 Reducing the size of the loops

We have proposed two techniques to abstract programs so that the abstracted program have much smaller loop bounds compared to the large bounds of the original programs. The programs use these loops to process large arrays. Whenever applicable, the techniques are fully automatic.

Our first technique, called loop shrinking, enables us to verify properties over loops that have large or even unknown (but with a statically over-approximable) bound by converting them to loops with a small bound. Towards this, we have defined a notion called shrinkability of a loop, and showed that arrays processed by k-shrinkable loops can be abstracted using only k elements. The abstracted program can then be checked using a bounded model checker as back-end. An important contribution of our method is an automated technique to find out the required bound k. Although there are approaches that are based on abstracting an array by fewer elements, none of these provide a way to find out the number of elements that are sufficient to reason about the array. Our experiments have shown that the approach is powerful enough to handle a variety of array processing programs. Depending on the sophistication of identifying loop accelerable variables, and the corresponding accelerated expressions, the technique can

handle nested loops after they are flattened as a single loop. The technique addresses only certain classes of cascaded multiple loops.

Our second technique also enables scaling up property checking for a class of large array processing programs. While loop shrinking uses an empirical way to find the bound on the shrunk loop, in this technique we find this bound using a static analysis. In addition, once a program satisfies the constraints under which the technique works, occurrence of multiple loops in the program poses no further challenges. However, nested loops still remain a limitation. We introduced a notion of last value assignment and variable dependence graph, using which we compute a loop specific upper bound for all the loops. Using the loop specific bound, we abstract the program in which we leave the loop body unchanged and only modify the loop bound. The experiments have shown that there are enough programs that benefit from the technique, while other techniques fail to verify such programs.

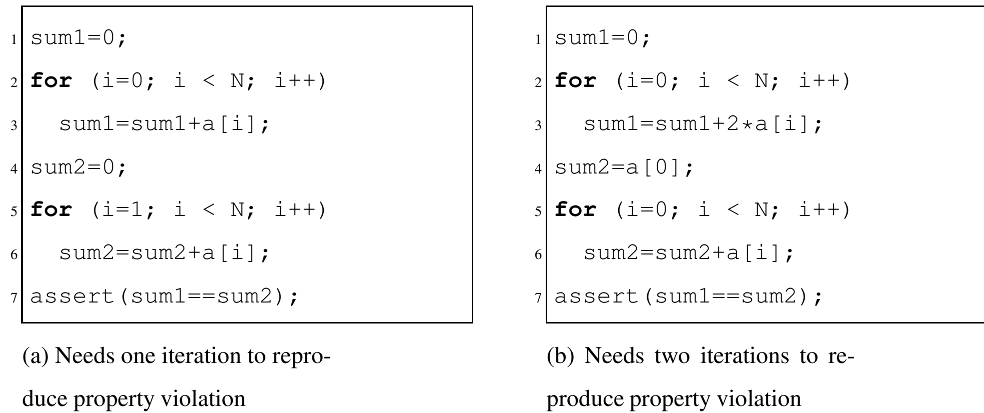
11.2.1 Nested loops and multiple loops

At present, these two techniques do not support nested loops. One possible future work is to extend these to support nested loops. For loop shrinking, it may involve extending the notion of shrinkability to nested loops. In the case of loop pruning, it may involve extending the notion of last value assigning lcv to nested loops. lcv can be modeled as a tuple of lcv of the loops in the nesting.

Although loop pruning has no limitation for multiple loops, the loop shrinking technique in its present form relies on whether multiple loops of a program can be coalesced into one. A future research direction can be to support multiple loops as they are (i.e., without transforming them into a single loop). One way could be to identify auxiliary properties for every loop separately, based on the original property and some loop invariant of the loop. In effect, multiple problems can be spawned off, each of which is solvable by the current loop shrinking method.

11.2.2 Refinement

In both the approaches, the tool declares the property as verified if it is verified on the abstracted program. But when the property is violated on the abstracted program, only in a few cases does the tool declare the original program to be incorrect too. In other cases, the tool remains



(a) Needs one iteration to reproduce property violation

(b) Needs two iterations to reproduce property violation

Figure 11.3: Illustration of property outcome reproduction based loop pruning

undecided. However, for the loop pruning approach, due to restrictions that it imposes on the input programs, we believe that the method is complete as well. Although we have not shown or proved it in this thesis, it would be an interesting future work to argue the completeness of this method.

In case of loop shrinking approach, an end-to-end property checking process is desired with the following two mechanisms: (a) to check if the failure of the property indicates incorrectness of the original program, and in the cases where it does not (b) to advance the property checking process further. Adding these mechanisms to the technique can be a direction for future work. In case of loop shrinking, we can play the input leading to failure of abstract program on the original program, and check if the failure is replicated. If yes, then we are done; else, we can try a possible refinement step by increasing the k for the abstraction. Recall that once a loop is k-shrinkable it is k'-shrinkable also for all $k' > k$. For the loop pruning approach, the challenge would be to know if the failure of property on pruned program is spurious for the original program.

11.2.3 Loop pruning based on property rather than operand values

The second technique addresses processing arrays in only one direction. It can be extended to address programs which process arrays by traversing it in both the directions. Loop pruning technique's basic premise is that an abstracted program with much smaller loop bounds reproduces the values of the variables in the property checking assertion of the original program. This is perhaps an unnecessarily strong requirement. Instead, we can extend our approach to

ensure that the pruned program reproduces value of the property rather than the value of its individual operands. That is, if the property is violated in the original program, the pruned program will also have an execution in which the property is guaranteed to be violated. This extension to the approach should remove several of the restrictions imposed on programs on which our technique is applicable in its current form. For example, consider two programs shown in Figure 11.3. Both the programs are not allowed as per our constraints because the variables `sum1` and `sum2` have cyclic data dependence. For both the programs, we observe that it would be difficult to reproduce a violation of the property in a pruned program with the values of `sum1` and `sum2` being same as they were in the original program's execution exhibiting the violation. However, we observe that a violation of the property in program (a) can be replicated in a pruned program with the pruned loop having only one iteration. But in case of program (b) the replication of violation of the property will be possible in a pruned program only if the pruned loop has at least two iterations. So, although both programs are similar in terms of syntactic structure and inter-variable dependence, their corresponding pruned programs require different number of iterations in the pruned loop to replicate the violation of the property. Therefore, for such programs one challenge would be to identify the number of iterations needed, i.e. the bound on the individual loops, to guarantee the reproduction of violation of the property if there is a violation in the original program. In spite of the challenges, we believe that problem is solvable, and worth looking at.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson Education, Inc., 2006.

[2] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Definability of Accelerated Relations in a Theory of Arrays and its Applications. In *FroCos*, volume 8152 of *Lecture Notes in Computer Science*, pages 23–39. Springer, 2013.

[3] Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, October 1987.

[4] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A Decade of Software Model Checking with SLAM. *Commun. ACM*, 54:68–76, July 2011.

[5] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, 2002.

[6] Jose Bernardo Barros, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based Slicing and Slice Graphs. In *Proceedings of SEFM*, 2010.

[7] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and Data-flow Analysis of While-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, January 1985.

[8] Dirk Beyer. SV-COMP 2017 - 6th International Conference on Software Verification. `https://sv-comp.sosy-lab.org`, 2017.

[9] Dirk Beyer. SV-COMP 2018 - 7th International Conference on Software Verification. `https://sv-comp.sosy-lab.org`, 2018.

[10] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 184–190, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[11] Gianfranco Bilardi and Keshav Pingali. A Framework for Generalized Control Dependence. In *Proceedings of PLDI*, 1996.

[12] David W Binkley and Keith Brian Gallagher. Program Slicing. *Advances in Computers*, 43:1–50, 1996.

[13] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and Implementation of a Special-purpose Static Program Analyzer for Safety-critical Real-time Embedded Software. In *The Essence of Computation*, pages 85–108. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[14] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-critical Software. In *Proceedings of the PLDI, 2003*, pages 196–207, New York, NY, USA, 2003. ACM.

[15] Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konečný, and Tomáš Vojnar. Automatic Verification of Integer Array Programs. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 157–172, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[16] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned Program Slicing. *Information & Software Technology*, 40(11-12):595–607, 1998.

[17] Montgomery Carter, Shaobo He, Jonathan Whitaker, Zvonimir Rakamarić, and Michael Emmi. SMACK Software Verification Toolchain. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 589–592, New York, NY, USA, 2016. ACM.

[18] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Verifying Array Manipulating Programs by Tiling. In *International Static Analysis Symposium*, pages 428–449. Springer, 2017.

[19] Bharti Chimdyalwar, Priyanka Darke, Avriti Chauhan, Punit Shah, Shrawan Kumar, and R. Venkatesh. VeriAbs: Verification by Abstraction Competition Contribution. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*, pages 404–408, New York, NY, USA, 2017. Springer-Verlag New York, Inc.

[20] Bharti Chimdyalwar and Shrawan Kumar. Effective False Positive Filtering for Evolving Software. In *Proceedings of ISEC*, 2011.

[21] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *J. ACM*, 50(5):752–794, September 2003.

[22] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs . In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[23] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 570–574. Springer, 2005.

[24] Joseph J Comuzzi and Johnson M Hart. Program Slicing Using Weakest Preconditions. In *International Symposium of Formal Methods Europe*, pages 557–575. Springer, 1996.

[25] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[26] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77:*

Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 238–252, New York, NY, USA, 1977. ACM.

[27] Patrick Cousot, Radhia Cousot, Jerme Feret, Laurent Mauborgne, Antoine Min, David Monniaux, and Xavier Rival. The ASTREE Analyzer. In *Programming Languages and Systems, Proceedings of the 14th European Symposium on Programming, volume 3444 of Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[28] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. *SIGPLAN Not.*, 46(1):105–118, January 2011.

[29] Priyanka Darke, Bharti Chimdyalwar, R. Venkatesh, Ulka Shrotri, and Ravindra Metta. Over-approximating Loops to Prove Properties Using Bounded Model Checking. In *Proceedings of the DATE, 2015*, pages 1407–1412, San Jose, CA, USA, 2015. EDA Consortium.

[30] Priyanka Darke, Sumanth Prabhu, Bharti Chimdyalwar, Avriti Chauhan, Shrawan Kumar, Animesh Basakchowdhury, R Venkatesh, Advaita Datar, and Raveendra Kumar Medicherla. VeriAbs: Verification by Abstraction and Test Generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 457–462. Springer, 2018.

[31] Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 161–166. IEEE, 2008.

[32] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[33] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid Updates: Beyond Strong vs. Weak Updates. In *Programming Languages and Systems: 19th European Symposium on Program-*

ming, ESOP 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings, pages 246–266, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[34] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants, 2006.

[35] Stephan Falke, Florian Merz, and Carsten Sinz. LLBMC: Improved Bounded Model Checking of C Programs Using LLVM. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 623–626. Springer, 2013.

[36] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[37] Cormac Flanagan and Shaz Qadeer. Predicate Abstraction for Software Verification. In *Proceedings of the POPL, 2002*, pages 191–202, New York, NY, USA, 2002. ACM.

[38] Nicolas Gold and Mark Harman. An Empirical Study of Static Program Slice Size. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 16:2007, 2007.

[39] Laure Gonnord and Nicolas Halbwachs. Combining Widening and Acceleration in Linear Relation Analysis. In *International Static Analysis Symposium*, pages 144–160. Springer, 2006.

[40] Laure Gonnord and Nicolas Halbwachs. Combining Widening and Acceleration in Linear Relation Analysis. In *International Static Analysis Symposium*, pages 144–160. Springer, 2006.

[41] Denis Gopan, Thomas Reps, and Mooly Sagiv. A Framework for Numeric Analysis of Array Operations. *SIGPLAN Not.*, 40(1):338–350, January 2005.

[42] Nicolas Halbwachs and Mathias Péron. Discovering Properties About Arrays in Simple Programs. *SIGPLAN Not.*, 43(6):339–348, June 2008.

[43] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy Abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM.

[44] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969.

[45] Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Invariant Generation in Vampire. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, TACAS'11/ETAPS'11, pages 60–64, Berlin, Heidelberg, 2011. Springer-Verlag.

[46] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *SIGPLAN Not.*, 23:35–46, June 1988.

[47] T. Huckle. Collection of Software Bugs. `http://www5.in.tum.de/~huckle/bugse.html`.

[48] Daniel Jackson and Eugene J. Rollins. Chopping: A Generalization of Slicing. Technical report, Pittsburgh, PA, USA, 1994.

[49] Anushri Jana, Uday P. Khedker, Advaita Datar, R. Venkatesh, and Niyas C. Scaling Bounded Model Checking by Transforming Programs with Arrays. In *Logic-Based Program Synthesis and Transformation, 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016. Revised Selected Papers*, pages 275–292, Cham, 2017. Springer International Publishing.

[50] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. Abstract Acceleration of General Linear Loops. In *Proceedings of POPL, 2014*, pages 529–540, New York, NY, USA, 2014. ACM.

[51] Ranjit Jhala and Kenneth L. McMillan. Array Abstractions from Proofs. In *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings*, pages 193–206, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[52] Ken Kennedy. Use-definition Chains with Applications. *Computer Languages*, 3(3):163–179, 1978.

[53] Shubhangi Khare, Sandeep Saraswat, and Shrawan Kumar. Static Program Analysis of Large Embedded Code Base: An Experience. In *Proceedings of ISEC*, 2011.

[54] B. Korel and J. Laski. Dynamic Program Slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[55] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Under-approximating Loops in C Programs for Fast Counterexample Detection. *Formal methods in system design*, 47(1):75–92, 2015.

[56] Shrawan Kumar, Amitabha Sanyal, and Uday P. Khedker. Value Slice: A New Slicing Concept for Scalable Property Checking. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 101–115, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[57] Shrawan Kumar, Amitabha Sanyal, R. Venkatesh, and Punit Shah. Property Checking Array Programs Using Loop Shrinking. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 213–231, Cham, 2018. Springer International Publishing.

[58] Mathworks. Polyspace.
`http://www.mathworks.in/products/polyspace.html`.

[59] K. L. McMillan. Applications of Craig Interpolants in Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings*, pages 1–12, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[60] David Monniaux and Francesco Alberti. A Simple Abstraction of Arrays and Maps by Program Translation. In *Static Analysis: 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, pages 217–234, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[61] David Monniaux and Laure Gonnord. Cell Morphing: From Array Programs to Array-Free Horn Clauses. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 361–382, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[62] Jeremy Morse, Mikhail Ramalho, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. ESBMC 1.22. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 405–407. Springer, 2014.

[63] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2015.

[64] Andreas Podelski and Andrey Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *In PADL*. Springer, 2007.

[65] Sriram Sankaranarayanan, Michael A. Colón, Henny Sipma, and Zohar Manna. Efficient Strongly Relational Polyhedral Analysis. In *Proceedings of the 7th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 111–125, Berlin, Heidelberg, 2006. Springer-Verlag.

[66] Peter Schrammel and Bertrand Jeannet. Applying Abstract Acceleration to (co-) Reachability Analysis of Reactive Programs. *Journal of Symbolic Computation*, 47(12):1512–1532, 2012.

[67] Josep Silva. A Vocabulary of Program Slicing-based Techniques. *ACM Comput. Surv.*, 44(3):1–41, June 2012.

[68] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin Slicing. In *ACM SIGPLAN Notices*, volume 42, pages 112–122. ACM, 2007.

[69] Robert E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[70] Dexi Wang. Tool Ceagle. `http://sts.thss.tsinghua.edu.cn/ceagle`, 2017.

[71] Mark Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[72] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, Ann Arbor, MI, USA, 1979. AAI8007856.

[73] Wikipedia. List of Software Bugs.
http://en.wikipedia.org/wiki/List_of_software_bugs.

Acknowledgments

First of all, I thank my advisers Prof. Amitabha Sanyal and Prof. Uday Khedker, for their constant guidance and encouragement that helped me stay the course and shape my work. At personal level, they helped me immensely in solving non-academic problems, especially during the year of my stay at IIT Bombay. I am grateful to Prof. Supratim Biswas and Prof. Supratik Chakraborty, members of my Research Progress Committee, for their useful insights to improve the overall quality of my work. I would like to thank the entire CSE staff for providing the assistance and the required infrastructure. Special thanks to Mr. Vijay Ambre for taking care of the administrative procedures, especially my off-line registration in every semester. I acknowledge the help provided by Mohammad Afzal, who experimented the idea of loop pruning as part of his M.Sc. thesis internship under my guidance. I am thankful to my work organisation for allowing me to take up this journey, and facilitating to complete it. I sincerely thank my colleagues cum friends at TRDDC who helped me in various ways in completing this research work. To name a few, R. Venkatesh for helping shape up some of the ideas of this work, Ulka Shrotri for always raising my spirits whenever chips were down, Kumar Madhukar for immense help with reviewing and proof reading of the thesis manuscript. Finally, I am thankful to my family for the emotional support that helped me overcome the tough times during this journey.

Shrawan Kumar