Combining Slicing and Constraint Solving for Validation of Measurement Software

Gregor Snelting

Technische Universität Braunschweig
Abteilung Softwaretechnologie
Bültenweg 88, D-38106 Braunschweig

Abstract. We show how to combine program slicing and constraint solving in order to obtain better slice accuracy. The method is used in a program analysis tool for the validation of computer-controlled measurement systems. It will be used by the Physikalisch-Technische Bundesanstalt for verification of legally required calibration standards. The paper describes how to generate and simplify path conditions based on program slices. An example shows that the technique can indeed increase slice precision and reveal manipulations of the so-called calibration path.
Keywords: Program Slicing, Constraint Solving, Measurement System, Software Validation, Path Condition.

1 Introduction and Background

The Physikalisch-Technische-Bundesanstalt (PTB) is a national institution which – among other tasks – is responsible for the verification of calibration standards. Every measurement system which is used in medical, commercial, and similar transactions (e.g. an electricity meter or a blood alcohol tester) must stick to legally required standards for accuracy of measurement, robustness and other quality factors. Therefore, the prototype of every measurement system must be certified by PTB. Once the prototype has been thoroughly examined and validated, the numerous specimen of a specific measurement system are (less intensively) checked by local authorities.

Today, 95% of all measurement systems checked by PTB are controlled by software. Even the cheese scale has a built-in microprocessor and a digital display. Thus the validation of measurement software is part of the certification process. In particular, it must be checked that measurement values are not – accidentally or intentionally – manipulated or garbled. The most sensitive parts of measurement software are those which handle the incoming raw values and prepare it for display, while other parts like e.g. user interface control are less important. The data flow path from the sensor input port to the display output port is called the *calibration path* and is subject to most painstaking scrutiny.

At the moment measurement software validation at PTB relies on manual code inspections, which is a time-consuming and error-prone method. PTB is thus strongly interested in tool support for software validation. Therefore, PTB in

cooperation with the Technical University of Braunschweig launched a project which aims at a tool for analysis and semiautomatic validation of measurement software. The tool must analyse source code and support the PTB engineer by visualizing the obtained information. It must in particular check that there are no unwanted influences on the calibration path. If the calibration path is not safe, the tool must provide a detailed analysis of the conditions which can lead to a garbling of measurement values. Our ultimate goal is to automatically generate statements like the following: "If CTRL-X is pressed on the keyboard, and the left mouse key is pressed as well, then the measurement value is 8.7% too high".
It is the aim of this paper to describe the underlying technology of the new tool. Basically, the tool is based on program slicing, a technique which has recently received much attention as a device for program analysis, understanding, and validation[1]. But slicing can sometimes deliver too imprecise information, in particular if the program contains complex data structures. We will show how slicing can be combined with constraint solving in order to increase precision. In particular, the method allows to extract precise (and understandable) necessary conditions under which a certain dataflow (e.g. from keyboard to calibration path) can happen. Thus the technique not only improves slicing, but also allows for the generation of error messages as sketched above.

2 Program Slicing

Program Slicing was originally introduced by Weiser as a technique to support debugging [16]. Informally, a slice is defined as follows: Given a statement s and a variable v in s, determine all statements which might affect the value of v at s. Today, there is a wealth of algorithms for slicing, as well as numerous applications besides debugging. It is now possible to handle complex languages like full C (perhaps with some restrictions for pointers), and slicing has found successful applications in program understanding and software maintenance. Tip [15] presents an overview of current slicing technology; [9] describes applications in software engineering. We will only present the most basic definitions.

Definition 1. A *control-flow graph* (CFG) contains one node for each statement and control predicate of a program. An edge from node i to node j indicates the possible control flow from the former to the latter. The variables which are referenced at node i are denoted $ref(i)$; the variables which are defined at i (that is, assigned a value), are denoted $def(i)$.

Usually, a CFG does not contain transitive flow dependencies; we always assume that the CFG is transitively irreducible. Furthermore, we assume that there are special START and STOP nodes corresponding to the beginning and ending of the program or procedure.

[1] In fact, Denning proposed as early as 1977 to use data flow analysis for the validation of safety critical software [6]

Definition 2. A statement j is called *data flow dependent* on statement i, if

1. there is a path p from i to j in the CFG;
2. there is a variable v, where $v \in \mathit{def}(i)$ and $v \in \mathit{ref}(j)$;
3. for all statements $k \neq i$ in p, $v \notin \mathit{def}(k)$.

In this definition, complex data structures like arrays and records are treated as single variables, that is, there are data flow dependencies even if different or disjoint subcomponents of the data structure are involved. Pointers are ignored alltogether in this definition (and in this paper), as well as problems of VAR-parameters and aliasing. There are good conservative approximations to data flow dependencies in the presence of aliases which can easily be integrated into the slicing framework [12, 3]. For pointers and complex data structures, several authors proposed the use of abstract memory locations [8, 1]. The analysis of dependencies between arrays and array components has been studied intensively in the framework of program optimization and parallelization. We will not discuss any of these extensions, but only remark that for pointers, arrays, and VAR-parameters the computed slices may be too big[2]. Thus constraint solving promises to be useful for these extensions as well. For arrays, we will see later how constraint solving can improve slice accuracy.

Definition 3. Statement j is called a *postdominator* of statement i, if any path p from i to STOP must go through j. i is called a *predominator* of j if any control flow from START to j must go through i.

In typical programs, statements in loop bodies are predominated by the loop entry point and postdominated by the loop exit point. In fact, the notion of pre- and postdominators can be used to identify loops in arbitrary flow graphs.

Definition 4. Statement j is called *control dependent* on control predicate i, if

1. there is a path p in the CFG from i to j;
2. j is a postdominator for every statement in p except i;
3. j is not a postdominator for i.

Intuitively, j is control dependent on i if j is "governed" by i. For languages with structured control constructs like IF and WHILE, j is control dependent on i, if i is the condition of an IF or WHILE, and j is in the body of this IF resp. WHILE. For languages with arbitrary GOTOs, the computation of control dependencies is much more expensive [4].

Usually, control dependencies are marked with TRUE or FALSE, in order to distinguish between THEN- and ELSE part of an IF statement. For CASE or SWITCH statements, control dependencies are marked with the value of the selection expression which leads to the statements of a certain case. Thus a

[2] For records and variant records, exact data flow dependencies for record components can easily be determined.

```
read(n);
i:=1;
sum:=0;
product:=1;
WHILE true DO
  BEGIN
    IF i>n THEN
      GOTO L;
    sum:=sum+i;
    product:=product*i;
    i:=i+1
  END;
L: write(sum);
write(product);
```

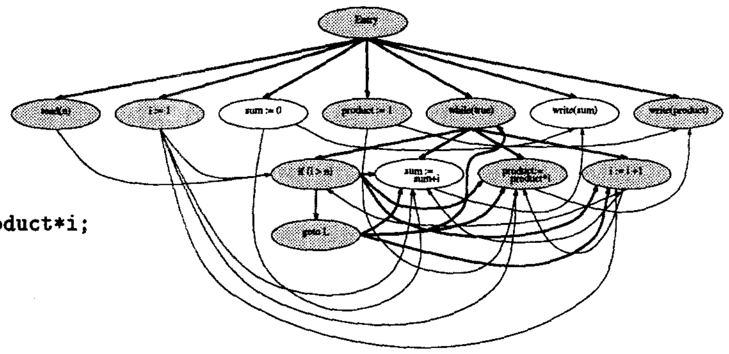

Fig. 1. A program and its PDG

control dependency can actually be followed only if the predicate has the value which is attached to the control dependency edge.

Definition 5. The *program dependency graph* (PDG) consists of all statements and control predicates as nodes, and all data flow and control dependencies as edges. In a PDG, $i \to j$ denotes both a data flow or a control flow edge from i to j. $\to^*$ is the transitive, reflexive closure of $\to$.

There are more elaborated versions of the PDG, but for slicing, this simple definition suffices. Figure 1 shows an example of a program and its PDG.

Definition 6. 1. The *forward slice* of a PDG node i consists of all nodes which depend on i:
$FS(i) = \{j \mid i \to^* j\}$.
2. The *backward slice* of a PDG node i consists of all nodes on which s depends:
$BS(i) = \{j \mid j \to^* i\}$.
3. The *chop* of two nodes i, j consists of all nodes on any path from i to j:
$CH(i,j) = FS(i) \cap BS(j)$.

In case there are procedures, simple PDGs do no longer suffice. First of all, the PDGs of all procedures must be determined. From these PDGs, summary graphs are computed, which contain transitive dependencies between input and output parameters. In the presence of recursion, the computation of summary graphs is nontrivial; an efficient algorithm is described in [13].

Every procedure call in the main PDG is then replaced by the corresponding summary graph (which in turn contains links to the procedure's PDG). The result is called system dependency graph (SDG). From the SDG, slices are computed by a two-step algorithm: first, the top level slice is determined; afterwards slices in called procedures are added as necessary. Due to lack of space, we omit a more detailed description; we just remark that the method described in this paper can easily be extended to system dependency graphs.

3 Path Conditions and Constraint Solving

In this core section, we will explain how slicing can be improved by collecting and solving path conditions. The aim of our technique is twofold:

1. Slices can be made more precise by evaluating additional constraints on PDG edges; it may even happen that PDG edges disappear (which means that data or control flow is in fact impossible).
2. The precise conditions which enable a PDG path or chop to be executed can be determined and – after simplification and pretty-printing – used for diagnostic messages about a program.

As a motivating example, consider the following code piece:

```
(1)   read(i,j);
(2)   a[i+3]:=x;
(3)   IF i>10 THEN
(4)     IF j<20 THEN
(5)       y:=a[2*j-32]
(6)     ELSE
(7)       y:=17;
```

The PDG will contain a dataflow dependency edge (2)→(5). But obviously, a data flow from (2) to (5) is possible only if $i+3 = 2j-32$; this condition is called a *data flow condition.* Furthermore, (5) is executed only if $i > 10 \wedge j < 20$; these conditions are called *control flow conditions.* But the data flow and control flow conditions cannot be satisfied simultaneously, as $i + 3 > 13$ and $2j - 32 < 8$. Thus a dataflow (2)→(5) is impossible! Furthermore, (7) will be executed only if $i > 10 \wedge j \geq 20$, and this control flow condition might be interesting to somebody analysing the program. But note that in general, different occurences of variables must be distinguished, that is, indexed with the PDG node they occur in:

```
(1)  WHILE x<7 DO
(2)    x:=y+z;
(3)    IF x=8 THEN
(4)      p();
     OD;
```

In this example, (4) will only be executed if $x_1 < 7$ and $x_3 = 8$. As we have a dataflow (2)→(3), x_3 is in fact equal to $y_2 + z_2$. Due to this *dependency equation,* the condition governing (4) becomes $x_1 < 7 \wedge y_2 + z_2 = 8$.

We will now precisely describe how path conditions are constructed and solved. The basic idea is to attach an additional constraint to each edge of the PDG. These constraints will be collected appropriately in order to obtain conditions for the execution of single statements or conditions for a data flow along a path in the PDG.

3.1 Dependency Equations

As demonstrated in the above example, all program variables occuring in a flow condition must be labelled with the statement or predicate they occur in. But due to dataflow dependencies, different occurences of the same variable may again be equated. In this section, these equations are formally introduced. The resulting dependency equations are independent of particular slices or execution paths. They are not necessary conditions for certain control or data flow, but are auxiliary conditions, which – as indicated above – can be used to simplify other data or control flow conditions. In contrast to the path conditions defined later, dependency equations always hold during program execution.

Definition 7. Let j be a PDG node, let v be a program variable. The notation v_j denotes the value of v after execution of j. The notation v^j denotes the value of v before execution of j.

Note that j might be executed several times, thus v_j, v^j can change during program execution. If $v \notin def(j)$, $v_j = v^j$.

Definition 8. Let $e_1 = i_1 \rightarrow j, \ldots, e_k = i_k \rightarrow j$ be dataflow edges to node j, which are due to variable v. The resulting dependency equation is $f(v^j) \equiv v^j = v_{i_1} \vee \ldots \vee v^j = v_{i_k}$. In particular, $k = 1$ leads to $f(v^j) \equiv v_i = v^j$. For a PDG G, the set of all dependency equations is denoted $F(G)$.

We assume $F(G)$ to include all equations $v_j = v^j$, as specified in definition 7. Sometimes it might be worthwile to take an even closer look at variable values, e.g. as resulting from assignments. Therefore, we provide the following definition.

Definition 9. Let i be an assignment `v:=E`, and let $i \rightarrow j$ be a dataflow dependency from i to j due to v. Let $\overline{E}$ denote expression `E`, where all variables $a, b, c, \ldots \in E$ have been replaced by $a_i, b_i, c_i, \ldots$. Then the extended dependency equation is $\overline{f}(v^j) \equiv v^j = v_i \wedge v_i = \overline{E}$. The set of all extended dependency equations is denoted $\overline{F}(G)$.

3.2 Flow Conditions

Control or data flow conditions are attached to certain PDG edges and must be satisfied in order that the edge be included in a slice. They will later be used for the construction of statement or path-governing expressions.

Definition 10. 1. Let $e = i \rightarrow^x j$ be a control dependency edge, where i is the predicate and the edge is marked with value x. The corresponding control flow condition is $c(e) \equiv i = x$.[3]

[3] For control dependencies originating from a GOTO statement (see Fig. 1), we define $c(e) = true$.

2. Let $e = i \rightarrow j$ be a data dependency edge, where v is the variable carrying the dependency. Let $B(v)$ be any condition on v. The corresponding data flow condition is $d(e) \equiv B(v)$.

   Example: For array references, data flow conditions can explicitely be stated. If v is an array, i is an assignment, $v[E_1]$ is the left hand side of the assignment, and another reference $v[E_2]$ occurs in j, the corresponding data flow condition is $d(i \rightarrow j) \equiv B(v) \equiv \overline{E_1} = \overline{E_2}$. [4]
3. The *control dependency graph* (CDG) is the subgraph of the PDG containing only control dependency edges. A missing (control or data) flow condition is considered equivalent to *true*.

Control flow conditions must be true in order that a statement be executed. For example, in

```
(1)  a[u]:=x;
(2)  IF a[v]=y THEN
(3)     p();
```

the control flow condition $a[v] = y$ must be true in order that (3) is executed. On the other hand, data flow conditions are not necessary for the execution of statements, but for a data flow along a certain arc in a PDG. In the example, the dataflow condition for the array references (as given above) is $d((1) \rightarrow (2)) \equiv u = v$; it is not necessary for the execution of (2) or (3), but $u = v$ must hold in order that a dataflow from (1) to (2) takes place.

3.3 Execution Conditions

In order to capture the conditions under which a certain statement may be executed, the following definitions are introduced. Obviously, a PDG node i can only be executed if all controlling predicates become successively true during program execution. If there are several paths from START to i, at least one must be executed.

Definition 11. Let i be a PDG node. Let $P = p_1, p_2, \ldots, p_n$ be a path from START to i, where $p_1 = \text{START}$, $p_n = i$, and $p_2, \ldots, p_{n-1}$ are control predicates. The execution condition for P is

$$E(P) = \bigwedge_{\nu=1}^{n-1} c(p_\nu \rightarrow p_{\nu+1})$$

Now let $P_1, P_2, \ldots, P_k$ be such paths from from START to i. The execution condition for i is

$$E(i) = \bigvee_{\mu=1}^{k} E(P_\mu)$$

[4] For other language constructs, other data flow conditions may be introduced – the generation of data flow conditions is outside the scope of this paper.

In order that node i is executed, it is necessary to find values for the program variables (indexed with PDG nodes) such that $E(i)$ evaluates to *true*. These values will usually not show up simultanuously during program execution; on the contrary, different instances v_i, v^i, v^j, v_j of variable v will obtain the required values for different program states. But is is a necessary condition that $E(i)$ is solvable, otherwise i cannot be executed. $F(G)$ and $\overline{F}(G)$ must hold as well.

In case there are only structured statements, the control dependencies form a tree, thus for every statement i there is at most one path from START to i, and the computation of $E(i)$ is easy. In general however, the control flow edges form a directed graph which may even contain cycles. In this case, the above formula for $E(i)$ is – albeit correct – not suitable for computation. Even if the control dependency subgraph of the PDG is cycle free, the above expression for $E(i)$ may contain countless copies of the same control predicates.

We therefore develop a simpler formula for $E(i)$. Note that the brute force method of computing a minimal normal form for $E(i)$ can have exponential time complexity – this is the motivation for the following derivations, which utilize the special structure of the $E(i)$.

Definition 12. Let $P_1 = p_1^1 \dots p_{l_1}^1, P_2 = p_1^2 \dots p_{l_2}^2, \dots, P_n = p_1^n \dots p_{l_n}^n$ be the paths from i to j. Then

$$E(i,j) = \bigvee_{\mu=1}^{n} \bigwedge_{\nu=1}^{l_\mu - 1} c(p_\nu^\mu \rightarrow p_{\nu+1}^\mu)$$

The expression $E(i,j)$ generalizes $E(j)$ and collects the control flow conditions between i and j. Obviously, $E(j) = E(\text{START}, j)$. The definition of $E(i,j)$ can easily be generalized for the case that – due to cycles in the PDG – there are infinitely many paths from i to j.

In practice, the paths from i to j will have long common subpaths. This can be used to simplify execution conditions by "factoring out" the subpath. Let $s_1 \dots s_l$ be such a common subpath, where there is no other path from s_1 to s_l. Then by the distributive law,

$$E(i,j) = E(i, s_1) \wedge \bigwedge_{\nu=1}^{l-1} c(s_\nu \rightarrow s_{\nu+1}) \wedge E(s_l, j)$$

as can easily be verified. This formula avoids redundant copies of the $c(e)$ on the common subpaths and can easily be generalized to more than one common subpath. It should be used with common subpaths as long as possible, such that in the remaining $E(x,y)$, the paths between x and y are hopefully disjoint.

Next, we will demonstrate that control dependency cycles can be ignored.

Lemma 13. *Let $P = p_1 p_2 \dots p_{k-1} q\ p_{k+1} \dots p_n$ be the one and only path from i to j (that is, $p_1 = i, p_n = j$), where a cycle $q_0 \dots q_m$ is attached to P (that is, $q_o = q_m = q$). Then the cycle does not contribute to $E(i,j)$ and can safely be ignored.*

Proof. Let μ index over the paths from p_1 to p_n. Then

$$\begin{aligned}
E(p_1,p_n) &= \bigvee_\mu \bigwedge_{\nu=1}^{l_\mu-1} c(p_\nu^\mu \to p_{\nu+1}^\mu) \\
&= \bigwedge_{\nu=1}^{n-1} c(p_\nu \to p_{\nu+1}) \\
&\quad \vee \bigwedge_{\nu=1}^{k-1} c(p_\nu \to p_{\nu+1}) \wedge \bigwedge_{\nu=0}^{m-1} c(q_\nu \to q_{\nu+1}) \wedge \bigwedge_{\nu=k}^{n-1} c(p_\nu \to p_{\nu+1}) \\
&\quad \vee \bigwedge_{\nu=1}^{k-1} c(p_\nu \to p_{\nu+1}) \wedge \bigwedge_{\nu=0}^{m-1} c(q_\nu \to q_{\nu+1}) \wedge \\
&\qquad \bigwedge_{\nu=0}^{m-1} c(q_\nu \to q_{\nu+1}) \wedge \bigwedge_{\nu=k}^{n-1} c(p_\nu \to p_{\nu+1}) \\
&\quad \vee \ \ldots \\
&= \bigwedge_{\nu=1}^{n-1} c(p_\nu \to p_{\nu+1})
\end{aligned}$$

due to the absorption law $(A \vee A \wedge B = A)$. □

The lemma is still true if there are two or more cycles attached to a path. Furthermore, the lemma can be applied to all paths from i to j. Hence we obtain the

Theorem 14. *For the computation of $E(i,j)$, all cycles can be ignored.* [5]

Proof. Let P_μ be the infinitely many paths from i to j. Let $P_1,\ldots,P_n$ be the finitely many "skeleton" paths obtained by removing any cycles attached to a P_ν. Then by the lemma
$E(i,j) = \bigvee_\mu \bigwedge_{p_\nu \to p_{\nu+1} \in P_\mu} c(p_\nu \to p_{\nu+1}) = \bigvee_{i=1}^{n} \bigwedge_{p_\nu \to p_{\nu+1} \in P_i} c(p_\nu \to p_{\nu+1})$. □

Hence the set of paths from i to j can always be considered finite. A path-finding algorithm can just stop if it encounters the same node for the second time. Intuitively, the reason is that cycles only make execution conditions weaker, and since we are interested in necessary conditions as strong as possible, cycles can be ignored.

Thus we can assume that the set of paths between two nodes is a directed acyclic graph. This can be utilized if all $E(i,p_\nu)$ for successors p_ν of i are needed: all nodes p_ν on paths from i to j can be topologically sorted, and the $E(i,p_\nu)$ are computed in topological order. Trivially, $E(i,i) = \mathit{true}$. Now let j be a PDG node with immediate control predecessors $p_1,p_2,\ldots,p_k$. We assume that $E(i,p_1),E(i,p_2),\ldots,E(i,p_k)$ have already been determined, as the nodes are processed in topological order. Then

$$E(i,j) = \bigvee_{\rho=1}^{k} E(i,p_\rho) \wedge c(p_\rho \to j)$$

Lemma 15. *For any j which is a successor of i, the above formula correctly computes $E(i,j)$.*

[5] This is even true for overlapping cycles, which are ignored here due to space limitations.

Proof. The proof is by induction on the topological order. For $j = i$, the statement is trivial. Otherwise we assume that the $E(i, p_\rho)$ are correct. Then $E(i,j) = \bigvee_{\mu=1}^{n} \bigwedge_{\nu=1}^{l_\mu-1} c(p_\nu^\mu \to p_{\nu+1}^\mu)$, where $p_1^\mu = i$, $p_{l_\mu}^\mu = j$, and $p_{l_\mu-1}^\mu = p_\rho$ for a suitable ρ. By grouping the paths which share the last edge $p_\rho \to j$ we obtain $E(i,j) = \bigvee_{\rho=1}^{k} \bigvee_{\mu_\rho=1}^{n_\rho} \bigwedge_{\nu=1}^{l_{\mu_\rho}-1} c(p_\nu^{\mu_\rho} \to p_{\nu+1}^{\mu_\rho}) = \bigvee_{\rho=1}^{k} \bigvee_{\mu_\rho=1}^{n_\rho} \bigwedge_{\nu=1}^{l_{\mu_\rho}-2} c(p_\nu^{\mu_\rho} \to p_{\nu+1}^{\mu_\rho}) \wedge c(i_\rho \to j)$. By induction hypothesis, $\bigvee_{\mu_\rho=1}^{n_\rho} \bigwedge_{\nu=1}^{l_{\mu_\rho}-2} c(p_\nu^{\mu_\rho} \to p_{\nu+1}^{\mu_\rho}) = E(i, p_\rho)$, hence the lemma follows. □

The execution conditions are needed for the computation of path conditions in the PDG. If many such conditions must be determined and solved for the same program, it might be wise to precompute all the needed $E(j) = E(\text{START}, j)$ once and attach them to the PDG nodes j. Computation in topological order will make generation of path conditions much faster.

3.4 Path Conditions

We will now establish necessary conditions which must be fulfilled in order that a data flow between two PDG nodes i and j can take place. Of course, there must be a path from i to j. Furthermore, all nodes on the path must be executable, that is, their execution conditions must be satisfyable. Finally, any data flow conditions on arcs along the path must be fulfilled as well. If there are several paths between i, j, at least one of them must have a satisfyable path condition.

Definition 16. Let $P = p_1, p_2, \ldots, p_n$ be any path in the PDG connecting nodes i and j. The path condition for P is

$$PC(P) = \bigwedge_{\nu=1}^{n} E(p_\nu) \wedge \bigwedge_{\nu=1}^{n-1} d(p_\nu \to p_{\nu+1})$$

In case there are several paths $P_1, P_2, \ldots, P_k$ from i to j, the path condition is

$$PC(i,j) = \bigvee_{\mu=1}^{k} PC(P_\mu) = \bigvee_{\mu=1}^{k} \bigwedge_{\nu=1}^{n_\mu} E(p_\nu) \wedge \bigwedge_{\nu=1}^{n_\mu-1} d(p_\nu \to p_{\nu+1})$$

Cycles can safely be ignored, due to the same argument as in the previous section. But again we face the problem that $PC(i,j)$ will contain lots of redundant copies of identical control flow conditions. Again, we try to factor out common subpaths (say, subpath $s_1 \ldots s_l$), before more subtle simplification takes place:

$$PC(i,j) = PC(i, s_1) \wedge \bigwedge_{\nu=1}^{l} E(s_\nu) \wedge \bigwedge_{\nu=1}^{l-1} d(s_\nu \to s_{\nu+1}) \wedge PC(s_l, j)$$

Next, we try to simplify $\bigwedge_{\nu=1}^{n} E(p_\nu)$. We first consider the very common situation that the CDG is a tree (or, more precisely, for any $p_\nu \in P$ there is only one PDG path to START). In this case, a redundancy-free formula for $\bigwedge_{\nu=1}^{n} E(p_\nu)$ can easily be obtained. As a first step, "inner" nodes are removed: if $p_\nu \rightarrow^* p_\mu$ in the CDG, $E(p_\mu) \Rightarrow E(p_\nu)$, thus by the absorption law, $E(p_\nu)$ does not contribute anything hence P_ν can be ignored. Now let $p_\nu, p_\mu \in P$, and let $a = \text{lca}(p_\nu, p_\mu)$ be their least common anchestor in the CDG. Let $q_1 \dots q_{\rho_\nu}$ be the path from a to p_ν, and $r_1 \dots r_{\rho_\mu}$ be the path from a to p_μ. Then by definition of E

$$E(p_\nu) \wedge E(p_\mu) = E(a) \wedge E(q_1 \dots q_{\rho_\nu}) \wedge E(r_1 \dots r_{\rho_\mu})$$

The right side does not contain redundant copies of any $c(e)$. Hence by succesively determining least common anchestor and applying the formula, we obtain an expression which contains any $c(e)$ at most once. These observations lead to

Theorem 17. *Let $p_1, p_2, \dots, p_n$ be CDG nodes (e.g. from a path in the PDG). If the CDG is a tree (or, more precisely, if any p_i has only one CDG path to START), then*

$$\bigwedge_{i=1}^{n} E(p_i) = \bigwedge_{\substack{a\to b \text{ on a path from} \\ \text{START to any } p_i}} c(a \rightarrow b)$$

Proof. The proof is by induction on n. For $n = 1$, the statement follows by definition of $E(p_1)$ (note there is only one path from START to p_1). If we add a new node p_{n+1}, let START$, p^1, p^2 \dots, p^{\mu_{n+1}}, p_{n+1}$ be the unique path from START to p_{n+1}. Then there must exist $p_k \in \{p_1, \dots, p_n\}$ and a $p^\nu = \text{lca}(p_{n+1}, p_k)$ such that subpath $p^{\nu+1} \dots p_{n+1}$ is disjoint from all paths from START to any $p_1, \dots, p_n$, while $START \dots p^\nu$ occur also in the path from START to p_k. Therefore, for any edge $a \rightarrow b$ in path $START \dots p^\nu$, $c(a \rightarrow b)$ already occurs in $\bigwedge_{\substack{a\to b \text{ on a path from} \\ \text{START to any } p_1 \dots p_n}} c(a \rightarrow b)$. Thus

$$\begin{aligned}
\bigwedge_{i=1}^{n+1} E(p_i) &= E(p_{n+1}) \wedge \bigwedge_{i=1}^{n} E(p_i) \\
&= \bigwedge_{i=0}^{\mu_{n+1}-1} c(p^i \rightarrow p^{i+1}) \wedge \bigwedge_{\substack{a\to b \text{ on a path from} \\ \text{START to any } p_1 \dots p_n}} c(a \rightarrow b) \\
&= \bigwedge_{i=0}^{\nu-1} c(p^i \rightarrow p^{i+1}) \wedge \bigwedge_{i=\nu}^{\mu_{n+1}-1} c(p^i \rightarrow p^{i+1}) \\
&\quad \wedge \bigwedge_{\substack{a\to b \text{ on a path from} \\ \text{START to any } p_1 \dots p_n}} c(a \rightarrow b) \\
&= \bigwedge_{i=\nu}^{\mu_{n+1}-1} c(p^i \rightarrow p^{i+1}) \wedge \bigwedge_{\substack{a\to b \text{ on a path from} \\ \text{START to any } p_1 \dots p_n}} c(a \rightarrow b) \\
&= \bigwedge_{\substack{a\to b \text{ on a path from} \\ \text{START to any } p_1 \dots p_{n+1}}} c(a \rightarrow b)
\end{aligned}$$

via definition of $E(p_{n+1})$, induction hypothesis and absorption law. □

For programs with non-structured control flow, it might be that $E(p_\nu)$ and $E(p_\mu)$ have common control conditions $c(e)$. By using the distributive law, these can always be factored out, hence $E(p_\nu) \wedge E(p_\mu)$ always has a redundancy-free definition. But these are difficult to obtain in general, and we omit the derivation of the general formula.

3.5 Solving the Path Conditions

We have seen how to construct path conditions which are necessary for a certain dataflow to take place. By construction, the expressions defining the path conditions usually are redundant free, that is, they do not contain multiple copies of the same control flow condition. But this property cannot always be guaranteed – if the CDG is not a tree, and there is more than one path between two PDG nodes, the elimination of multiple control flow conditions during generation of path conditions requires an enourmous effort.

Hence path conditions must be simplified, and even if they are in minimal normal form, they are not necessarily in solved form.

Definition 18. A path condition is in solved form, if all its equations are of the form $v_i = P(\overline{v})$, where $\overline{v}$ is the set of (indexed) program variables, and there are no direct or indirect recursive dependencies between two variables v_i and v_j.

Simplification and subsequent solving can be done in three steps:

1. Simplification based on the following rewrite rules:
$$A \wedge true \rightarrow A, \quad A \wedge A \rightarrow A, \quad A \vee A \rightarrow A,$$
$$A \vee (A \wedge B) \rightarrow A, \quad A \wedge B \vee A \wedge C \rightarrow A \wedge (B \vee C)$$
A simplifier based on these – and similar – rewrite rules can easily be implemented; it should apply rewrite rules with priority from left to right. Simplification based on rewriting has always polynomial time complexity. Note that factorizing common subpaths as described above can in principle be omitted and replaced by simplification. But it seems simpler to factor out common paths and execution conditions right from the beginning. The example in section 4 will show that simplification alone can produce reasonable understandable path conditions.
2. If the resulting simplified path condition still contains redundant copies of elementary flow conditions, a computation of minimal disjunctive normal form may be appropriate. The standard minimization algorithms (e.g. Quine/McCluskey) all can have exponential time complexity. The Quine-McCluskey algorithm, for example, first determines the set of prime implicands (which is trivial in our application, as there are usually no negated flow conditions); afterwards, a branch-and-bound algorithm computes a minimal cover of prime implicands which already generate the whole path condition.
3. As a last step, one might try to solve the minimized path condition by using a constraint solver or a system for symbolic mathematics. Constraint solving is the basic mechanism in constraint logic programming [11], and CLP(R) was one of the first available systems [10]. Several powerful constraint solvers are available today. The system CLP(BNR) can solve constraints on booleans, integers, and reals; it does not aim at symbolic simplification of constraints, but will try to determine actual intervals for the possible values of variables [5]. Other systems like Mathematica use sophisticated algorithms for symbolic simplification and solution of mathematical problems.

Most constraint solving systems use specialized algorithms for specific problems, e.g. the simplex algorithm for systems of linear inequations. In our case, the generated constraints can contain arbitrary target language expressions (see the example in section 4). Hence it is unlikely that a general-purpose problem solver will produce sophisticated solutions for our application, namely the validation of measurement software. But automatic solving of linear constraints like $(i + 3 = 2 \cdot j - 32) \wedge (i > 10) \wedge (j < 20)$ (see the example at the beginning of the section) is easy for today's systems, and this is at least a starting point. Current research in constraint solving is described in [2].

4 An Example

Figure 2 presents an excerpt of a fictious measurement system program. The example has been modelled after typical programs analysed by PTB. It reads a weight value from hardware port p_ab and an article number from port p_cd. The article number and the calibrated weight are displayed in an LCD unit. The program contains a calibration path violation: in case the "paper out" signal is active and the keyboard input is "+" or "-", the calibration factor is multiplied by 1.1 and thus 10% too high. It is the aim of this section to demonstrate how slicing discovers the calibration path violation, and how the formulas defined in the previous section yield the precise conditions under which this takes place.

Figure 3 shows the corresponding PDG. Node 1 is the START node (there is no extra START node, as the program contains only one top level statement). The PDG also contains the input and output ports, as well as a port for the initial value of kal_kg. Thick edges are control dependency edges. Note that there is no data dependency edge from node (idx==0) to (12), as elementary data flow analysis will detect that the loop will be executed at least once.

The backward slice from statement 14 (which is the printout of the weight value) not only contains the calibration statement (statement 3) and the data port for the weight value, but also statements 10 and 11 (modification of the calibration factor) and – via statement 7 – also the keyboard input port. Thus there is a possible influence of the keyboard to the weight value, as $p\_cd \in BS(14)$. This is certainly very suspicious and requires further investigation.

The chop $CH(p\_cd, 14)$ between keyboard and weight value display consists of several paths:

$(p\_cd) \rightarrow (5) \rightarrow (6) \rightarrow (8) \rightarrow (9) \rightarrow (11) \rightarrow (4) \rightarrow (14)$
$(p\_cd) \rightarrow (5) \rightarrow (6) \rightarrow (8) \rightarrow (9) \rightarrow (10) \rightarrow (4) \rightarrow (14)$
$(p\_cd) \rightarrow (5) \rightarrow (6) \rightarrow (7) \rightarrow (9) \rightarrow (11) \rightarrow (4) \rightarrow (14)$
$(p\_cd) \rightarrow (5) \rightarrow (6) \rightarrow (7) \rightarrow (9) \rightarrow (10) \rightarrow (4) \rightarrow (14)$
$(p\_cd) \rightarrow (5) \rightarrow (idx = 0) \rightarrow (6) \rightarrow (8) \rightarrow (9) \rightarrow (11) \rightarrow (4) \rightarrow (14)$
$(p\_cd) \rightarrow (5) \rightarrow (idx = 0) \rightarrow (6) \rightarrow (8) \rightarrow (9) \rightarrow (10) \rightarrow (4) \rightarrow (14)$
$(p\_cd) \rightarrow (5) \rightarrow (idx = 0) \rightarrow (6) \rightarrow (7) \rightarrow (9) \rightarrow (11) \rightarrow (4) \rightarrow (14)$
$(p\_cd) \rightarrow (5) \rightarrow (idx = 0) \rightarrow (6) \rightarrow (7) \rightarrow (9) \rightarrow (10) \rightarrow (4) \rightarrow (14)$
$(p\_cd) \rightarrow (5) \rightarrow (idx = 0) \rightarrow (8) \rightarrow (9) \rightarrow (11) \rightarrow (4) \rightarrow (14)$
$(p\_cd) \rightarrow (5) \rightarrow (idx = 0) \rightarrow (8) \rightarrow (9) \rightarrow (11) \rightarrow (4) \rightarrow (14)$

```
(1)  while(TRUE) {
(2)    if ((p_ab[CTRL2] & 0x10)==0) {
(3)      u = ((p_ab[PB] & 0x0f) << 8) + p_ab[PA];
(4)      u_kg = u * kal_kg;
         }
(5)    if ((p_cd[CTRL2] & 0x01) != 0) {
(6)      for (idx=0;idx<7;idx++) {
(7)        e_puf[idx] = p_cd[PA];
(8)        if ((p_cd[CTRL2] & 0x10) != 0) {
(9)          switch(e_puf[idx]) {
(10)            case '+': kal_kg *= 1.1; break;
(11)            case '-': kal_kg *= 0.9; break;
                }
             }
           }
(12)     e_puf[idx] = '\0';
         }
(13)   printf("Artikel: %07.7s\n",e_puf);
(14)   printf("     %6.2f kg     ",u_kg);
       }
```

Fig. 2. Excerpt from measurement software

$$\begin{array}{l}
(p\_cd) \rightarrow (5) \rightarrow (idx = 0) \rightarrow (7) \rightarrow (9) \rightarrow (11) \rightarrow (4) \rightarrow (14) \\
(p\_cd) \rightarrow (5) \rightarrow (idx = 0) \rightarrow (7) \rightarrow (9) \rightarrow (10) \rightarrow (4) \rightarrow (14) \\
(p\_cd) \rightarrow (8) \rightarrow (9) \rightarrow (11) \rightarrow (4) \rightarrow (14) \\
(p\_cd) \rightarrow (8) \rightarrow (9) \rightarrow (10) \rightarrow (4) \rightarrow (14) \\
(p\_cd) \rightarrow (7) \rightarrow (9) \rightarrow (11) \rightarrow (4) \rightarrow (14) \\
(p\_cd) \rightarrow (7) \rightarrow (9) \rightarrow (10) \rightarrow (4) \rightarrow (14)
\end{array}$$

In this list, all cycles involving `idx++` have been ignored, which is allowed according to the previous section.

The path condition is determined according to $PC(p\_cd, 14) = \bigvee_\mu \bigwedge_\nu E(p_\nu)$.[6] There is only one subpath common to all paths, namely $(4) \rightarrow (14)$. We therefore factor out $PC(4, 14) \equiv E(4) \wedge E(14)$. Furthermore, $E(p\_cd) = E(5) = E(14) = \mathit{true}$.[7] Now it turns out that the CDG of the program is a tree. Hence an $E(p_\nu)$ contributes nothing if p_ν is an inner node. This leads to a dramatic reduction of the path condition:

[6] There is only one data flow condition: $d((7) \rightarrow (9)) \equiv idx = idx$, which is trivial and hence deleted. Thus the data flow conditions contribute nothing.

[7] The control flow conditions outgoing from (1) are all TRUE and hence deleted.

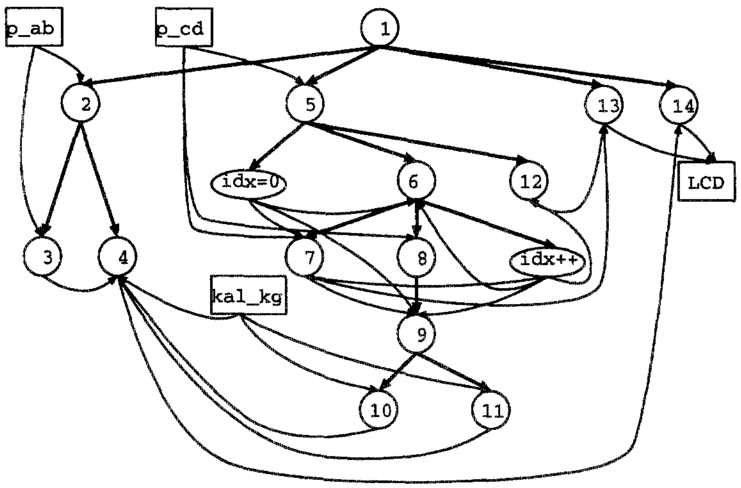

Fig. 3. PDG for Fig. 2

$$\begin{aligned}PC(p\_cd,14) = \Big(&E(11) \vee E(10) \vee (E(7) \wedge E(11)) \vee (E(7) \wedge E(10))\\ &\vee(E(idx=0) \wedge E(11)) \vee (E(idx=0) \wedge E(10))\\ &\vee(E(idx=0) \wedge E(7) \wedge E(11)) \vee (E(idx=0) \wedge E(7) \wedge E(10))\\ &\vee(E(idx=0) \wedge E(11)) \vee (E(idx=0) \wedge E(10))\\ &\vee(E(idx=0) \wedge E(7) \wedge E(11)) \vee (E(idx=0) \wedge E(7) \wedge E(10))\\ &\vee E(11) \vee E(10) \vee (E(7) \wedge E(11)) \vee (E(7) \wedge E(10))\Big)\\ &\wedge E(4)\end{aligned}$$

Simplification via idempotency and absorption laws leads to a collaps of this condition:

$$PC(p\_cd,14) = \big(E(10) \vee E(11)\big) \wedge E(4)$$

According to the CDG,

$$E(10) \equiv c((5) \rightarrow (6)) \wedge c((6) \rightarrow (8)) \wedge c((8) \rightarrow (9)) \wedge c((9) \rightarrow (10))$$
$$E(11) \equiv c((5) \rightarrow (6)) \wedge c((6) \rightarrow (8)) \wedge c((8) \rightarrow (9)) \wedge c((9) \rightarrow (11))$$

which leads – after factorization – to the path condition

$$PC(p\_cd,14) = (c((9) \rightarrow (10) \vee c((9) \rightarrow (11)) \wedge c((5) \rightarrow (6)) \wedge c((6) \rightarrow (8)) \wedge c((8) \rightarrow (9)) \wedge c((2) \rightarrow (4))$$

The control flow conditions are:

$$\begin{aligned}c((2) \rightarrow (3)) \equiv c((2) \rightarrow (4)) &\equiv p\_ab_2[CTRL2]\&0x10 = 0\\ c((5) \rightarrow (6)) \equiv c((5) \rightarrow (12)) &\equiv p\_cd_5[CTRL2]\&0x01 \neq 0\\ c((6) \rightarrow (7)) \equiv c((6) \rightarrow (8)) &\equiv idx_6 < 7\\ c((8) \rightarrow (9)) &\equiv p\_cd_8[CTRL2]\&0x10 \neq 0\\ c((9) \rightarrow (10)) &\equiv e\_puf_9[idx_9] = \text{“+”}\\ c((9) \rightarrow (11)) &\equiv e\_puf_9[idx_9] = \text{“–”}\end{aligned}$$

which leads to the explicit path condition

$$\begin{aligned} PC(p\_cd, 14) = & \big(e\_puf_9[idx_9] = \text{“+”} \vee e\_puf_9[idx_9] = \text{“–”}\big) \\ & \wedge p\_cd_5[CTRL2]\&0x01 \neq 0 \wedge idx_6 < 7 \\ & \wedge p\_cd_8[CTRL2]\&0x10 \neq 0 \wedge p\_ab_2[CTRL2]\&0x10 = 0 \end{aligned}$$

The dependency equations are $F(G) \equiv u_2 = u_3 \wedge u\_kg_4 = u\_kg_{14} \wedge (kal\_kg_4 = kal\_kg_{10} \vee kal\_kg_4 = kal\_kg_{11} \vee kal\_kg_4 = kal\_kg_0) \wedge e\_puf_7 = e\_puf_9 \wedge e\_puf_{12} = e\_puf_{13} \wedge (idx_7 = idx_{idx=0} \vee idx_7 = idx_{idx=0}) \wedge (idx_9 = idx_{idx=0} \vee idx_9 = idx_{idx++}) \wedge idx_{12} = idx_{idx++} \wedge (idx_6 = idx_{idx++} \vee idx_6 = idx_{idx=0})$. Hence $idx_7 = idx_9$, and $\overline{F}(G)$ contains $e\_puf_7[idx_7] = p\_cd_7[PA]$. Applying $\overline{F}(G)$ to the path condition, and removing the conditions which do not invole input ports yields

$$\begin{aligned} PC(p\_cd, 14) = & \big(p\_cd_7[PA] = \text{“+”} \vee p\_cd_7[PA] = \text{“–”}\big) \\ & \wedge p\_cd_5[CTRL2]\&0x01 \neq 0 \wedge p\_cd_8[CTRL2]\&0x10 \neq 0 \\ & \wedge p\_ab_2[CTRL2]\&0x10 = 0 \end{aligned}$$

Further constraint solving does not make sense, thus the last equation is presented to the user. Informally, it reads as follows: “If the keyboard input is + or –, and (not necessarily at the same time) the ‘paper out’ signal is active, there is data flow from the keyboard to the displayed weight value”. A human would have a hard time to extract such statements from large programs!

5 Conclusion

We have shown how to extend program slicing with constraint solving. For any slice or chop, path conditions can be generated, which are necessary conditions for data flow along a slice or chop. We have seen how path conditions are constructed, simplified and perhaps solved by a constraint solver. This leads to more precise information than traditional slicing. The method will be applied in a validation tool for measurement software, where any influences on the calibration path must be detected and analysed.

Although the implementation of the tool has just begun, and although several algorithmic details have been omitted in this paper, an example demonstrated the feasibility of the approach. Our ultimate hope is to analyse real-world programs written in full C, thereby providing semi-automatic software validation. We expect our tool to be useful not just for measurement software, but for other safety-critical software as well.

Acknowledgements. The work described in this paper is funded by the Bundesministerium für Bildung und Forschung, FKZ 01 IS 513 C9. U. Grottker provided the example program. The PDG example in section 2 was originally provided by T. Reps. J. Krinke provided valuable comments on a preliminary version of this paper.

References

1. H. Agrawal, R. DeMillo, E. Spafford: Dynamic slicing in the presence of unconstrained pointers. Proc. 4th Symposium on Testing, Analysis, and Verification. ACM 1991, pp. 60 - 73.
2. F. Benhamou, A. Colmerauer (Ed.): Constraint Logic Programming: Selected Research. MIT Press 1993.
3. J. Choi, M. Burke, P. Carini: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. Proc. 20th Principles of Programming Languages, ACM 1993, pp. 232 - 245.
4. J. Choi, J. Ferrante: Static slicing in the presence of GOTO statements. ACM TOPLAS 16(1994), pp. 1087 - 1113.
5. F. Benhamou, W. Older: Applying Interval Arithmetic to Real, Integer and Bolean Constraints. To appear in Journal of Logic Programming (1995).
6. D. Denning, P. Denning: Certification of programs for secure information flow. Communications of the ACM 20(7), S. 504 - 513, Juli 1977.
7. J. Field, G. Ramalingam, F. Tip: Parametric program slicing. Proc. 21th Symposium on Principles of Programming Languages, ACM 1995, S. 379 - 392.
8. S. Horwitz, P. Pfeiffer, T. Reps: Dependence analysis for pointer variables. Proc. SIGPLAN Programming Language Design and Implementation, ACM 1989, pp. 28 - 40.
9. S. Horwitz, T. Reps: The use of program dependence graphs in software engineering. Proc. 14th Int. Conference on Software Engineering, IEEE 1992, pp. 392 - 411.
10. J. Jaffar, S. Michaylow, P. Stuckey, R. Yap: The CLP(R) language and system. ACM TOPLAS 14(3), pp. 339 - 395 (Juli 1992).
11. J. Jaffar, M. Maher: Constraint logic programming: a survey. To appear in Journal of Logic Programming (1995).
12. W. Landi, B. Ryder: A safe approximation algorithm for interprocedural pointer aliasing. Proc. SIGPLAN Programming Language Design and Implementation, ACM 1992, pp. 93 - 103.
13. T. Reps, S. Horwitz, M. Sagiv, G. Rosay: Speeding up Slicing. Proc. 2nd SIGSOFT Foundations of Software Engineering, ACM 1994, pp. 11 - 20.
14. G. Smolka, M. Henz, J. Würz: Object-Oriented Concurrent Constraint Programming in Oz. DFKI Research Report 93–16.
15. F. Tip: A survey of program slicing techniques. Journal of Programming Languages 3 (1995), pp. 121 - 189.
16. M. Weiser: Program Slicing. IEEE Transactions on Software Engineering, 10(4), pp. 352 - 357, Juli 1984.