Validation of Measurement Software as an application of Slicing and Constraint Solving ⋆

Jens Krinke and 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 the VALSOFT slicing system. One particular application is the validation of computer-controlled measurement systems. VALSOFT will be used by the Physikalisch-Technische Bundesanstalt for verification of legally required calibration standards.

The article describes the VALSOFT slicing system. In particular, we describe how to generate and simplify path conditions based on program slices. A case study shows that the technique can indeed increase slice precision and reveal manipulations of the so-called calibration path.

Key words: 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

⋆ A preliminary version of parts of this article appeared in the proceedings of the Third Static Analysis Symposium[22]

† Article published in Information and Software Technology 40 (1998) 661–675. The original publication is available via doi:10.1016/S0950-5849(98)00090-1

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, called VALSOFT, analyses C source code and supports the PTB engineer by visualizing the obtained information. VALSOFT checks in particular that there are no unwanted influences on the calibration path. If the calibration path is not safe, VALSOFT can 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".

As we will see, such precision of the analysis is not really possible, because the underlying technology sometimes gives false alarms. But for a validation tool, at least the following correctnes criterion—called *principle of conservative approximation*—must hold: whenever the tool says that the calibration path is *not* influenced, then this information *must* be reliable. Occasional false alarms however are acceptable while analysing safety-critical software. The principle of conservative approximation strongly influenced the design of VALSOFT.

It is the aim of this paper to describe the underlying technology of the new tool. Basically, VALSOFT 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

[1] In fact, Denning proposed as early as 1977 to use data flow analysis for the validation of safety critical software [9]

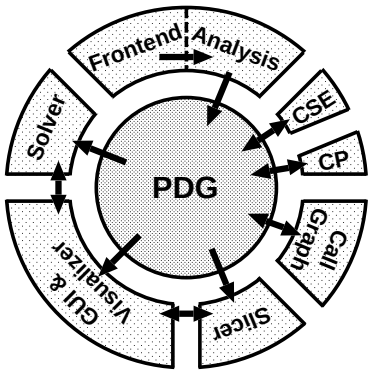

Fig. 1. VALSOFT kernel architecture

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 the generation of error messages as sketched above.

For the rest of this article, we assume that the reader has basic knowledge about program slicing, as described e.g. in [24] and in other articles in this volume. We will first describe the general design of the VALSOFT slicer, then introduce path conditions as a means to improve slice accuray. A final case study will show how our techniques can reveal manipulations in measurement system software.

2 The VALSOFT Slicing System

The VALSOFT Slicing System is a set of tools which are used together to analyze C source code to help the engineer to understand and validate the code. The main application is the calculation and visualization of slices for ANSI-C.

We are using a "whole world" approach for our analysis: the frontend (scanner/parser etc.) reads all preprocessed sources, constructs an attributed abstract syntax tree and a symbol table, and "links" all corresponding symbols of different sources together. The next step is a traversal of the AST which does a simple, flow insensitive data flow analysis. It calculates the *gmod* and $gref$ sets, the call graph, the points-to set and the frame of the PDGs, which consists of control flow and control dependence edges. A finer, flow sensitive analysis follows, which traverses the frame and calculates the data dependencies. The resulting PDGs are linked together to the SDG [14] at last.

The created SDG is persistently saved to disk and all other tools are working with the saved SDG. The set of tools and its architecture is shown in figure 1.

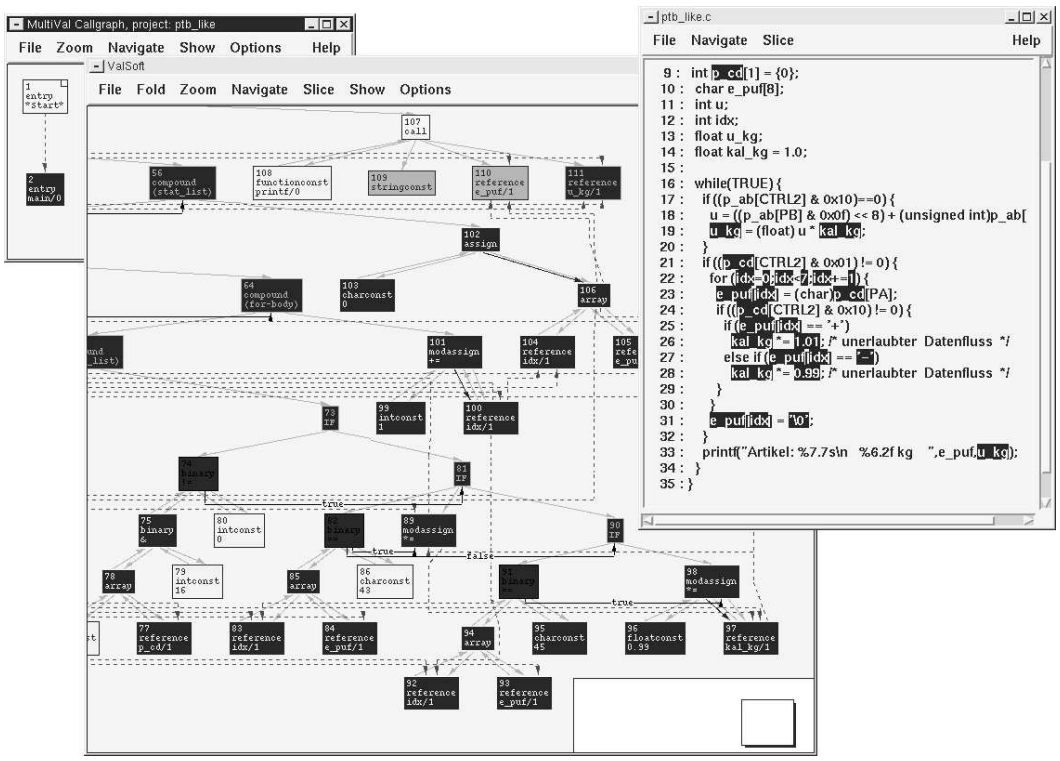

Fig. 2. VALSOFT user interface

The system consists of:

- The analyzer, which generates the SDG for a set of C sources.
- The slicer, which can do backward and forward slices and chops.
- The solver, which will be discussed in a following section.
- CP, which does constant propagation over the SDGs and tries to simplify them (eliminating edges and nodes).
- CSE, which tries to find common subexpressions and to simplify the SDG.
- A call graph simplifier, which eliminates redundant call edges in presence of function pointers.
- A GUI, which visualizes the PDGs and the corresponding sources and controls the execution of the slicer and the solver.

The GUI is used to navigate in the PDGs and the source. It first visualizes the call graph of the program, where the user may select any procedure node to let the GUI visualize the corresponding PDG. Every node is selectable and the corresponding parts of the source may be visualized in a textual manner. The user may select a set of nodes as a slicing criterion and let the GUI execute the slicer. The resulting set of nodes is visualized in the graph based and the textual presentation. Fig. 2 shows an example. The main window contains the PDG, where between an unvisible node and node 111 (upper right) a chop has been calculated. The right window shows the corresponding source with the chop inverted. The left window shows the call graph.

Our SDG is a specialization of the traditional [14]. Because one of our goals was to keep the SDG similar to the AST and the sources, transformations like SSA (static single assignment form) were not applicable and we had to directly deal with the possibility of multiple side effects in expressions. Therefore we decided to use a fine grained representation and to keep it similar to the AST and the traditional SDG. On the level of statements and expressions, the AST nodes are almost mapped one to one onto SDG nodes. The definitions of variables and procedures have special nodes. The nodes may be attributed with a class, an operator and a value. The class specifies the kind of node: statement, expression, procedure call etc. The operator specifies the kind further, e.g. binary expression, constant etc. The value carries the exact operator, like "+" or "–", constant values or identifier names. Other attributes are the enclosing procedure and a mapping to the source text.

To handle the specialized nodes we had to specialize the edges too. In most cases the execution order of expression components is undefined which results in representation problems. Consider the statement `z=x+y` for example. It is undefined if `x` is evaluated before `y` or vice versa. It is only defined that both have been evaluated before the values will be added. This behavior can not be represented in a normal CFG. However, its representation in the VALSOFT PDG is that the nodes of `x` and `y` are control dependent on the `+` node. We call this control dependence *immediate (control) dependence*, because it is independent of the value of a predicate.

Another goal was the ability to calculate slices for any node in the SDG. Therefore the data flow between the expression components had to be represented in the SDG. In our example, the value of the addition is dependent on the value of `x` and `y`. If we would add simple data dependence edges between the nodes of `+` and `x` and between the nodes of `+` and `y`, we would induce circles between the nodes of `+`, `x` and `y`. One way to solve this would be to split the node of `+` into a "before evaluation of subcomponents" node and an "after evaluation of subcomponents" node. We took a different approach and introduced another specialized edge: the *value dependence edge*, which is like a data dependence edge between expression components. The resolution of the arising circles has been delegated to the slicer and the other tools.

Another specialized edge was needed to represent the assignments of values to variables. The *reference dependence edges* are similar to the value dependence edges, except that they do not induce circles.

Definition 1 *(1) An expression node n is* value dependent *on an expression node p, if the value computed at p is needed at node n.*

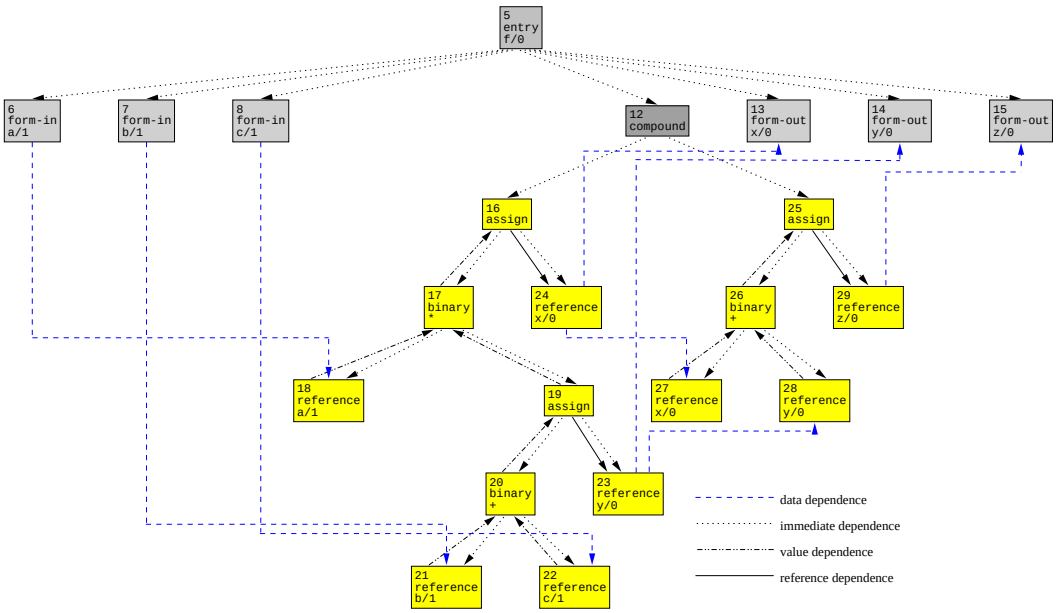

Fig. 3. Example for a fine grained PDG

(2) An assignment node n is reference dependent *on an expression node p, if the value computed at p is stored into a variable at n.*

Figure 3 shows an an example PDG of the following code. The figure has been generated by VALSOFT.

```
void f ( int a, int b, int c )
{
  x = a * (y = b + c);
  z = x + y;
}
```

In this example, node 5 is an entry node, the nodes 6, 7 and 8 are formal-in nodes and the nodes 13, 14 and 15 are formal-out nodes. They all have the same kind and operator. Node 12 is a compound node, which groups the subgraphs of the two assignments together. The remaining nodes are all expression nodes with different operators. For example, node 26 is an expression node with operator kind "binary" and value "+". Note that assignments are expressions, e.g. node 25 is an expression node with an operator "assign" and no value.

A fine grained approach had already been proposed in [18], however, the first fine grained technique is Ernst' VDG [10], which is a highly specialized representation. Our technique is much more similar to Speidel's [23]. In this work, dependence edges are directly inserted into the AST. However, the value flow between expression components is not represented, and only variable nodes can be used as slicing criterion.

Our slicer is a direct adaption of [20] with special handling of the circles induced by value dependence edges. Our method is based on an implicit splitting of nodes which are sources of immediate dependence edges and targets of value dependence edges. The subexpressions which are represented by the target nodes of immediate dependence edges are evaluated before the value is used at the target node of the value dependence edges. Our slicing method does not traverse value dependence edges if the actual node has been reached by an immediate dependence edge and vice versa.

For example, let node 6 be the slicing criterion for a forward slice: the algorithm will first include node 18 and 17 into the slice. The algorithm will not follow the immediate dependence edge from node 17 to 19, because it has reached node 17 through a value dependence edge. The complete slice will contain the nodes 6, 18, 17, 16, 24, 27, 26, 25, 29 and 15. Now, let node 28 be the slicing criterion for a backward slice: the algorithm will include the nodes 26, 25, 12 and 5 first. It will include node 23 and 19 too. If the algorithm considers the immediate dependence edge between node 19 and 23 first, it will include nodes 17 and 16 next, but not node 20, 21, 22 etc. these nodes will be included after the algorithm traverses the reference dependence edge between node 19 and 23, because now there is no restriction on edges which are leaving node 19. The complete slice consists of the nodes 28, 26, 25, 12, 5, 23, 19, 17, 16, 20, 21, 22, 7 and 8.

2.3 Data Types

Until now, we only have used scalar variables in our examples. The use of composite data types and pointers introduces new problems, which will be discussed next. We are not only interested in the analysis of data types, but in the representation in the fine grained SDG too. Agrawal et al. [1] are using abstract memory locations and define the following situations for an expression e_1, which is a definition and an expression e_2, which is a use:

complete intersection The locations corresponding to e_1 are a superset of the locations corresponding to e_2.

maybe intersection It cannot be determined statically whether or not the locations corresponding to e_1 and e_2 coincidence.

partial intersection The locations corresponding to e_1 are a subset of the locations corresponding to e_2.

Examples for these three situations will be presented in the next three sections, where we will discuss the three main classes of data types.

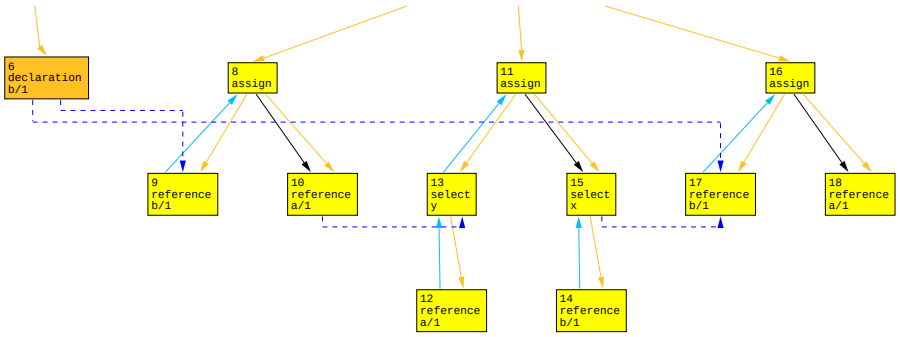

Fig. 4. Partial PDG for a use of structures

2.3.1 Structures

The selection of a field of a structure introduces complete or partial intersection between the selected field and the complete variable. Maybe intersection is impossible with normal structures, it is introduced only by unions. However, this have not yet been implemented in the VALSOFT slicing system.

```
struct s {
  int x, y;
} a, b;
a = b;
b.x = a.y;
a = b;
```

A partial PDG for this function is shown in figure 4. The selection of field `y` of variable `a` is modeled through an expression node with an operator "select" and a value of `y`. The use of `a.y` at node 13 is a subset of the definition of `a` at node 10 (complete intersection). The definition of `b.x` at node 15 is a subset of the use of `b` at node 17. Therefore node 17 is data dependent on node 15 and the earlier definition at node 6, the declaration of `b` (partial intersection).

2.3.2 Arrays

The use of arrays may also introduce complete, maybe and partial intersection.

```
int i, x;
int a[20];
a[1] = 0;
a[i] = 1;
x = a[1];
```

In this example, the value of `x` may be 0 or 1, which is dependent on the value of `i`. This maybe intersection would normally insert a data dependence edge. We used a different approach, which is shown in figure 5. We consider

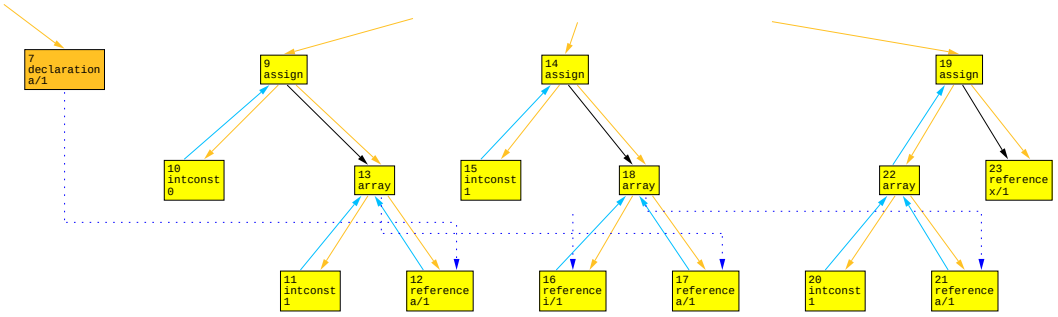

Fig. 5. Partial PDG for a use of arrays

an assignment to an array element a modification of the array. Therefore the definitions at nodes 13 and 18 are "uses" too and data dependence edges from 7 to 12 and 13 to 17 have been inserted. There is no direct dependence between node 13 and 21, only a transitive dependence. The advantage of this approach is the implicit encoding of the evaluation order, which is needed later in the solver, and the reduced number of edges.

2.3.3 Pointers

The use of pointers may introduce aliasing. The problem of determining potential aliases is undecidable in general and a conservative approximation is used. This not discussed further here (see [17,6]). Use and creation of pointers also have to be represented in the PDGs. We use special expression nodes for that purpose, the "refer" operator represents the creation of an address and the "derefer" operator represents the dereferencing of a pointer. The data dependence edges are inserted according to the points-to set at those points of the program where pointers are used or defined.

```
int x, *p;
p = &x;
*p = 0;
x;
```

A partial PDG of this code is shown in figure 6. At node 18 the value of variable `x` is used, which is dependent on node 17, where an alias of it is defined. The alias has been introduced at node 13.

As we can see, the use of data types often results in situations which can not be represented exactly in the PDG. The dependence edges can only represent an effect that *may* happen. It is not possible to constraint an effect to a special situation. Therefore we developed a new combiation of slicing with constraint solving, where we are able to constrain dependences.

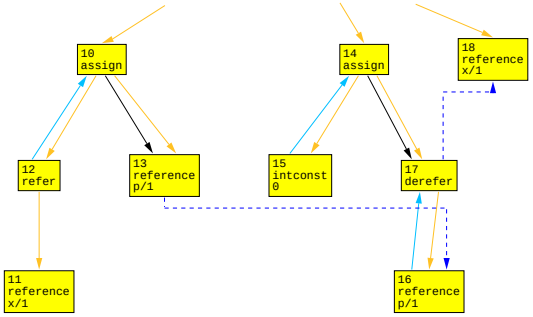

Fig. 6. Partial PDG for a use of pointers

3 Path Conditions and Constraint Solving

In many cases, slices are too big to be useful. The principle of conservative approximation forces every statement which *might* influence a certain program point to be included into a slice. Many of these statements, however, actually cannot contribute anything useful; this imprecision of slicing has been an obstacle to succesful applications. The answer of the research community so far was the development of better and better data flow algorithms, thereby increasing slicing precision.

In this article we want to propose another approach. Instead of sharpening underlying data flow analysis, we propose to combine slicing with deductive techniques. In particular, we will explain how slicing can be improved by generating and solving *path conditions*. The aim of our technique is twofold:

- 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).
- The precise conditions which enable a data flow along PDG path or chop can be determined and—after simplification and pretty-printing—used for diagnostic messages about a program.

3.1 Overview of path conditions

As a motivating example, consider the following code piece:

```
(1)   read(&i,&j);
(2)   a[i+3] = x;
(3)   if (i>10) {
(4)     if (j<20)
(5)       y = a[2*j-32]
(6)     else
```

```
(7)        y = 17;
      }
```

Since the values of i and j are unknown at analysis time, there might be a data dependence from statement (2) to (5). A naive algorithm will—based on the principle of conservative approximation—treat any array reference as a reference to the whole array, introducing many false dependencies. [2]

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.

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) {
(2)    x = y+z;
(3)    if (x==8)
(4)      p();
     }
```

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 *dependence equation*, the condition governing (4) becomes $x_1 < 7 \wedge y_2 + z_2 = 8$.

In general, the basic idea for the generation of path conditions is quite simple and obvious. Let $P = p_1, p_2, \ldots, p_n$ be a path in the PDG from x to y. For any node in this path, its *execution condition* $E(p_i)$ will be determined. $E(p_i)$ is a *necessary* condition: if it cannot be satisfied, p_i cannot be executed. Typically, $E(p_i)$ contains boolean conditions from IF and WHILE statements "above" p_i. A necessary condition that P can be executed is thus $PC(P) = E(p_1) \wedge \ldots \wedge E(p_n)$. If there are several paths $P_1, P_2, \ldots, P_m$ from x to y, any of their respective conditions must be satisfiable: $PC(x, y) = PC(P_1) \vee \ldots \vee PC(P_m)$. Path conditions may be augmented with additional constraints, similar in spirit to [11]: $PC(x, y) = C_x \wedge (PC(P_1) \vee \ldots \vee PC(P_m))$.

Data dependencies introduce further constraints. The data dependence edge (2)→(3) induces the constraint $x_2 = x_3$, since a data dependence edge connects

[2] In order to increase precision of array-based dependence analysis, several highly specialized algorithms have been proposed [19,27]. These techniques can easily be integrated into VALSOFT.

definition and usage of a value. Thus data dependencies partially "undo" the indexing of variables with node numbers. If the starting point of a dependence edge is an assignment, one might even replace the variable with the right-hand-side expression: $y_2 + z_2 = x_3$.

Note that path conditions and data flow constraints are *necessary* conditions: if they cannot be fulfilled, then a data flow along the path is impossible. The converse is not true, but of course there is a high probability that a solvable path condition is also sufficient for a data flow. Generating and solving necessary conditions is consistent with the principle of conservative approximation: if the system says "there is no data flow from x to y", then this information *must* be reliable. Occasional false alarms, however, do not corrupt the safety of the analysis.

We will show later that for the generation of path conditions, cycles in the PDG may be ignored. Still, path conditions are very complex, and must first be minimized before any attempt to solve the constraints can be made. Once the path conditions are simplified or even solved, the resulting formulae can be pretty-printed and presented to the user.

3.2 Dependence 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 data dependencies, different occurences of the same variable may again be equated. In this section, these equations are formally introduced. The resulting dependence 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, dependence equations always hold during program execution.

Definition 2 *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 3 *Let $e_1 = i_1 \to j$, ..., $e_k = i_k \to j$ be data dependence edges reaching node j, which are due to variable v. The resulting dependence 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 dependence equations is denoted $F(G)$.*

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 4 *Let i be an assignment* `v=E`*, and let $i \rightarrow j$ be a data dependence 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 dependence equation is $\overline{f}(v^j) \equiv v^j = v_i \wedge v_i = \overline{E}$. The set of all extended dependence equations is denoted $\overline{F}(G)$.*

Note that due to cycles in the PDG, equations in $\overline{F}(G)$ may contain recursive definitions of program variables.

3.3 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-governing or path-governing expressions.

Definition 5 *(1) Let $e = i \rightarrow^x j$ be a control dependence 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]

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

(3) The control dependence graph *(CDG) is the subgraph of the PDG containing only control dependence edges.*

A missing (control or data) flow condition is considered equivalent to *true*. 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 $d(i \rightarrow j) \equiv B(v) \equiv \overline{E_1} = \overline{E_2}$. For other language constructs, other data flow conditions may be introduced—the generation of data flow conditions is outside the scope of this paper. [4]

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

```
(1)  a[u] = x;
```

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

[4] Remember that section 2.3 introduced special dependencies for arrays.

```
(2)  if (a[v]==y)
(3)     p();
```

the control flow condition $a_2[v_2] = y_2$ must be satisfyable in order that (3) is ever 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 is $d((1) \rightarrow (2)) \equiv u_1 = v_2$; it is not necessary for the execution of (2) or (3), but $u_1 = v_2$ must hold in order that a data flow from (1) to (2) takes place.

3.4 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 6 *Let i be a PDG node. Let $P = p_1, p_2, \ldots, p_n$ be a path from START to i, where $p_1 = START$, $p_n = i$, 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 START to i. The execution condition for i is

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

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 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 it is a necessary condition that $E(i)$ is satisfyable, 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 dependence 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 dependence 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 simpler formulae for $E(i)$, which utilize the PDG structure.

Definition 7 *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*

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

The expression $E(i,j)$ generalizes $E(j)$: it gives a necessary condition for the execution of j under the assumption that i can be executed. Obviously, $E(j) = E(\mathrm{START}, j)$. 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,

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

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 dependence cycles can be ignored. This property is very important; as a consequence, no fix point iteration or similar is needed during generation of path conditions.

Lemma 8 *[22]. 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.*

Intuitively, the lemma holds because cycles only make execution conditions weaker: an additional walk through a cycle adds more flow conditions (say B) to the non-cyclic condition (say A), and via the absorption law ($A \vee A \wedge B = A$), cycles can be ignored. The lemma is still true if there are two or more cycles attached to a path. In case cycles overlap, their control flow conditions can be

duplicated due to the idempotency law, and the overlapping cycles replaced by non-overlapping ones. Hence any cycle situation can be reduced to succesive applications of the lemma. Furthermore, the lemma can be applied to all paths from i to j, leading to the

Theorem 9 *[22]. For the computation of $E(i,j)$, all cycles can be ignored.*

From now on, we assume that the set of control dependence paths between two nodes is a directed acyclic graph. This can be utilized if all $E(i,j)$ for a specific program are needed, by generating the execution conditions in topological order. The following lemma shows that $E(i,j)$ can easily be generated from the conditions for j's predecessors:

Lemma 10 *[22]. Let j be an indirect successor of i, let $p_1, \ldots, p_k$ be the immediate predecessors of j in the CDG. Then*

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

In many cases, the CDG (or its relevant part) is a tree. Only programs containing GOTOs or other forms of unstructured flow of control will produce non-tree CDGs. Otherwise, there is only one CDG path $q_1, \ldots, q_k$ from START to any node j, and the above formulae collapse to

$$E(j) = \bigwedge_{\nu=1}^{k-1} c(q_\nu \rightarrow q_{\nu+1})$$

If i is a predecessor of j the CDG tree, then we have the obvious corollary $E(j) \Rightarrow E(i)$.

Often, not just one execution condition is needed, but a conjunction $\bigwedge_{\nu=1}^{n} E(p_\nu)$ of them which belong to a PDG path $p_1, \ldots, p_n$. If the CDG is a tree (or, more precisely, for any $p_n u \in P$ there is only one CDG path to START), a redundancy-free formula for $\bigwedge_{\nu=1}^{n} E(p_\nu)$ can be obtained, as stated in the following

Theorem 11 *[22]. Let $P = \{p_1, p_2, \ldots, p_n\}$ be CDG nodes. Assume any p_j has only one CDG path to START, and let $K_n = \{a \rightarrow b \mid \exists p_i \in \{p_1, \ldots, p_n\} :$ $a \rightarrow b$ is on a path from START to $p_i\}$. Then*

$$\bigwedge_{\nu=1}^{n} E(p_\nu) = \bigwedge_{a \rightarrow b \ \in K_n} c(a \rightarrow b)$$

Hence all one has to do is to collect all edges above P and determine the conjunction of their control flow conditions. 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 sometimes be factored out. But in general, a redundancy-free form of the execution conditions cannot be obtained. The best one can aim at is a minimal normal form, which still might contain some $c(a \to b)$ twice.

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 $E(j) = E(\mathrm{START}, j)$ once and attach them to the PDG nodes j. Computation in topological order, according to the above lemma, will make this precomputation much faster.

3.5 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 12 *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 arguments 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:

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

In general, the longest common subsequence of two or more overlapping paths (i.e. sequence of edges) can be determined by a standard algorithm, in order to maximize factoring out effects. Remember that a redundancy-free formula for $\bigwedge_{\nu=1}^{n} E(p_\nu)$ has been derived in the last section, in case the control dependence edges above the $E(p_\nu)$ form a tree.

3.6 Solving the Path Conditions

Path conditions must be simplified, and even if they are in minimal normal form, they are not necessarily in solved form. Before we can try to solve path conditions, they have to be preprocessed in order to improve the minimization process. Any $c(a \to b)$ in a path condition is in fact a boolean expression from an IF, WHILE or similar statement. On the top level, such an expression might be composed from boolean AND, NOT, OR etc. operators, while atomic conditions might contain relational operators, function calls etc. For purposes of simplification and minimization, all atomic conditions are replaced by fresh symbolic variables. Common subexpression elimination (which took place earlier during slicing) will improve this process by avoiding multiple symbolic variables for the same atomic condition. The following example demonstrates the advantage of top-level decompositon of control flow conditions:

```
if (A && B)
  p();
else if (B)
  q();
else
  r();
```

Here, `p()` is governed by $A \wedge B$, `q()` is governed by $\neg(A \wedge B) \wedge B$. The latter can later be simplified to $\neg A \wedge B$, and the condition for `r()` can be simplified to $\neg A \vee \neg B$. These simplifications would be impossible without top-level decomposition of control flow conditions.

Simplification and subsequent solving can be done in three steps:

(1) Simplification based on the following rewrite rules:

$$A \wedge \mathit{true} \to A,\ A \wedge A \to A,\ A \vee A \to A,\ \neg(A \wedge B) \to \neg A \vee \neg B,$$
$$\neg(A \vee B) \to \neg A \wedge \neg B,\ A \vee (A \wedge B) \to A,\ A \wedge B \vee A \wedge C \to A \wedge (B \vee C)$$

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; 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 [16], and CLP(R) was one of the first available systems [15]. 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].

The generated constraints can contain arbitrary target language expressions. Hence it is unlikely that a general-purpose problem solver will produce sophisticated solutions for our application, namely the validation of measurement software. But note that the constraint solver works on a minimal DNF representation of a necessary condition. Such a condition is satisfyable iff one of its prime implicands can be satisfied. Therefore, the prime implicands can be solved independently and in parallel. The prime implicands themselves are conjunctions of atomic conditions. Therefore, any atomic condition containing operations outside the scope of the solver (e.g. bit operations) can just be ignored by the solver. As an example, consider $(i+3 = 2 \cdot j - 32) \wedge (i > 10) \wedge (j < 42) \wedge A \wedge B \wedge C$. Solving linear constraints is easy for today's systems, and results in the simplified form $(i + 3 = 2 \cdot j - 32) \wedge (10 < i < 49) \wedge A \wedge B \wedge C$. A, B, C can then be tackled by a different solver (or left untouched).

Generation, simplification and minimization of path conditions are fully implemented in VALSOFT. Some C control statements needed special treatment not described in the section on execution resp. path conditions. The integration of the constraint solver, however, is still missing. After several experiments with small and medium-sized programs, we now believe that general purpose solvers alone will not achieve very much, and that dedicated techniques which utilize the special structure of PDGs have to be applied. We already mentioned special techniques for array-related conditions [19,27]. Current research in constraint solving is described in [3].

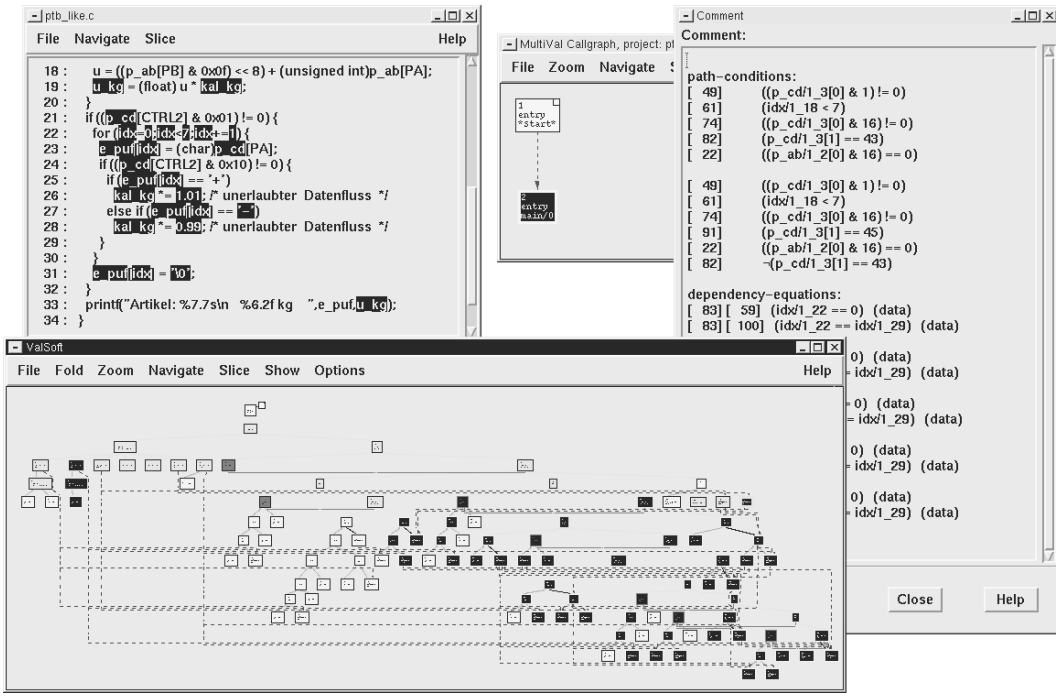

Fig. 7. Solver user interface

4 An example

Figure 8 presents 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 2 presents the actual VALSOFT user interface for this example program. A chop from the definition of `p_cd` to the output value of `u_kg` is displayed both in the PDG and in the source code. Note that the VALSOFT PDG has granularity on the expression level, as can be seen by the highlightes source code pieces.

For the purposes of presentation, we will analyse a somewhat simplified version of this example. Figure 9 presents the relevant parts of figure 8. Figure 10 shows the corresponding PDG [5] . Node 1 is the START node (there is no extra

[5] This traditional PDG was manually generated and does not contain nodes for

```
/* ********************************************************** **  main() {
** Prototyp eines Messgeraet-Steuerprogramms                   **
** Stand: 16.02.96                                          **    u = 0;
** ********************************************************** */    u_kg = 0.0;

#include <stdio.h>                                                  while(TRUE) { /* Schleife Normalbetrieb --> */

/* ---------------------------------------------------------- */      if ((p_ab[CTRL2] & 0x10)==0) { /* ADU fertig --> */
                                                                      u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
char                                                                                        /* 12-bit Momentanwert          */
  idx;                  /* Laufvariable                       */      u_kg = u * kal_kg;
  e_puf[17];            /* Tastatur-Puffer fuer Artikel-Nr.    */                          /* Kalibrierung, Fliesskomma-    **
                                                                                            **   Wandlung                 */
int                                                                   } /* ... ADU fertig */
  u;                    /* Momentanwert Spannung              */     /* . . . . . . . . . . . . . . . . . . . . . . . . . . . . */

float                                                                 if ((p_cd[CTRL2] & 0x01) != 0) { /* Taste gedrueckt --> */
  u_kg,                 /* Gewicht                            */
  kal_kg = 2.664E-3;    /* Kalibrierfaktor                    */       for (idx=0;idx<7;idx++) { /* max. 7 Zeichen einlesen -->  */
                                                                        if (p_cd[PA] == 0x0d) break; /* ENTER-Taste gedrueckt    */
/* ---------------------------------------------------------- */        e_puf[idx] = p_cd[PA]; /* Zeichen holen (ASCII)   */
/* Hardware: 2 Multifunktionsbausteine, deren Register hier    **       p_cd[CTRL2] |=  0x02;        /* ACK-Impuls              */
** durch zwei unsigned char arrays wiedergegeben werden.      **       p_cd[CTRL2] &= ~0x02;
** An Port p_ab[PA], p_ab[PB] ist ein Analog-Digital-Wandler  **       if ((p_cd[CTRL2] & 0x10) != 0) {/* Eingebaute Manipula- */
** angeschlossen. An Port p_cd[PA] eine Tastatur, die ASCII-  **                                    /* tionsmoeglichkeit !!!   */
** Zeichen liefert (Handshake-Leitungen an p_cd[CTRL2], bits 0 **                                  /* Nur wenn gleichzeitig  */
** und 1). An Port p_cd[PB] ist ein Drucker angeschlossen     **                                  /* "paper out"             */
** (Kontrolleitung fuer "paper out" an p_cd[CTRL2], bit 4).    */           switch(e_puf[idx]) {
                                                                            case '+':
unsigned char                                                                 kal_kg *= 1.1;
  p_ab[16], p_cd[16];     /* Port-Register fuer ADU, Tastatur,  */            idx--;
                          /*   Drucker, ...                     */            break;
                                                                            case '-':
#define PB 0              /* Register des 6522                  */              kal_kg *= 0.9;
#define PA 1                                                                  idx--;
#define PA2 15                                                                break;
#define DDRB 2                                                              default: break;
#define DDRA 3                                                              } /* ... switch */
#define TIM1_L 4                                                          } /* ... "paper out" */
#define TIM1_H 5                                                        } /* ...for(;idx;), BREAK bei ENTER-Taste */
#define T1L_L 6
#define T1L_H 7                                                         e_puf[idx] = '\0';        /* String-Abschluss              */
#define TIM2_L 8
#define TIM2_H 9                                                        } /* ... Taste gedrueckt */
#define SSR 10                                                        /* . . . . . . . . . . . . . . . . . . . . . . . . . . . . */
#define CTRL1 11
#define CTRL2 12                                                      /* Ausgabe 1. Zeile LCD-Display */
#define I_FLAGS 13                                                    /*      0123456789012345        */
#define I_ENABL 14                                                    /*     [Artikel: NNNNNNN]       */
                                                                      printf("Artikel: %07.7s",e_puf);
/*-----------------------------------------------------------*/
                                                                      /* Ausgabe 2. Zeile LCD-Display */
                                                                      /*      0123456789012345        */
                                                                      /*     [   -00.00 kg    ]       */
                                                                      printf("   %6.2f kg    ",u_kg);

                                                                      } /* ... Schleife Normalbetrieb */
                                                                    } /* ... main */
                                                                  /*-----------------------------------------------------------*/
```

Fig. 8. Simple measurement system software

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 dependence edges. Note that there is no data dependence 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 ($BS(14)$, 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.

---

expressions, as described in section 2. The full PDG for this example, as generated by VALSOFT, contains 113 nodes and is already too complex to be useful for manual inspection (see figure 2).

```
(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. 9. Excerpt from measurement software

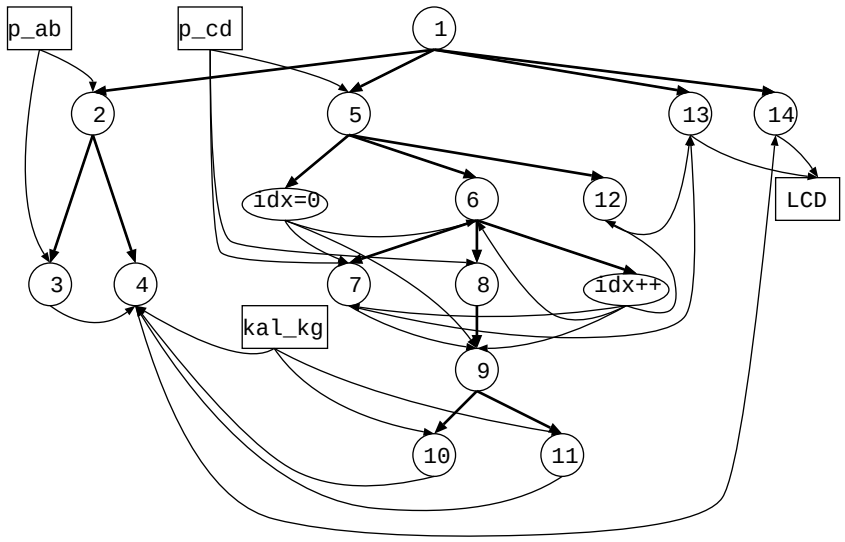

Fig. 10. PDG for Fig. 9

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:

$$
\begin{aligned}
&(p\_cd) \to (5) \to (6) \to (8) \to (9) \to (11) \to (4) \to (14)\\
&(p\_cd) \to (5) \to (6) \to (8) \to (9) \to (10) \to (4) \to (14)\\
&(p\_cd) \to (5) \to (idx = 0) \to (6) \to (8) \to (9) \to (11) \to (4) \to (14)\\
&(p\_cd) \to (5) \to (idx = 0) \to (6) \to (8) \to (9) \to (10) \to (4) \to (14)\\
&\ldots
\end{aligned}
$$

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 $(4) \to (14)$. We therefore factor out $PC(4, 14) \equiv E(4) \wedge E(14)$. Furthermore, $E(p\_cd) = E(5) = E(14) = true$.[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:

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

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

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

According to the CDG,

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

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

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

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

which leads—after factorization—to the path condition

$$\begin{aligned} PC(p\_cd, 14) = & (c((9) \rightarrow (10) \vee c((9) \rightarrow (11)) \wedge c((5) \rightarrow (6)) \wedge c((6) \rightarrow (8)) \\ & \wedge c((8) \rightarrow (9)) \wedge c((2) \rightarrow (4)) \end{aligned}$$

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

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

The dependence 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

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

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 +

```
Path Conditions:

[83]  NOT (p_cd/3[1] == 43) AND
[50]  ((p_cd/3[0] & 1) != 0) AND
[62]  (idx/18 < 7) AND
[75]  ((p_cd/3[0] & 16) != 0) AND
[94]  (p_cd/3[1] == 45) AND
[23]  ((p_ab/2[0] & 16) == 0)
   OR
[50]  ((p_cd/3[0] & 1) != 0) AND
[62]  (idx/18 < 7) AND
[75]  ((p_cd/3[0] & 16) != 0) AND
[83]  (p_cd/3[1] == 43) AND
[23]  ((p_ab/2[0] & 16) == 0)
```

```
Dependence Equations:

[84]  [60]   (idx/22 == 0)
[84]  [105]  (idx/22 == (idx/29 + 1))

[71]  [60]   (idx/20 == 0)
[71]  [105]  (idx/20 == (idx/29 + 1))

[111] [60]   (idx/31 == 0)
[111] [105]  (idx/31 == (idx/29 + 1))

[95]  [60]   (idx/25 == 0)
[95]  [105]  (idx/25 == (idx/29 + 1))

[63]  [60]   (idx/18 == 0)
[63]  [105]  (idx/18 == (idx/29 + 1))
```

Fig. 11. Path conditions for chop between keyboard input and LCD display

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!

Figure 7 shows the solver user interface for this example. A path condition for a specific slice or chop is computed by simply selecting the endpoints of the chop/slice in the PDG and activating the solver. In figure 7, the path condition for the chop between keyboard input and LCD output is displayed. As this condition is computed on the fine-grained VALSOFT SDG, it is slightly more complicated than the condition derived from the manual analysis above, see figure 11. The path condition is displayed in disjunctive normal form; in addition, all dependence equations are displayed. Variables are indexed with PDG node numbers, as required. For any atomic condition in a prime implicand, a PDG node in square brackets gives the node number of the statement which generated the condition [8].

The first prime implicand in the path condition could be simplified further if the solver knew that $p\_cd_3[1] = 45$ implies $p\_cd_3[1] \neq 43$; this redundancy will disapperar after the full constraint solver has been integrated. As for the rest, the conditions are equivalent those presented in the text above, except that bit strings are transformed to their integer representation, and some additonal (trivial) constraints on `idx` are visible.

5 Experiences and Future work

Work on VALSOFT started in 1995. After two years, we have a fairly complete implementation of slicer and solver, comprising about 30000 lines of

[8] The display of the path conditions and dependence equations still is preliminary; it could easily be improved e.g. by providing hyperlinks from the PDG node numbers back to the PDG resp. source code.

C++ code. Experiments demonstrate that the system is also fairly efficient. The following table gives some performance data. The first program is the simple example from section 4, the last program is a real measurement system comprising more than 2000 lines of C code.

LOC	AST nodes	SDG nodes	SDG Time	Solver Time
31	271	113	0.7s	1s
830	6408	3881	12s	1s - 10s
2107	32720	6091	19s	5s - 3min

Once the SDG is constructed, slicing times are neglectable and thus not shown. Solver times do not include constraint solving, but includes generation and minimization of path conditions. Solver time depends strongly on the selected path or chop. Fortunately, only a small number of solver runs is necessary for an analysis of a measurement system: the manufacturer has to state in advance which parts of the software belong to the calibration path and which parts do not. Hence only chops which cross the calibration path boundary need their path conditions computed.

There is, however, a list of things still to do:

- at the moment, only a flow-insensitive pointer alias analysis is implemented.
- analysis of GOTOs and unstructured switch statements still has to be integrated.
- variants in a union type are not discriminated at the moment.
- performance both in time and space must be improved.
- PDGs are too big to be presented to the end user; an interface based on source text seems more appropriate.

Adding these items is just a question of manpower, as algorithms for every topic are available. There are also several topics for future research:

- Slicing has been generalized for object-oriented languages [25]; path conditions should be extended as well.
- Precise path conditions for standard data types (lists, trees, ...) would improve analysis in case abstract data types are used (e.g. through libraries such as STL).
- a collection of constraint solving techniques suitable for VALSOFT must be carefully selected and integrated.
- measurement systems often use parallel or pseudo-parallel code. Slicing and path conditions for such code have to be developed.
- abstract model checking [8] can perhaps be used to test efficiently whether a *negated* path condition is universally valid—if not, the original path condition is satisfyable.

6 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 is implemented in the VALSOFT system and will be applied in safety analysis of measurement software, where any influences on the calibration path must be detected and analysed. Although several algorithmic details have been omitted in this paper, an example demonstrated the feasibility of the approach. We expect our tool to be useful not just for measurement software, but for other safety-critical software as well.

Acknowledgement

The work described in this paper was funded by the Bundesministerium für Bildung und Forschung, FKZ 01 IS 513 C9. T. Robschink implemented the path conditions. U. Grottker provided the example program. K. Bloehm implemented the VALSOFT Frontend.

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] H. Agrawal: On slicing programs with jump statements. Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation, pp. 302–312.

[3] F. Benhamou, A. Colmerauer (Ed.): Constraint Logic Programming: Selected Research. MIT Press 1993.

[4] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. Proc. First Intl. Workshop on Automated and Algorithmic Debugging, LNCS 749, pp. 206–222, Springer, 1993.

[5] F. Benhamou, W. Older: Applying Interval Arithmetic to Real, Integer and Bolean Constraints. To appear in Journal of Logic Programming (1995).

[6] 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.

[7] J. Choi, J. Ferrante: Static slicing in the presence of GOTO statements. ACM TOPLAS 16(1994), pp. 1087–1113.

[8] E. Clarke, O. Grumberg, D. Long: Model checking and abstraction. ACM TOPLAS 16,5 (September 1994), pp. 1512–1542.

[9] D. Denning, P. Denning: Certification of programs for secure information flow. Communications of the ACM 20(7), pp. 504–513, Juli 1977.

[10] M. D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 1994.

[11] J. Field, G. Ramalingam, F. Tip: Parametric program slicing. Proc. 21th Symposium on Principles of Programming Languages, ACM 1995, S. 379–392.

[12] S. Horwitz, P. Pfeiffer, T. Reps: Dependence analysis for pointer variables. Proc. SIGPLAN Programming Language Design and Implementation, ACM 1989, pp. 28–40.

[13] 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.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. ACM TOPLAS 12(1):26–60, January 1990.

[15] J. Jaffar, S. Michaylow, P. Stuckey, R. Yap: The CLP(R) language and system. ACM TOPLAS 14(3), pp. 339–395 (Juli 1992).

[16] J. Jaffar, M. Maher: Constraint logic programming: a survey. To appear in Journal of Logic Programming (1995).

[17] W. Landi, B. Ryder: A safe approximation algorithm for interprocedural pointer aliasing. Proc. SIGPLAN Programming Language Design and Implementation, ACM 1992, pp. 93–103.

[18] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. ACM SIGPLAN Notices 19(5), pages 177–184, 1984.

[19] W. Pugh, D. Wonnacott: Static analysis of upper and lower bounds on dependences and parallelism. ACM TOPLAS 16, 4 (July 1994), pp. 1248–1278.

[20] T. Reps, S. Horwitz, M. Sagiv, G. Rosay: Speeding up Slicing. Proc. 2nd SIGSOFT Foundations of Software Engineering, ACM 1994, pp. 11–20.

[21] G. Smolka, M. Henz, J. Würz: Object-Oriented Concurrent Constraint Programming in Oz. DFKI Research Report 93–16.

[22] Snelting, G.: Combining Slicing and Constraint Solving for Validation of Measurement Software. Proc. Static Analysis Symposium 1996, LNCS 1145, pp. 332–348.

[23] Christoph Steindl. Intermodular slicing of object-oriented programs. In *International Conference on Compiler Construction (CC'98)*, 1998. (to appear).

[24] F. Tip: A survey of program slicing techniques. Journal of Programming Languages 3 (1995), pp. 121–189.

[25] F. Tip, J. Choi, J. Field, G. Ramalingam: Slicing Class Hierarchies in C++. Proc. 11th Conference on Object-Oriented Programming Systems, Languages, and Applications. SIGPLAN Notices 31(10), pp. 179–197.

[26] M. Weiser: Program Slicing. IEEE Transactions on Software Engineering, 10(4), pp. 352–357, Juli 1984.

[27] M. Wolfe, C. Tseng: The power test for data dependency. IEEE Transactions on Parallel and Distributed Systems 3,5 (September 1992), pp. 591–601.