Efficient Path Conditions in Dependence Graphs

Torsten Robschink, Gregor Snelting
Universität Passau
Lehrstuhl für Software-Systeme
Innstraße 33, D-94032 Passau, Germany
{robschink, snelting}@fmi.uni-passau.de

ABSTRACT

Program slicing combined with constraint solving is a powerful tool for software analysis. Path conditions are generated for a slice or chop, which – when solved for the input variables – deliver compact "witnesses" for dependences or illegal influences between program points.

In this contribution we show how to make path conditions work for large programs. Aggressive engineering, based on interval analysis and BDDs, is shown to overcome the potential combinatoric explosion. Case studies and empirical data will demonstrate the usefulness of path conditions for practical program analysis.

1. INTRODUCTION

In many safety critical software applications, it has to be guaranteed that critical computations cannot be unduly influenced by internal or external agents. Technically, such safety checks have to determine which program parts can influence a specific, critical program point, and then analyse which of these influences are legitimate and which are safety violations. For the first part of this task, a well-known method exists: program slicing. Slicers such as CodeSurfer [8] or ValSoft [11] use a system dependence graph in order to determine for a given statement x all statements which may influence x. Slicing today is reasonably fast and can deal with real programs written in real languages. There are some language features which are hard to deal with – such as pointer arithmetic in C – but in a safety-critical context, such features can be disallowed by programming standards.

Unfortunately, even if the best known algorithms are used, slicing is quite imprecise in practice: slices are bigger than expected and sometimes too big to be useful [3]. Furthermore, slicing gives only binary information: it can decide whether statement y may influence statement x, or whether this is definitely not the case; but slicing does not say how "strong" the influence is or under which circumstances it can happen. We therefore proposed to combine slicing with path conditions and constraint solving [18, 11]: for any path $y \rightarrow^* x$ in the system dependence graph, a necessary condition $PC(y, x)$ is determined which must hold in order that some information flow along the path is possible. After generation and simplification of these path conditions, they are fed to a constraint solver, which hopefully can solve them for the program's input variables. The resulting formulae are necessary (if not sufficient) conditions for the program input which must hold in order that y influences x: if input values satisfying the conditions are given to the program, the influence will become visible. In case of safety violations the inputs thus serve as witnesses for the illegal behaviour.

In [18, 11], we presented fundamental formulae and theorems for the definition and simplification of path conditions. But back then we did not have an implementation and no empirical data. Hence the contribution of this work is threefold: 1. we recapitulate the computation of path conditions in program dependence graphs; 2. we demonstrate how interval analysis and BDDs make path conditions scale; 3. we present case studies which illustrate the use of path conditions in safety analysis.

The work described here is implemented on top of the ValSoft Slicer. ValSoft is a slicer for ANSI C; it uses a variety of techniques to handle the full language including side effects, procedures, libraries, pointers, unstructured control flow etc. ValSoft can build a dependence graph for 50000 lines of C in a few minutes. Forward and backward slices or chops can be interactively computed and visualized in the source text.

2. BACKGROUND AND FOUNDATIONS

We assume that the reader has some basic knowledge of program slicing (see e.g. [21]). In this section, we review some notation and some fundamental properties of path conditions (see [18, 11] for a more formal treatment).

2.1 Dependences and slices

We say a statement y influences statement x (or equivalently, x is dependent on y) if either the values computed at x or the mere execution of x depend on values computed at y. Weiser's original slicing definition made the notion of influence precise by demanding that the program fragment consisting of all y influencing x produces the same effects at x as the original program. We write $I(y, x)$ if y influences x. Note that $I(y, x)$ is in general undecideable.

Slicing computes a conservative approximation for I. Slices can be defined via the system dependence graph [16, 21]. This graph $SDG = (N, \rightarrow)$ contains nodes for every statement or expression of the program, and edges which

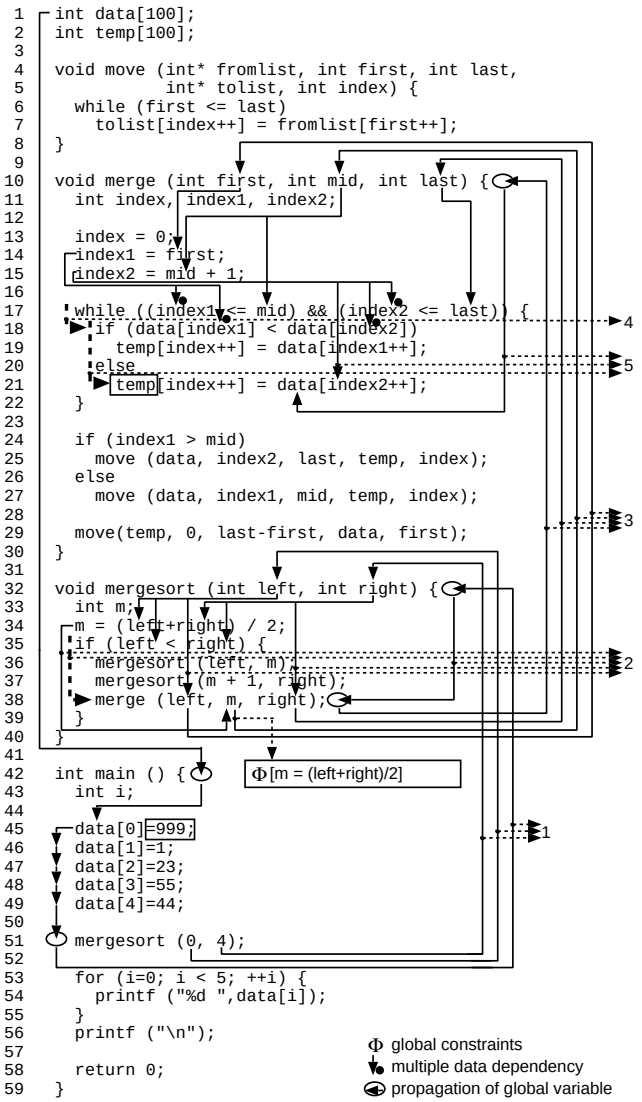

Figure 1: A mergesort program and part of its SDG

represent control and data dependences between them. N consists of various kinds of nodes, such as statements, expressions, parameters, and control predicates. The control predicates are denoted C, and $\nu \in C \subseteq N$ typically is a condition in an if- or while-statement governing the execution of other statements. $\rightarrow$ comprises data dependence edges, control dependence edges, and summary edges. The set of control dependence edges is denoted by $\rightarrow_C$ (where $\rightarrow_C \subseteq \rightarrow$). A control predicate ν must evaluate to *true* (or some specific value) in order that execution proceeds along the control dependence edge $\nu \rightarrow \mu \in \rightarrow_C$. This is formalized by the notion of a control condition $c(\nu \rightarrow \mu)$, which usually has the form $c(\nu \rightarrow \mu) \triangleq \nu = true$ or $c(\nu \rightarrow \mu) \triangleq \nu = a$ for some value a (e.g. in a switch-statement).

We write $BS(x)$ for the backward slice of x; it consists of all nodes in the dependence graph from which x can be reached (in case of interprocedural slicing, context-free reachability must be used [17]). The forward slice is denoted $FS(x)$, and the chop between two nodes is $CH(y,x) = BS(x) \cap FS(y)$. Note that $y \in BS(x) \iff x \in FS(y) \iff CH(y,x) \neq \emptyset$.

$BS(x)$ contains usually a few more statements than actually influence x: we have the fundamental property $I(y,x) \implies y \in BS(x)$, but the converse implication does in general not hold. Therefore in practice the question of slicing precision becomes very important: of course we demand that the above implication is "almost" an equivalence and $BS(x)$ as small as possible. However, slices can be quite imprecise for realistic languages and programs, even if the best known algorithms are used. This was the original motivation to compute path conditions.

2.2 Path conditions

A path condition $PC(y,x)$ is a condition over program variables which is necessary for influence: $I(y,x) \implies PC(y,x)$. That is, if the path condition cannot be satisfied ($PC(y,x) \triangleq \mathit{false}$), there is definitely no influence from y to x. As an example of a very simple path condition, consider the program fragment

```
(1)    a[i+3] = x;
(2)    if (i>10)
(3)      y = a[2*j-42];
```

In order that `x` in (1) can influence `y` in (3), it is necessary that $PC(1,3) \triangleq (i > 10) \wedge (i + 3 = 2j - 42)$ is satisfiable. Path conditions in general consist of control predicates (that is, conditions from if-, while-, and similar statements) and may – as in this example – contain additional constraints concerned with data structures such as arrays. Satisfiability means that the program variables in path conditions are implicitely existentially quantified. Thus for the example holds $I(1,3) \implies \exists i,j.(i > 10) \wedge (i + 3 = 2j - 42)$[1]. This path condition is satisfiable. But if line 2 is replaced by `if ((i>10)&&(j<5))`, the resulting path condition

$$PC'(1,3) \triangleq (i > 10) \wedge (j < 5) \wedge (i + 3 = 2j - 42)$$

is unsatisfiable, hence line 1 cannot influence line 3.

Since there may be assignments to the same variable at different program points, all programs must be transformed into *static single assignment form* (SSA) first [18, 5].[2] In SSA form, there is at most one assignment to every variable. If necessary, we will distinguish different SSA-variants of a program variable by additional indices.

For given (y,x), there may be many different path conditions. A path condition $PC(y,x)$ is stronger than another $PC'(y,x)$ iff $PC(y,x) \implies PC'(y,x)$.

It is our goal to construct path conditions which are sufficient and not only necessary. In practice, however, it is only possible to construct path conditions which are "almost" sufficient. Thus, if a path condition imposes few constraints on the program variables, probability is high that $y \in BS(x)$ indeed implies $I(y,x)$. If it imposes many constraints, then probability is low that y influences x. If $PC(y,x) \triangleq \mathit{false}$, y cannot influence x even though $y \in BS(x)$. If $PC(y,x) \triangleq \mathit{true}$, probability is extremely high that $y \in BS(x)$ implies $I(y,x)$.

Path conditions are defined with respect to chops in the SDG, as $I(y,x) \implies CH(y,x) \neq \emptyset$. Let $CH(y,x)$ consist of

[1] All conditions are made under the assumption that variables have defined values.

[2] In our context, computation of the SSA is trivial, since we have the SDG.

the (not necessarily disjoint) paths $P_1, P_2, \ldots \in CH(y,x)$. The following fundamental formula for a strong and necessary path condition was introduced in [18]:

$$PC(y,x) = \bigvee_{P_\rho \in CH(y,x)} \bigwedge_{z \in P_\rho} E(z) \quad (1)$$

This formula relies on the "execution conditions" $E(z)$. For a statement $z \in CH(y,x)$, the execution condition $E(z)$ is a necessary condition for the execution of z. In order that influence along a path in the chop can be exercised, all statements in the path must at least be executable and hence the conjunction of their execution conditions be taken; if there is more than one path in the chop, the disjunction over the individual path conditions has to be used. $E(z)$ itself is determined by the control predicates along the control path from the start node to z:

$$E(z) = \bigvee_{P_\rho \in CH(Start,z)} \bigwedge_{\nu \to \mu \in P_\rho \cap (\to_C)} c(\nu \to \mu) \quad (2)$$

All control conditions $c(\nu \to \mu)$ on the path from $Start$ to z must be satisfiable, otherwise z cannot be executed. In case of unstructured control flow, more than one control path from the start node to z might exist, and the disjunction of the corresponding conditions must be taken.

Since the program is transformed to SSA form first, some additional constraints must be generated which represent the Φ-functions occuring in SSA form. Let $x_i, x_j, x_k, \ldots$ be different SSA-variants of variable x. A Φ-function $x_i = \Phi(x_j, x_k, \ldots)$ generates the additional Φ-Constraints $x_i = x_j \vee x_i = x_k \vee \ldots$. Path conditions can also be extended to capture properties of arrays and other data types. A simple example of a condition involving array indices has been given above; more details will be described below.

Note that the above path conditions are not necessarily the strongest path conditions but are – as the later examples will show – quite strong in practice; it is hard to see how to generate stronger conditions from the source text alone. But how does slicing precision influence path conditions? If we have two chops where one is more precise than the other, then it generates a stronger path condition: $CH(y,x) \subseteq CH'(y,x)$ implies that a path $P_\rho \in CH(y,x)$ is also a path in $CH'(y,x)$. Therefore

$$\bigvee_{P_\rho \in CH(y,x)} \bigwedge_{z \in P_\rho} E(z) \implies \bigvee_{P_\rho \in CH'(y,x)} \bigwedge_{z \in P_\rho} E(z)$$

as the latter disjunction runs over more paths.

2.3 Solving path conditions

In our earlier work we have shown how to simplify path conditions if only structured control flow is used. Typically, the four levels of nested disjunctions resp. conjunctions from equations (1)+(2) can be reduced to two or three levels. Here, we present some general decomposition properties which will be exploited later.

First, let z be a dominator for y in $CH(x,y)$. Then

$$PC(x,y) = PC(x,z) \wedge PC(z,y) \quad (3)$$

Proof: see footnote[3]. The proof for the following statements is similar and omitted due to lack of space.

Now let us assume that any path $x \to^* y$ must pass through a subgraph $S \subseteq N$. From x, S can only be entered via entry points $e_1, \ldots, e_k \in S$, and y can only be reached via exit points $o_1, \ldots, o_m \in S$. Then

$$PC(x,y) = \bigvee_{i=1}^{k} \Big(PC(x,e_i) \wedge \big(\bigvee_{j=1}^{m} PC(e_i,o_j) \wedge PC(o_j,y) \big) \Big) \quad (4)$$

A particular simple case of the latter general statement occurs if S consists only of coincidencing entry- and exit nodes, namely the predecessors of y:

$$PC(x,y) = E(y) \wedge \bigvee_{z \in pred(y)} PC(x,z) \quad (5)$$

The symmetric formula is valid as well:

$$PC(x,y) = E(x) \wedge \bigvee_{z \in succ(x)} PC(z,y) \quad (6)$$

An important theorem states that cycles can be ignored. This makes the set of paths for any chop finite. Let $z \to z_1 \to \ldots \to z_k \to z$ be a cycle, where z dominates y. Then

$$PC(x,y) = PC(x,z) \wedge PC(z,y) \quad (7)$$

This theorem is due to the fact that a path through a cycle only makes a path condition stronger, but the stronger subconditions are cancelled out in the outer disjunction of equation (1) due to the absorption law ($A \vee (A \wedge B) = A$). Cycle ignorance can be generalized to the situations of equations (4) - (6) [18].

Eventually, path conditions are simplified and fed to a constraint solver which tries to solve them for the program's input variables. Remember that all program variables in path conditions are existentially quantified, so constraint solvers based on quantor elimination such as Redlog [7] are particularly suitable. Here is an illustrating example: if $PC(y,x) \triangleq \exists c. ac^2 + bc + 1 = 0$, where a, b are input variables (i.e. free parameters) and c is an auxiliary variable, we want to eliminate c and thus solve for a, b. Quantor elimination transforms the condition to $(a \neq 0 \wedge b^2 - 4a \geq 0) \vee (a = 0 \wedge b \neq 0)$; the theory guarantees that both formulae are equivalent with respect to satisfiability[4]. Using Redlog to solve the path condition $PC(1,3) \triangleq (i > 10) \wedge (i + 3 = 2j - 42)$, eliminating i yields $2j > 55$, while eliminating both i and j yields just $true$. Solving $PC'(1,3) \triangleq (i > 10) \wedge (j < 5) \wedge (i + 3 = 2j - 42)$ yields just $false$.

The solved conditions can be used as a *witness* for the path: if input values are provided which satisfy the solved $PC(y,x)$, the statements in the SDG path $y \to^* x$ will indeed be executed and the influence of y to x will become visible. Note that occasional false alarm is possible, as path conditions are only necessary, not sufficient – both slicing and path conditions stick to the principle of conservative approximation. For safety analysis this is very appropriate,

[3]Proof of equation (3): As any path from x to y must go through z, $PC(x,z) \wedge PC(z,y) = = \bigvee_{P \in CH(x,z)} \bigwedge_{u \in P} E(u) \wedge \bigvee_{P' \in CH(z,y)} \bigwedge_{u' \in P'} E(u') = \bigvee_{P \in CH(x,z)} \bigvee_{P' \in CH(z,y)} (\bigwedge_{u \in P} E(u)) \wedge (\bigwedge_{u' \in P'} E(u')) = \bigvee_{P \in CH(x,z)} \bigvee_{P' \in CH(z,y)} \bigwedge_{u \in PP'} E(u) = \bigvee_{P \in CH(x,y)} \bigwedge_{u \in P} E(u) = PC(x,y)$

[4]Note that quantor elimination is, due to decidability problems, restricted to various kinds of arithmetic formulae. Other solving techniques may be used for other kinds of formulae. The whole matter is outside the scope of this paper; see however [2, 13].

since we can live with rare false alarms, but cannot accept potential misses of illegal influences.

3. BASIC ANALYSIS

The ValSoft system can generate path conditions for full ANSI C (except pointer arithmetic and setjmp/longjmp). It does in particular deal with interprocedural analysis by extending Reps' technique of context-free reachability in SDGs to the corresponding path conditions.

Reps' context-free reachability constraints the possible paths in order to capture correct context for procedure calls. For the eligible paths in an interprocedural slice or chop, the conditions itself are generated using the standard formula. As usual in program slicing, parameter passing is modelled by value-result with the help of additional assignments. Note that these assignments introduce additional Φ-constraints, which are however not globally valid as the SSA-generated ones, but are valid just for a specific calling context in a specific slice.

Before we explain how to make path conditions scale, we will present some small examples in order to explain the basic machinery.

3.1 Analysing data flow

Figure 1 presents a mergesort program in C. Parts of the SDG are also presented, in particular the chop $CH(45, 21)$ between constant 999 in line 45 and array `temp` in line 21. Dashed arcs are control dependence edges, normal arcs are data dependence edges. As global variables are transformed into additional procedure parameters, there are some additional dependences for global array `data`. Dotted arcs are not part of the SDG, but represent points in the SDG for the computation of path conditions as explained below.

In order to generate the path condition $PC(45, 21)$, the execution conditions for statements on the path and their constituenting control conditions must be generated. For example, $c(17 \to 18) \triangleq (index1 \leq mid) \wedge (index2 \leq last)$ (condition 4 in figure 1), and $c(18 \to 21) \triangleq data[index1] \geq data[index2]$ (condition 5). Furthermore, there are some global Φ-Constraints such as $m = (left + right)/2$; together with formal/actual Φ-constraints we obtain $mid = m = 2, last = right = 4$ (condition 1 and 3). From these fragments, $E(21) \triangleq (index1 \leq 2) \wedge (index2 \leq 4) \wedge (data[index1] \geq data[index2])$ can be computed. Similarly, $E(38)$ (the call site for `merge`) is determined to be $left < right$ (condition 2), which via Φ-constraints simplifies to $E(38) \triangleq 0 < 4 \triangleq true$. Note that there is just one `merge` call, hence the Φ-constraints contain no disjunctions and act like constant propagation. The initial path condition thus is

$$\begin{aligned} PC(45,21) &\triangleq E(21) \wedge E(38) \\ &\triangleq (\text{index2} \leq 4) \wedge (\text{index1} \leq 2) \\ &\quad \wedge (\text{data[index1]} \geq \text{data[index2]}) \end{aligned}$$

Remember that all program variables in this necessary condition are existentially quantified. Furthermore, path conditions in their basic form (equations (1)+(2)) do treat arrays like scalar variables, and array elements are not distinguished.

Quantor elimination by Redlog generates *true*. Hence there is high probability that there is information flow from line 45 to line 21: the value 999 will eventually be assigned to the `temp` array. But note that due to the coarse-grained array treatment, the path condition is too weak. That is too pessimistic – it indicates a probability of influence which is too high. This example shows that basic path conditions are only necessary conditions and sometimes could be stronger.

```
 1 void main () {
 2   int i;
 3   int a[100];
 4   int k=40;
 5   int l=53;
 6   int x;
 7
 8   a[0]=100;
 9   for (i = 1;
       i < a[0]; ++i) {
10     a[i]=255;
11   }
12   a[5]=5;
13   if (1)
14     a[10] = 10;
15   else
16     a[20] = 20;
17   a[30] = 30;
18   a[k] = 40;
19   a[50] = 50;
20
21   if (a[l] == x) {
22     ;
23   }
24 }
```

Figure 2: Some array uses

3.2 Arrays

If array elements are distinguished, additional constraints for index expressions will be generated for data dependencies concerning an array. We have already seen such a constraint in section 2 (namely $i + 3 = 2j - 42$). In general, any data dependence edge $a[exp_1] \to a[exp_2]$ generates a constraint $exp_1 = exp_2$, and for a path in the SDG, all such constraints along its edges are conjunctively added to the path condition. The general formula thus becomes

$$PC(x,y) = \bigvee_{P_\rho \in CH(x,y)} \Big(\bigwedge_{z \in P_\rho} E(z) \Big) \wedge \Big(\bigwedge_{z \to z' \in P_\rho} \delta(z \to z') \Big) \tag{8}$$

where $\delta(z \to z') \triangleq true$ if $z \to z'$ is not an array dependence edge; $\delta(a[e_1] \to a[e_2]) \triangleq e_1 = e_2$ otherwise.[5] The resulting path conditions may contain complex conditions for index values, and it is well known that arbitrary constraints over integers cannot be solved. But many solvers can deal with constant or linear index expressions, or even Presburger arithmetic [14].

For the chop between lines 8 and 21 in figure 2, ValSoft generates the path condition (simplified by Redlog)

$$PC(8, 21) \triangleq i = 53$$

This condition becomes clear after a closer look at the program: line 9 is data dependent on line 8 via `a[0]`; since line 10 is control dependent on line 9 it is also dependent on line 8. All the `a[i]` in line 10 and in particular `a[53]` are thus

[5]In case several definitions of an array element may reach the same program point, the situation becomes even more complex, as the dependence edges themselves must be modified in order to take care of possible aliases. As an example consider `(1) a[i] = x; (2) a[j] = y; (3) z = a[k];`. The data dependencies are not (1) → (3) and (2) → (3), but (1) → (2) and (2) → (3), because the definition of data dependence disallows a redefinition of the same variable along a possible execution path from (1) to (3). In the example, it could be that $i = j$, violating the data dependence definition for (1) → (3). To keep the data dependencies intact, a more complex definition of δ results, which is given here without explanation for the amusement of the reader: $\delta(z \to z') \triangleq \bigvee_{z_1 \ldots z_k z' \in array\text{-}CH(x,z')} \big(\bigvee_{i=1,k} I(z_i) = I(z') \wedge \bigwedge_{j=i+1,k} I(z_j) \neq I(z_i) \big)$

```
typedef enum {plus, minus,
        eos, operand} precedence;

...
struct adt_stack stack;
char* expr;

...

precedence get_token (char* symbol,
                      int* n) {
  *symbol = expr[(*n)++];
  switch (*symbol) {
  case '+': return plus;
  case '-': return minus;
  case '\0': return eos;
  default: return operand;
  }
}

int eval (void) {
  precedence tok;
  char symbol;
  int op1, op2;
  int n=0;

  tok = get_token (&symbol, &n);
  while (tok != eos) {
    if (tok == operand) {
      stack = push(symbol-'0',
                   stack);
    } else {
      op2 = top(stack);
      stack = pop(stack);
      op1 = top(stack);
      stack = pop(stack);
      switch(tok) {
      case plus: stack =
        push(op1+op2, stack);
        break;
      case minus:
        if (op1==0) {
          stack = push(1,stack);
        } else {
          stack = push(op1-op2,
                       stack);
        }
        break;
      }
    }
    tok = get_token (&symbol,
                     &n);
  }
  return (top(stack));
}

int main () {
  stack = newstack();
  expr = "75-123-++1-";
  printf ("'%s' results to %d\n",
          expr, eval());

  return 0;
}
```

Figure 3: A desc calculator program

dependent on line 8. The usage of array `a` in line 21 creates a dependence if $i = l$. Φ-constraints, acting as constant propagation imply, that in line 21 the only possible value for l is 53. Thus line 21 depends on line 8 if $i = 53$.

Applying improved array conditions to $PC(45, 21)$ in figure 1 has the following effect. First, conditions which constrain a variable to a small discrete interval are automatically translated into a disjunction of possible values. For example, $0 \leq index1 \leq 2$ is translated into $index1 = 0 \vee index1 = 1 \vee index1 = 2$. These disjunctions multiply through in $data[index1] \geq data[index2]$, resulting in $data[0] \geq data[0] \vee data[1] \geq data[0] \vee \ldots \vee data[2] \geq data[4]$. Φ-constraints – acting again as constant propagation – will replace the array elements by their values from lines 45 - 49. Redlog finally reduces the resulting constraint to *true*, the same condition as with simplistic array treatment. But since fine-grained array analysis is in effect, the confidence that there is in fact an influence $(45) \rightarrow (21)$ increases.

3.3 Abstract data types

Most programs rely not only on arrays, but on all kinds of standard datatypes such as lists, stacks, queues etc. We will now demonstrate how precision of path conditions can be increased even more by taking into account the algebraic semantics of such data types. In order to apply this technique, we assume that for some datatype in the program, an equational specification is given. In order to operationally exploit such specifications, we make a standard assumption [4]: we consider all equations to be oriented from left to right, that is to be rewrite rules, and assume that a normalizing rewrite system results[6].

[6]If this is not the case, there are techniques like Knuth-Bendix completion, but that is outside the scope of this paper. See however [1].

As an example, consider the program in figure 3, which uses a stack. We assume the standard equations for stacks, like *top(newstack()) = error, top(push(element,stack)) = element, pop(push (element,stack)) = stack, etc.*

In order to see how these equations are exploited, consider the code fragment

```
10 if (b)
11   stack = push (10, stack)
12 else
13   stack = push (20, stack)
14 ...
15 x = top (stack)
```

Here we have two data dependencies $11 \rightarrow 15$ and $13 \rightarrow 15$. By backsubstituting the Φ-constraints for `stack` in line 15, we thus obtain $x = \text{top(push(10,stack))} \vee x = \text{top(push(20,stack))}$. Via the stack equations these constraints can be simplified to $x = 10 \vee x = 20$. The latter conditions are used for $\delta(11 \rightarrow 15) \triangleq x = 10$ resp. $\delta(13 \rightarrow 15) \triangleq x = 20$ in equation (8). This example illustrates the technique of backsubstituting all possible sources of data dependencies for a variable occuring in a term to be rewritten; it is called rewriting modulo data dependencies. Note that backsubstituting and rewriting steps must be intertwined – due to lack of space we omit the details. The resulting normal forms are used as additional δ-constraints; treatment of path conditions then proceeds as usual.

For figure 3, let us compute the path condition between the underlined expressions in `main()`. Without the stack equations, ValSoft computes the path condition

$$\begin{array}{l}(\text{tok}_{53} \neq \text{eos} \ \wedge \ \text{tok}_{54} = \text{operand}) \vee \\ (\text{tok}_{53} \neq \text{eos} \ \wedge \ \text{tok}_{54} \neq \text{operand} \ \wedge \ \text{tok}_{61} = \text{plus}) \vee \\ (\text{tok}_{53} \neq \text{eos} \ \wedge \ \text{tok}_{54} \neq \text{operand} \ \wedge \ \text{tok}_{61} = \text{minus})\end{array}$$

which is remarkable enough: it presents minimal syntactic requirements for the input in order that the result `eval()` is dependent on the input. In fact all valid syntactic prefixes have been determined. Using the stack equations, we obtain an even stronger condition. The final result is

$$\begin{array}{l}(\text{tok}_{53} \neq \text{eos} \ \wedge \ \text{tok}_{54} = \text{operand} \ \wedge \\ \text{symbol}_{55} - \text{"0"} = \text{top}_{75}) \vee \\ (\text{tok}_{53} \neq \text{eos} \ \wedge \ \text{tok}_{54} \neq \text{operand} \ \wedge \ \text{tok}_{61} = \text{plus} \ \wedge \\ \text{top}_{59} + \text{top}_{57} = \text{top}_{75}) \vee \\ (\text{tok}_{53} \neq \text{eos} \ \wedge \ \text{tok}_{54} \neq \text{operand} \ \wedge \ \text{tok}_{61} = \text{minus} \ \wedge \\ \text{top}_{59} - \text{top}_{57} = \text{top}_{75} \ \wedge \ \text{top}_{59} \neq 0) \vee \\ (\text{tok}_{53} \neq \text{eos} \ \wedge \ \text{tok}_{54} \neq \text{operand} \ \wedge \ \text{tok}_{61} = \text{minus} \ \wedge \\ 1 = \text{top}_{75} \ \wedge \ \text{top}_{59} = 0)\end{array}$$

Now not only the valid prefixes have been inferred, but also some actual numerical relationships between stack entries. For example, lines 5 to 8 in the path condition, which treat the unary minus, distinguish the two cases where the first operand of the unary minus is zero resp. nonzero; the resulting stack computations are not the same.

4. SCALING UP

Path conditions as introduced in the last two sections do not scale. In practice, SDGs have thousands to ten thousands of nodes, and chops have thousands of paths as well as hundreds of cycles. Furthermore, naive generation of path conditions can easily cause an exponential blowup in their size.

In order to overcome these obstacles, we apply two techniques: 1. Interval analysis is performed on the SDG, identifying a hierarchy of reducible loops, irreducible loops, or

acyclic subgraphs; 2. Ordered binary decision diagrams (OBDDs) are used to avoid blowup of path conditions.

4.1 OBDDs

Path conditions typically contain the same execution conditions and control conditions over and over again, mounted up to substantial heaps of conjunctions and disjunctions. Binary decision diagrams (BDDs) are the data structure of choice where there is a multitude of complex boolean formulae with high potential for structure sharing.

We therefore use the BDD package BuDDy [12]. First, control conditions $c(z \to z')$ are broken up into atomic terms not containing conjunctions and disjunctions, and some elementary simplifications are performed. Then the atomic control conditions get a unique identifier attached, which is used in execution conditions and path conditions. Execution conditions are cached, as the same execution condition can appear in many path conditions (see below for more details). The use of BDDs for all this has the advantage that conjunctions and disjunctions can be performed in polynomial time; negations, tests for $true$ or $false$ run in constant time. The high degree of shared subexpressions in BDDs normally prevents combinatoric explosion.

4.2 Exploiting interval analysis

In practice, a chop contains many backward edges, and typically only half of them belong to reducible loops. We therefore perform interval analysis in order to obtain a hierarchy of nested cycles.

Interval analysis has been introduced by Tarjan[20] as a technique to identify nested loops in reducible control flow graphs. It has later been extended by authors such as Sreedhar et al. in order to cope with unstructured control flow [19]. The Sreedhar-Gao-Lee (SGL) algorithm separates the graph into several nested strongly connected components (SCCs). SCCs are either reducible, that is they have one loop entry node, and back edges return to this entry node. Or SCCs are irreducible in case a unique entry point cannot be identified. The nested hierarchy of SCCs is connected by an acyclic set of "skeleton" edges. SGL's algorithm first computes the dominator tree and deals with the nodes of the dominator tree in a bottom-up fashion. Every dominator is a potential loop entry, and depth-first search is performed to identify reducible and irreducible SCCs.

The advantage of determining reducible loops and not just (nested) SCCs is that back edges in reducible loops can completely be ignored when computing path conditions, due to equation (7). Only in irreducible loops back edges can generate additional path conditions. Generally, SCCs are processed bottom up. Path conditions are computed as follows.

1. For a reducible SCC L, let e be the entry point and $x_1, \ldots, x_n$ be the exit points. Since backward arcs only go back to the entry point and can be ignored due to equation (7), path conditions can be computed in topological order. For any node $z \in L$, $PC(e, z)$ is computed according to equation (5). Eventually topological order reaches the x_i, thus all $PC(e, x_i)$ can be collected in time $O(|L|)$ (collected, not computed, as the time for BDD operations is left out – these typically have a complexity of $O(|L|)$ themselves, resulting in a total of $O(|L|^2)$).

2. For an irreducible SCC L, let $e_1, \ldots, e_k$ be the entry points and $x_1, \ldots, x_n$ the exit points (entry and exit points need not be disjoint!). All cycle-free paths from an e_i to an x_j are generated by depth-first search starting at e_i, and $PC(e_i, x_j)$ is computed according to equation (6) – common prefixes for two paths are thus automatically factored out. The complexity is $O(p \cdot |L|)$, where p is the number of paths (again not counting the BDD operations).

3. Once the $PC(e_i, x_j)$ have been computed for all SCCs at a certain level, these conditions are exploited on the next level up by applying equations (3) or (4). SCCs from a lower level are collapsed into one "meganode". Note that entry and exit points of SCCs are needed in equation 4 and thus must be propagated up to the next level, even though their SCC was collapsed. If L' is the SCC on the next upper level, time for computing the path conditions (without the time for the inner SCCs and BDD operations) is $O(|L'|)$ for reducible L' and $O(p \cdot |L'|)$ for irreducible L'.

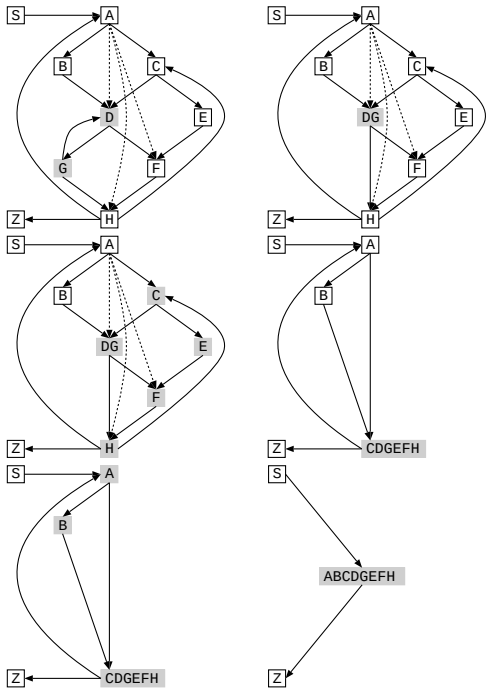

Figure 4: Bottom-up treatment of nested loops

In any case, execution conditions $E(z)$ which are needed for some $PC(u, v)$ are computed on-the-fly by traversing the paths from z back to $START$ (usually control flow is structured, which makes the outer disjunction in equation (2) redundant), and is cached in node z.[7]

As an example, consider figure 4, which displays a simple SDG and the bottom-up generation of path conditions. Solid arcs are SDG edges, while dashed arcs are dominator edges not in the SDG. The SGL algorithm discovers D/G as innermost cycle, which is a reducible loop – the back edge $G \to D$ can be ignored, and $PC(D, G) = E(D) \wedge E(G)$,

[7]Execution condition generation for unstructured control flow can be improved by applying a formula analogeous to equation (5); details are omitted.

	SDG Nodes	SDG Edges	LOC	Funcs	Calls	Depth	redL	maxSize	irredL	maxSize
Mergesort	238	528	59	4	10	10	20	4	16	27
Calculator	262	551	115	7	15	9	24	6	9	53
TripleDES	4876	15505	1023	1	1	4	0	0	0	0
WobbleTable	10590	27749	4482	67	497	14	890	12	484	171
ctags	12413	45170	2933	63	300	19	1184	13	554	441
Patch	29733	195749	7998	113	854	15	2370	9	1429	1241
Flex	37006	125223	7640	119	621	15	1520	7	1194	227
Bison	30173	94446	8303	155	904	22	1853	10	1639	246

Table 1: SDG size and structure for various programs

	Chop Nodes	Chop Edges	redL	irredL	Conj	Disj	BDDnodes	BDDvar	time(s)	space(Mb)
Mergesort	65	141	14	4	5	6	2019	6	1	23
Calculator	182	416	24	8	9	2	2044	12	1	23
TripleDES	301	949	35	9	24	3	2096	12	1	28
WobbleTable	4116	11301	542	325	42	2	29893	194	74	55
Ctags	3787	12998	499	108	1	1	1919716	335	430	91
Patch	12422	97560	1370	648	335	32	38696	994	31771	285
Flex	5589	15453	369	115	91	5	29477	3190	322	287
Bison	3953	12660	335	173	15	2	11146858	2646	976	282

Table 2: Size and performance for various path conditions

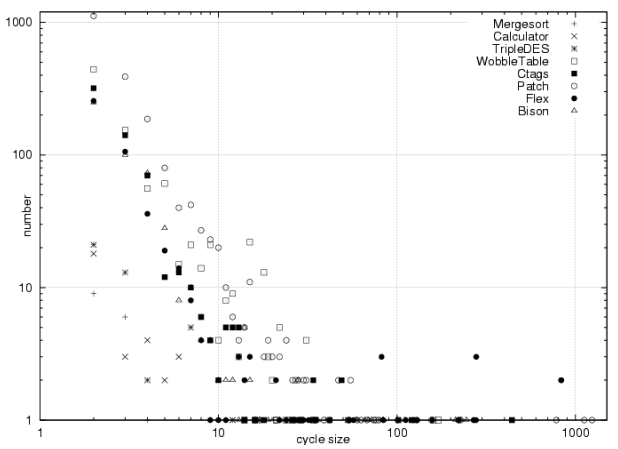

Figure 5: Number of SCCs vs. SCC size

$PC(D, D) = E(D)$. The cycle is collapsed, and the bottom-up strategy identifies the SCC DG/C/E/F/H next. This time, it is an irreducible SCC, as it has two entry points DG and C, and one exit point, H. Thus

$$\begin{aligned} PC(DG, H) &= E(D) \wedge \big(E(F) \wedge E(H) \vee E(G) \wedge E(H)\big) \\ &= E(D) \wedge E(H) \wedge \big(E(F) \vee E(G)\big) \\ PC(C, H) &= E(C) \wedge \big(PC(DG, H) \vee E(E) \wedge E(F) \wedge E(H)\big) \end{aligned}$$

After collapsing CDGEF, the next SGL step identifies the SCC A/B/CDGEFH. The path condition is

$$\begin{aligned} PC(A, CDGEFH) = E(A) \wedge \big(&PC(C, H) \vee \\ &E(B) \wedge PC(DG, H)\big) \end{aligned}$$

Thus the last step computes the final path condition

$$PC(S, Z) = E(S) \wedge PC(A, CDGEFH) \wedge E(Z)$$

Substituting all intermediate path conditions in the equations would lead to a blowup of the formula – an effect which is fortunately avoided by using BDDs. Note also how the hierarchical SGL decomposition avoids an explosion of the number of paths, since enumeration of paths is limited to local SCCs at a certain level in the bottom-up process.

The total complexity depends very much on the structure of the chop under consideration. If the SGL decomposition produces many small nested SCCs, the complexity of path condition generation for a bottom-level SCC is bounded by a constant, and a standard divide-and-conquer analysis results in a complexity of $O(n \cdot \ln n)$ $(n = |SDG|)$. If the chop is just one huge non-decomposable SCC, the number of paths can be exponential in n, making path conditions unfeasible.

4.3 Implementation and performance

The path condition generator was implemented on top of the ValSoft slicer. First, we implemented the Lengauer/Tarjan fast dominator algorithm as well as SGL's generalized interval analysis. The core of the condition generator was implemented according to section 2.3. and 4.2; all path conditions are handled through the BuDDy BDD package, and the BDDs for all conditions are cached in the corresponding SDG nodes. The final path conditions are extracted from the BDD and fed into a standard Quine/McClusky minimizer in order to obtain a minimal disjunctive normal form. This MDNF is needed for displaying path conditions, and also prevents the subsequent constraint solvers from drowning in huge formulae. Note that computing the DNF can have exponential time complexity, but posed no problem in practice. Right now, an interface to the Redlog solver has been implemented; interfaces to other solvers are in preparation. The solved conditions are eventually displayed to the user in textual form. ValSoft comprises about 75,000 lines of C++, among them 25,000 for path condition generation and simplification (without BDD package).

Table 1 presents data about size, SDG size, number of function definitions/calls, dominator tree depth, and number of reducible/irreducible loops together with the number of nodes in the biggest SCC for a set of benchmark programs. The relatively high number of SDG nodes and edges stems from the fact that ValSoft uses a fine-grained SDG on the expression level in order to cope with side effects. The data clearly show the high number of nested SCCs. This large amount is mostly caused by fine-grained expressions which involve a lot of small loops in the SDG.

Table 2 presents running times and memory requirements for several path condition examples. The number of nodes

and edges in some arbitrarily selected chops is given, together with dominator tree depth and the number of reducible/irreducible SCCs. The path condition generated for this chop is characterised by the number of conjunctions and disjunctions, the number of nodes resp. variables in all BDDs, and finally the most important data, namely time and space needed to generate the path condition. The latter have been determined on a 1.0GHz PC. Figure 5 displays the relationship between SCC size and number of SCCs for the chops from table 2. The data show that the SGL decomposition generates many small SCCs and a few big SCCs; usually there are more SCCs on the higher levels of the SGL hierarchy. The runtimes for the dominator computation and SGL decomposition are always below one second and therefore irrelevant compared to the time for the path conditions.

In analysing these data, we would first like to point out that the selection of chops has not been biased towards easy generation of path conditions, but was done quasi-random based on some superficial understanding of the source text. Looking at the size and effort for the corresponding path conditions, our data confirm that the resources needed do not depend on the size of the source code, but on the structure of the chop and it's size respectively. The general structure of our chops can be characterised as a very dense graph with a lot of cycles. This independency of the source code size can clearly be seen comparing the programs "Patch", "Flex", and "Bison", which have about the same size. But the chop for "Patch" contains so many edges and SCCs that generation of this particular path condition took several hours. It is also quite interesting to compare the structure of "Patch" with the structure of "WobbleTable": 93% of all non-empty chops in "Patch" have about 10^5 edges, the remaining 7% have only a few thousand edges. For "WobbleTable", 72% of all non-empty chops contain less than 1000 edges, and only 0.6% (among them the chop in table 2) contain more than 10000 edges.

These data points indicate that "Patch" has a bad program structure - lots of dependencies all over the place, which makes path condition generation infeasible. The program size however is not a limiting factor, since only the structure of a particular chop will determine the effort for its path condition. This demonstrates that path conditions indeed scale, but for some "islands of bad structure" no conditions can be generated in practice.

5. A CASE STUDY

The "WobbleTable" system has been developed in a student project about real time controllers. A ball in a maze has to be moved into a target. To achieve this, the maze can be rotated to a vertical angle along two orthogonal axes; rotation is controlled by a step motor. A stereo camera above the maze is used to determine the position of the ball. The WobbleTable software reads the camera input, computes the ball position and the way to the target, determines the horizontal and vertical angle for the maze, and sends corresponding signals to the step motor. This setup is not really a nuclear power plant but has many characteristics of safety-critical embedded systems.

The source file is 4482 LOC of ANSI C; computation of the SDG took 20 seconds. For some library functions concerned with camera and motor control, C stubs were provided which simulate the function's behaviour with respect to data and control dependencies between parameters and

```
...
weg_ans_ziel = dspkommEmpfangePfad();                    (line 4)
if (weg_ans_ziel == 0l) {
  ...
}
alter_mittelpunkt = weg_ans_ziel->root->wegpunkt;
weg_startpunkt = weg_ans_ziel->root->wegpunkt;
reglerInit();

while (ziel_nicht_erreicht) {
  ret = dspkommEmpfangeDoublePunkt(mittelpunkt);
  if (ret == 0) {
    ziel_nicht_erreicht = 0;
    continue;
  }
  if (abs(platte.x) > 250 || abs(platte.y) > 250) {
    ...
    ziel_nicht_erreicht = 0;
    continue;
  }
  ziel_nicht_erreicht = pfadNaechsterZielpunkt(weg_ans_ziel,
                        mittelpunkt, abstand, geschwindigkeit);
  aktueller_zielpunkt = weg_ans_ziel->root->wegpunkt;
  geschwindigkeit->x = mittelpunkt->x - alter_mittelpunkt.x;
  geschwindigkeit->y = mittelpunkt->y - alter_mittelpunkt.y;

  reglerBerechneMotorschritte(abstand->x, geschwindigkeit->x,
                       platte.x,abstand->y, geschwindigkeit->y,
                       platte.y, &schritte_x, &schritte_y);
  vektor = calcSteuerungsVektor(schritte_x, schritte_y);
  if (ziel_nicht_erreicht) {
    dspkommSendeMotorschritte(vektor, 0, vektorlaenge);
    platte.x = platte.x + schritte_x;
    platte.y = platte.y + schritte_y;
  }
  free(vektor);
  alter_mittelpunkt = *mittelpunkt;
}

schritte_x = -platte.x;
schritte_y = -platte.y;
vektor = calcSteuerungsVektor(schritte_x, schritte_y);
dspkommSendeMotorschritte(vektor, vektorlaenge, 0);   (line 45)
...
```

Figure 6: Source code of central wobble loop

global variables. In our experiment, we wanted to check whether the step motor is influenced by an outside agent, and if so determine witnesses for suspicious behaviour.

Figure 6 displays the central loop of the source code. Since all the function names are in German, figure 7 displays the functional structure of the system (the columns correspond to function nestings). While the ball did not reach the target, the ball position is read from the camera and converted to maze coordinates (function "dspkommEmpfangeDoublePunkt"). The function "pfadNaechsterZielpunkt" computes the euclidian distance to the next intermediate ball position, and the function "reglerBerechneMotorschritte" uses a neural net to compute the rotation of the maze. Function "calcSteuervektor" transforms this information into a control vector which is sent to the motor ("dspkommSendeMotorschritte"); the maze angles are adjusted accordingly.

Figure 8 displays the path condition for the chop between line 4 and line 45, that is, a necessary condition for influence of the motor by the camera. Performance data for this chop is given in table 2, line 4. For all atomic conditions their source file and source line is given. Path conditions are LaTexed automatically; Φ-constraints without disjunctions (i.e. simple value propagations) are automatically substituted, complex Φ-constraints are shown only on request. SSA indices of program variables (italic font) and function

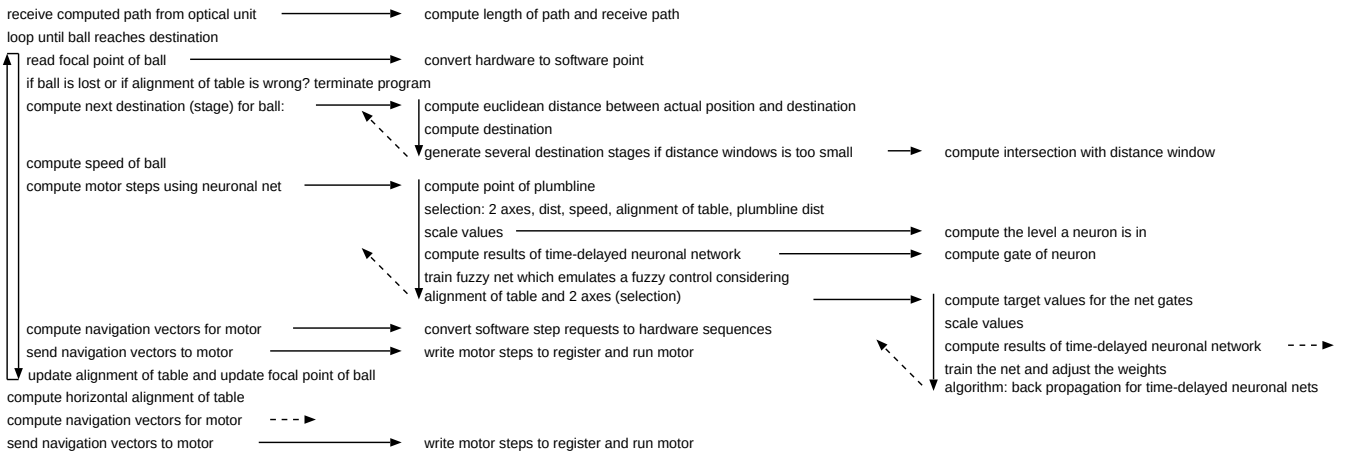

Figure 7: Functional decomposition of WobbleTable software

return values (bold font) are usually shown, because they can be used as back references to their (unique) assignment in the source code. The condition was not fed into a constraint solver, as it is already in solved form.

The condition is surprisingly small and becomes quite clear after a look at the source code (and after determining the source positions of the SSA indices). The last part of the condition requires that the target has not been reached ($\top$ stands for a non-zero, that is *true* value), the ball has a definite position, the vertical angels of the maze in x- and y direction do not exceed a value of 250 "steps", and the next intermediate target for the ball is defined. The inner disjunction demands that either the euclidian distance between ball and intermediate target does not exceed a maximum value and the ball velocity is bounded, or some condition on the neural network must be satisfied. The latter is not understandable from the path condition directly, but the source code reveals that the number of firing neurons distinguishes various cases of x/y angles, target distance and ball velocity. This part of the program needs closer examination, but so far no hints for illegal motor manipulations can be observed. Obviously understanding path conditions requires some knowledge of the program, but path conditions are less complex than one might expect.

For our next experiment we asked the WobbleTable programmers to introduce a safety violation by manipulating the motor from the keyboard. In fact the keyboard input buffer variable `taste` ("key" in German) was already declared and easy to spot. According to the programmers, `taste` was used in a debugging version, but all references to `taste` were removed later. Indeed, in the existing program there is no SDG path from the initialization $def(\texttt{taste})$ to the motor control call in line 45, thus trivially $PC(def(\texttt{taste}), 45) \triangleq false$.

After introduction of the manipulation, the path condition $PC(def(\texttt{taste}), 45)$ was computed again. The result is no longer $false$, but the condition in figure 9; computation of this path condition took 29 seconds. Among various constraints similar to figure 8, the condition contains the atomic control condition $ping_{3502}\&128 > 0$. A global Φ-constraint (displayed in the last line) states that $ping_{5294} = ping_{8855} \vee ping_{5294} = taste_{9187}$, and a formal/actual Φ-constraint adds that $ping_{5294} = ping_{3502}$. SSA indices can be used as back references to the source text: $taste_{9187}$ is indeed $def(\texttt{taste})$, $ping_{5294}$ is a formal parameter of "reglerBerechneMotorschritte". Thus we see that the step motor is manipulated by the keyboard input via variable `ping`.

Here is what the programmers did: in file `variable.h` they added declaration `extern int* ping;`, in file `dspkomm.c` they added declaration `int* ping;`. In file `regler.c` they added the statement `ping = (int*) taste;`. Deeply hidden inside `neuronal.c` they added the statement

```
if ((*ping)&0x80 > 0) {
   wert *= 1.2;
}
```

which increases the scale factor in the neural net (see figure 7, right column) by 20% if the 8th bit of `*ping` (that is, `taste`) is set. It has been used to avoid weaknesses of the system in particular test cases. Interestingly, the variable `wert` does not occur in the path condition, as it is never used in any control condition. But the SSA index $ping_{3502}$ in the witness condition links back to the source and immediately identifies the malicious if-statement. Note that this is a constructed manipulation, but not at all an obvious manipulation – a few lines of manipulative statements are distributed over various source files. A human expert would have a hard time to discover such a manipulation!

6. RELATED WORK

Our work is similar in spirit to constraint-based test data generation (e.g. [9, 10, 6]). All such methods for test data generation are based on the control flow graph and generate constraints which enforce a specific control flow. Hence they cannot generate constraints for data flow, which are essential for our analysis purposes. Furthermore, all these methods have only been applied to small programs, while our contribution centers about scaling up. Some use heuristics which do not obey the principle of conservative approximation required for safety analysis. Some are restricted to specific domains, as they use very specialized solving techniques; our approach is to provide an efficient general path condition generator which can then be connected to specialised solvers.

$$
\begin{aligned}
PC(4,45) \triangleq \Big(\big(\quad & i_{207} < |m\_x_{582}| + |m\_y_{583}| \qquad (\mathsf{calc.c} : 80, 157, 137) \\
\wedge \quad & \sqrt{\big((\mathrm{dspkommEmpfangePfad}_{9134}.root_{7876}.wegpunkt_{7878}.x_{7880} - \mathrm{calloc}_{9095}(1,8).x_{7882}) * \\
& \quad (\mathrm{dspkommEmpfangePfad}_{9134}.root_{7886}.wegpunkt_{7888}.x_{7890} - \mathrm{calloc}_{9095}(1,8).x_{7892}) + \\
& \quad (\mathrm{dspkommEmpfangePfad}_{9134}.root_{7897}.wegpunkt_{7899}.y_{7901} - \mathrm{calloc}_{9095}(1,8).y_{7903}) * \\
& \quad (\mathrm{dspkommEmpfangePfad}_{9134}.root_{7907}.wegpunkt_{7909}.y_{7911} - \mathrm{calloc}_{9095}(1,8).y_{7913}) \big)} \\
& \quad < MAX\_ZIELPUNKTABSTAND_{9332} \qquad (\mathsf{pfad.c} : 71, \mathsf{regler.c} : 71 - 74, 160, 171, 226) \\
\wedge \quad & |\mathrm{calloc}_{9109}(1,8).x_{7925}| < MAX\_ZIELPUNKTGESCHWINDIGKEIT_{9333} \qquad (\mathsf{pfad.c} : 111, \mathsf{regler.c} : 162, 226) \\
\wedge \quad & |\mathrm{calloc}_{9109}(1,8).y_{7934}| < MAX\_ZIELPUNKTGESCHWINDIGKEIT_{9333} \qquad (\mathsf{pfad.c} : 112, \mathsf{regler.c} : 162, 226) \\
\wedge \quad & \mathrm{dspkommEmpfangePfad}_{9134}.root_{7943}.naechster_{7945} = 0 \qquad (\mathsf{pfad.c} : 115, \mathsf{regler.c} : 177) \\
\big) \quad \vee \quad & \big(\\
& i_{207} < |m\_x_{582}| + |m\_y_{583}| \qquad (\mathsf{calc.c} : 80, 157, 137) \\
\wedge \quad & i_{4520} < anzahl\_neuronen_{4474} \qquad (\mathsf{neuronal.c} : 1101, 1108) \\
\wedge \quad & anzahl\_neuronen\_in\_schicht_{5403}[0] \neq 2 \qquad (\mathsf{neuronal.c} : 1348) \\
\wedge \quad & anzahl\_neuronen\_in\_schicht_{5670}[0] \neq 3 \qquad (\mathsf{neuronal.c} : 1390) \\
\wedge \quad & anzahl\_neuronen\_in\_schicht_{5979}[0] = 8 \qquad (\mathsf{neuronal.c} : 1434) \\
\big) \big) & \\
\wedge \quad & ziel\_nicht\_erreicht_{9265} = \top \qquad (\mathsf{regler.c} : 200) \\
\wedge \quad & \mathrm{dspkommEmpfangeDoublePunkt}_{9269}(\mathrm{calloc}_{9095}(1,8)) \neq 0 \qquad (\mathsf{regler.c} : 160, 203, 206) \\
\wedge \quad & |platte_{9291}.x_{9292})| \leq 250 \qquad (\mathsf{regler.c} : 213) \\
\wedge \quad & |platte_{9298}.y_{9299})| \leq 250 \qquad (\mathsf{regler.c} : 213) \\
\wedge \quad & \mathrm{pfadNaechsterZielpunkt}_{9327}(\mathrm{dspkommEmpfangePfad}_{9134}, \mathrm{calloc}_{9095}(1,8), \\
& \quad \mathrm{calloc}_{9102}(1,8), \dots) = \top \qquad (\mathsf{regler.c} : 160 \dots, 177, 226)
\end{aligned}
$$

Figure 8: Path condition for step motor

Pugh [14] uses Presburger arithmetic for solving constraints concerning array dependencies. Pugh's goal is automatic parallelization of loops, and he describes dedicated constraints and solving techniques. Our array constraints are in fact a subset of Pugh's constraints, hence not as strong; furthermore, we did not yet employ a Presburger solver. But in principle it would be possible to "plug in" his sophisticated analysis techniques into ValSoft.

Reps [15] also investigated the use of abstract data types in dependence graphs. He extends his earlier technique of context-free reachability in order to model equations for abstract types. It turns out that in connection with interprocedural dependencies, data dependence becomes undecideable. Our approach, on the other hand, is based on rewriting modulo data dependences, which is a mechanism completely orthogonal to dependence analysis. While perhaps slightly less precise, it avoids any decideability problems and is completely decoupled from the rest of the path condition generator.

7. CONCLUSION AND FUTURE WORK

Path conditions in dependence graphs are a valuable tool for various kinds of program analysis, such as program understanding or safety checks. This contribution concentrated on the practical possibilities of path conditions. Our results can be summarized as follows:

1. Path conditions are very helpful to reduce the imprecision of slicing, and can demonstrate that some slices are in fact impossible;

2. Subsequent constraint solving will generate witnesses for specific data flow, in particular for illegal influences to safety-critical computations;

3. Naive generation of path conditions does not scale;

4. Interval analysis and BDDs are the key devices for taming complexity;

5. The improved path condition generator produced a witness for a safety violation in a medium-sized C program in less than a minute.

Of course, our work is not finished at this point. One obvious task is to make path conditions work for chops with more than 10^5 edges in minutes instead hours, and we are optimistic. Future efforts will also have to compare the behaviour of various solvers, such as Redlog, Mathematica, Pugh's Omega test, and constraint logic programming. Another issue is an adaption and extension of ValSoft for Java; this requires static approximation of dynamic lookup behaviour for slicing, and generation of corresponding path conditions. Of highest priority however is the application of ValSoft to more case studies. In particular we hope to obtain commercial safety-critical C programs, and then perhaps discover a hidden trapdoor into the system – or prove that such trapdoors do not exist.

Acknowledgements. Making path conditions in ValSoft work and scale would not have been possible without earlier work in slicing algorithms, intervall analysis, BDDs, and constraint solvers; we collectively thank all the researchers involved in these topics. Jens Krinke implemented the ValSoft slicer and thus laid the basis for path conditions. Frank Tip provided valuable comments.

$$
\begin{array}{lll}
PC(def(\texttt{taste}), 45) \triangleq & i_{207} < |m\_x_{582}| + |m\_y_{583}| & (\text{calc.c} : 80, 157, 137) \\
& \text{schichtVon}_{3594}(neuron\_id_{3500}) \neq 0 & (\text{neuronal.c} : 696, 719) \\
& ping_{3502}\&128 > 0 & (\text{neuronal.c} : 696, 731) \\
& \text{schichtVon}_{4170}(i_{4538}) = 0 & (\text{neuronal.c} : 964, 966, 1113) \\
& i_{4533} < anzahl\_neuronen_{4487} & (\text{neuronal.c} : 1105, 1112) \\
& anzahl\_neuronen\_in\_schicht_{5419}[0] \neq 2 & (\text{neuronal.c} : 1352) \\
& anzahl\_neuronen\_in\_schicht_{5698}[0] \neq 3 & (\text{neuronal.c} : 1394) \\
& anzahl\_neuronen\_in\_schicht_{6023}[0] = 8 & (\text{neuronal.c} : 1438) \\
\wedge & ziel\_nicht\_erreicht_{9364} = \top & (\text{regler.c} : 202) \\
\wedge & \text{dspkommEmpfangeDoublePunkt}_{9368}(\text{calloc}_{9191}(1, 8)) \neq 0 & (\text{regler.c} : 160, 205, 208) \\
\wedge & |platte_{9390}.x_{9391})| \leq 250 & (\text{regler.c} : 215) \\
\wedge & |platte_{9397}.y_{9398})| \leq 250 & (\text{regler.c} : 215) \\
\wedge & \text{pfadNaechsterZielpunkt}_{9426}(\text{dspkommEmpfangePfad}_{9233}, \text{calloc}_{9191}(1, 8), & \\
& \quad \text{calloc}_{9198}(1, 8), \dots) = \top & (\text{regler.c} : 160 \dots, 179, 228) \\
\Phi \triangleq & ping_{5294} = ping_{8855} \vee ping_{5294} = taste_{9187} & (\text{regler.c} : 95, 158, \text{neuronal.c} : 1336)
\end{array}
$$

Figure 9: Path condition revealing a safety violation

This work is funded by Deutsche Forschungsgemeinschaft, grants DFG Sn11/5-1 and Sn11/5-2.

8. REFERENCES

[1] Franz Baader and Tobias Nipkow. *Term rewriting and All That*. Cambridge University Press, 1998.

[2] Frederic Benhamou and Alain Colmerauer. *Constraint Logic Programming: Selected Research*. MIT Press, 1993.

[3] Leeann Bent, Darren C. Atkinson, and William G. Griswold. A comparative study of two whole program slicers for C. Technical Report CS2000-0643, University of California, San Diego, Computer Science and Engineering, 2000.

[4] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic specifications*. ACM Press/Addison Wesley, 1989.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 451–490, October 1991.

[6] R.A. deMillo and A.J. Offut. Constraint-based automatic test data generation. *IEEE Transactions in Software Engineering*, pages 900–910, September 1991.

[7] Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, 1997.

[8] Tim Teitelbaum et al. Code surfer user guide and reference. Technical report, Gramma Tech Product Documentation, 2001. http://www.grammatech.com/csurf-doc/manual.html.

[9] Arnaud Gotlieb, Bernard Botella, and Michael Rueher. Automatic test data generation using constraint solving techniques. In *Proc. International Symposium on Software Testing and Analysis*, pages 53–62. ACM, 1998.

[10] Neelam Gupta, Aditya Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation model. In *Proc. International Symposium on Foundations of Software Engineering*, pages 231–244. ACM, 1998.

[11] Jens Krinke and Gregor Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, pages 661–675, November/December 1998. Special issue on Program Slicing.

[12] Jorn Lind-Nielsen. BuDDy - a binary decision diagram package. Technical report, University of Copenhagen, 2001. http://www.itu.dk/reserach/buddy.

[13] Kim Marriott and Peter Stuckey. *Programming with Constraints*. MIT Press, 1998.

[14] William Pugh and David Wonnacott. Constraint-based array dependency analysis. *ACM Transaction on Programming Languages and Systems*, pages 1248–1278, May 1998.

[15] Thomas Reps. Undecideability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems*, pages 162–186, January 2000.

[16] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proc. Foundations of Software Engineering*, pages 11–20. ACM, 1994.

[17] Tom Reps. Program analysis via graph reachability. *Information and Software Technology*, pages 701–726, November/December 1998. Special issue on program slicing.

[18] Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. In *Proc. Static Analysis Symposium*, volume 1145 of *LNCS*, pages 332–348, 1996.

[19] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, pages 649–658, November 1996.

[20] Robert Endre Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[21] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.