Efficient Path Conditions in Dependence Graphs for Software Safety Analysis

GREGOR SNELTING, TORSTEN ROBSCHINK, JENS KRINKE
Universität Passau

A new method for software safety analysis is presented which uses program slicing and constraint solving to construct and analyze "path conditions," conditions defined on a program's input variables which must hold for information flow between two points in a program. Path conditions are constructed from subgraphs of a program's dependence graph, specifically, slices and chops. The article describes how constraint solvers may be used to determine if a path condition is satisfiable and, if so, to construct a witness for a safety violation, such as an information flow from a program point at one security level to another program point at a different security level. Such a witness can prove useful in legal matters.

The article reviews previous research on path conditions in program dependence graphs; presents new extensions of path conditions for arrays, pointers, abstract data types, and multi-threaded programs; presents new decomposition formulae for path conditions; demonstrates how interval analysis and BDDs (binary decision diagrams) can be used to reduce the scalability problem for path conditions; and presents case studies illustrating the use of path conditions in safety analysis. Applying interval analysis and BDDs is shown to overcome the combinatorial explosion that can occur in constructing path conditions. Case studies and empirical data demonstrate the usefulness of path conditions for analyzing practical programs, in particular, how illegal influences on safety-critical programs can be discovered and analyzed.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*validation, reliability*; F.3.1 [**Logics and meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical Verification*; F.3.2 [**Logics and meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*

General Terms: Algorithms, Reliability, Security, Verification, Theory

Additional Key Words and Phrases: safety analysis, program slicing, path condition, information flow control

1. INTRODUCTION

In many safety-critical software applications, guarantees are needed that internal or external agents cannot illegally influence critical computations, and that confidential information cannot leak to the application environment. For example, in the case of software-controlled actuators in automobiles, aircraft, or rockets, guarantees are needed that critical computations, such as the position of an aircraft rudder, cannot be influenced by external software manipulation. As a less spectacular,

A preliminary version of parts of this article appeared in the Proceedings of the ACM/IEEE International Conference on Software Engineering [Robschink and Snelting 2002]. This work is supported by Deutsche Forschungsgemeinschaft, grant Sn11/5-2.
Authors' addresses: Fakultät für Mathematik und Informatik, Universität Passau, Innstr. 33, 94032 Passau, Germany. Third author's current address: FernUniversität in Hagen, Fachbereich Elektrotechnik, 58084 Hagen, Germany

but economically relevant example, consider professional measurement systems: for such systems, it must be guaranteed that the data path from the physical sensors to the displayed measurement values cannot be manipulated.

Information flow control [Common Criteria Project Sponsoring Organizations 2004] is the technology concerned with the above problem. Much work has been done on formal security models to control flow of information, in particular external influences on critical computations. The classical model from Bell and La Padula [1973] introduces different security levels and demands that subjects may not read objects (e.g. documents) with higher security level than their own, and may not write objects with lower security levels. Later, Goguen and Meseguer [1984] introduced a noninterference model, which partitions a system into a finite number of security domains. Noninterference means that a certain domain cannot influence the observational behavior of another domain, and the approach presents a noninterference criterion that determines whether a system is safe with respect to a given noninterference relation.

Noninterference is used in many certification processes. For example, the German Physikalisch-Technische Bundesanstalt – which is, among other tasks, responsible for the certification of commercial measurement systems – demands that the software for such systems be partitioned into two domains: (1) the so-called calibration path, which contains the critical data flow from the sensor to the value display; (2) the rest of the system which contains, for example, the user interface code. Enforcing noninterference means that statements outside the calibration path cannot influence data controlled by statements in the calibration path. The need to check measurement software for safety violations has, in fact, provided one motivation for the research described in this article.

For the development of new safety-critical systems, well-developed guidelines such as the common criteria [Common Criteria Project Sponsoring Organizations 2004] define requirements for information flow control, and allow achievement of any specific level of security. But for existing software systems – which have perhaps not been developed with security criteria in mind – *program analysis* has to be used to check whether criteria, such as Bell/La Padula or Goguen/Meseguer, are met. In particular it is necessary, (1) to determine which program parts can influence other, perhaps critical program points, and (2) to determine which of these influences are acceptable and which are safety violations.

For the first part of this task, a well-known method exists: program slicing. Slicing, first introduced by Weiser [1984], determines, for a given statement, those program parts which can influence this statement, and those program parts which definitely cannot influence the statement. Slicers such as CodeSurfer [Teitelbaum 2001] or ValSoft [Krinke and Snelting 1998] use a system dependence graph [Ottenstein and Ottenstein 1984; Ferrante et al. 1987; Horwitz et al. 1990] to determine for a given statement x all statements which may influence x. Appendix 1 explains the relationship between slicing and Goguen/Meseguer noninterference.

Slicing today is reasonably fast and can deal with production programs written in commercial programming languages. There are some language features that are hard to deal with – such as pointer arithmetic in C – but in a safety-critical context, such features could be disallowed by programming standards.

Unfortunately, even if the best known algorithms are used, slicing is quite im-

precise in practice: slices are bigger than expected and sometimes too big to be useful [Bent et al. 2000]. 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. For purposes of information flow control, we therefore proposed to combine slicing with path conditions and constraint solving [Snelting 1996; Krinke and Snelting 1998]: Let $y \rightarrow^* x$ denote any path in a dependence graph from node y to node x.

—A *path condition* $PC(y, x)$ is a necessary condition for information flow from y to x, i.e. along the path $y \rightarrow^* x$.

—After a path condition $PC(y, x)$ is generated and simplified, it is analysed using a constraint solver to determine if the path condition is satisfyable by the program's input variables.

—The values or constraints computed by the constraint solver for the input variables are necessary, but generally not sufficient, for the path condition $PC(y, x)$ to hold, i.e., for y to influence x.

—If the program executes with input values satisfying the constraints, $PC(y, x)$ is satisfied, and the influence of the statement represented by y on the statement represented by x becomes visible. In analyzing a program for safety violations, these inputs serve as witnesses for a safety violation.

—If $PC(y, x)$ is unsatisfiable, y does not influence x even though the dependence graph indicates otherwise.

Thus path conditions can make slicing more precise by reducing the number of "false positives" of potential influences on a computation. Note that our usage of the term "path condition" differs from its traditional usage in test case generation. We do not determine necessary conditions for execution flow along a specific path in the control flow graph; we determine necessary conditions for information flow between two points in the dependence graph – that is, conditions that must hold for information flow to occur between those two points.

Snelting [1996] presented fundamental formulae and theorems for the definition and simplification of path conditions. But at that time we had no implementation, no efficient algorithms, no support for the full C language, and no empirical data. Hence the contributions of this article include:

(1) a review of the role of path conditions in program dependence graphs;

(2) an extension of the notion of path conditions for arrays, pointers, abstract data types and multi-threaded programs;

(3) new techniques for decomposing path conditions;

(4) a demonstration of how interval analysis and BDDs increase the scalability of path conditions;

(5) and a set of case studies illustrating the use of path conditions in safety analysis.

Thus path conditions today can be applied to realistic programs written in commercial languages, and proved to be a useful instrument for program understanding and safety analysis.

The work described here is implemented on top of the ValSoft [Krinke and Snelting 1998; Krinke 2003a] Slicer. ValSoft is a slicer for full ANSI C; it incorporates the best known algorithms for slicing and chopping [Reps et al. 1994a; Reps and Rosay 1995; Krinke 2003a; 2002]. 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.

This article is organized as follows. Section 2 reviews dependence graphs and program slicing, and introduces basic definitions, properties, and examples of path conditions in dependence graphs. Section 3 describes extensions of the basic formulae for real languages, in particular interprocedural path conditions, treatment of data structures, and path conditions for multithreaded programs. Section 4 shows how to reduce the complexity of analysing path conditions by applying binary decision diagrams (BDDs) and interval analysis to the dependence graph. Section 5 presents a set of empirical data. Section 6 presents a case study, namely the analysis of a safety-critical real-time system. Section 7 discusses related work, and section 8 presents some conclusions.

2. BACKGROUND AND FOUNDATIONS

This section reviews program slicing and fundamental properties of path conditions as first published in Snelting [1996]. It also describes the usage of static single assignment form, weak and strong path conditions, and solving techniques for path conditions. We assume that the reader has some basic knowledge of program slicing (see, e.g., [Tip 1995]).

2.1 Intraprocedural dependences, slices, and chops

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 definition of slicing makes the notion of influence precise by requiring 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 undecidable.

Slicing computes a conservative approximation of the predicate I. Intraprocedural slices can be computed using a program dependence graph $PDG = (N, \rightarrow)$, where N is a set of nodes representing statements, expressions, and control predicates, and $\rightarrow$ denotes control dependence edges and data dependence edges [Ottenstein and Ottenstein 1984; Ferrante et al. 1987]. Control dependence and data dependence are defined next. To define control dependence, we require the control dependence graph $(C, \rightarrow_C)$ of the PDG, where $C \subseteq N$ is the set of control condition nodes, and $\rightarrow_C$ is the set of control dependence edges.

In a program dependence graph $PDG = (N, \rightarrow)$, $x \in N$ is control dependent on $y \in N$, if its execution is determined by y (e.g. in an *if* or *while* statement). Formally, given the control dependence graph $(C, \rightarrow_C)$ of the PDG, y is control dependent on x, iff a path from y to x exists in C, that is $y \rightarrow_C^* x$; x is a post-dominator for all nodes on this path except y; and x is not a post-dominator for y [Ferrante et al. 1987]. A *post-dominator* x of a node y is a node which must be executed on any control flow from y to $STOP$.

Two nodes are data dependent on each other, if a definition at one node might

be used at the other node. Formally, $x \in N$ is data dependent on $y \in N$, iff there is a path from y to x in the control flow graph, there is a variable v which is defined at y and referenced at x, and v is not killed at any node on this path.

Every PDG contains a $START$ node and all top-level statements are control dependent on $START$. Sometimes nodes are control dependent on themselves; such self-cycles can be ignored in our setting as they do not contribute to path conditions according to equation 15 (section 4.1).

Once a program dependence graph $PDG = (N, \rightarrow)$ has been computed, an intraprocedural backward slice for PDG node $x \in N$, written $BS(x)$, contains the set of all nodes in the PDG from which x can be reached: $BS(x) = \{y \mid y \rightarrow^* x\}$. The forward slice for node $x \in N$, written $FS(x)$, contains the set of all nodes reachable from x: $FS(x) = \{y \mid x \rightarrow^* y\}$. For the intraprocedural case the chop is defined as $CH(y, x) = \{z \in N \mid y \rightarrow^* z \rightarrow^* x\} = BS(x) \cap FS(y)$. The following properties of a chop are equivalent: (1) $y \in BS(x)$; (2) $x \in FS(y)$; (3) $CH(y, x) \neq \emptyset$. We consider a slice or chop to be a *subgraph* of the PDG, that is, edges between slice/chop nodes are considered to be part of the slice or chop.

Usually $BS(x)$ contains 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 $BS(x) \setminus \{y \mid I(y, x)\}$ 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 for computation of path conditions.

2.2 Control conditions

We now explain how to construct path conditions. In this section, only intraprocedural path conditions are introduced; interprocedural path conditions are introduced in section 3.

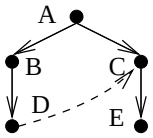
To lay the background for path conditions, we provide some additional information about control conditions. Given a control dependence graph $(C, \rightarrow_C)$ of a program dependence graph $PDG = (N, \rightarrow)$, a condition $\nu \in C$ is typically a condition associated with an *if* or *while* statement which controls the execution of other statements.

A control condition $\nu \in C$ 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 $c(\nu \rightarrow \mu)$. A control condition has the form $c(\nu \rightarrow \mu) \equiv (\nu = \mathit{true})$ or $c(\nu \rightarrow \mu) \equiv (\nu = \mathit{false})$ or $c(\nu \rightarrow \mu) \equiv (\nu = a)$ if $\nu \rightarrow \mu$ is labeled with a (where a is e.g. a specific value of a switch expression in a switch statement).

For a given node $x \in N$, the *control dependence chop* of x within the control dependence graph $(C, \rightarrow_C)$ consists of all control predicates between $START$ and x, where $CP(x)$ is defined by by

$$CP(x) = \{y \in C \mid START \rightarrow_C^* y \rightarrow_C^* x\}$$

We consider $CP(x)$ to be a sub*graph* of $(C, \rightarrow_C)$, hence edges connecting $y, z \in CP(x)$ are considered to be part of $CP(x)$.



Note that always $CP(x) \subseteq CH(START, x) \cap (C, \rightarrow_C)$, but in general $CP(x) \neq CH(START, x) \cap (C, \rightarrow_C)$. Consider the example (right): solid arcs are control dependences $\in (C, \rightarrow_C)$, and dashed arcs are data dependences, in particular $(D, C) \in (N, \rightarrow)$. Then $CH(A, E) = \{A, B, C, D, E\}$, $CH(A, E) \cap (C, \rightarrow_C) = \{A, B, C, D, E\}$, but $CP(E) = \{A, C, E\}$.

In case control flow is structured, $(C, \rightarrow_C)$ is a tree and $CP(x)$ is a single path. In practice, unstructured control flow is rare, hence even if $(C, \rightarrow_C)$ is not a tree, $CP(x)$ can very well be a single path from $START$ to x.

2.3 Path conditions

A path condition $PC(y, x)$ is a condition over program variables $\overline{v}$ which is necessary for influence:

$$I(y, x) \implies \exists \overline{v} : \ PC(y, x)$$

That is, there must exist values for the program variables such that the path condition becomes true. If the path condition cannot be satisfied ($PC(y, x) \equiv \mathit{false}$ for all possible values of $\overline{v}$), 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];
```

Obviously, `(1)` $\in BS($`(3)`$)$, but in order that `x` in `(1)` can really influence `y` in `(3)`, it is necessary that

$$PC(1, 3) \equiv (i > 10) \wedge (i + 3 = 2j - 42)$$

is satisfiable. In our approach, path conditions in general consist of control conditions: in the example, $PC(1, 3)$ contains $c(2 \rightarrow 3) \equiv (i > 10)$. They may contain additional constraints concerned with data structures such as arrays (in the example, $i + 3 = 2j - 42$ is such a constraint). Satisfiability means that the program variables in path conditions are implicitly existentially quantified. For the example in particular, any influence from (1) to (3) implies that there must exist values for the program variables which satisfy the path condition:

$$I(1, 3) \implies \exists i, j.(i > 10) \wedge (i + 3 = 2j - 42)$$

This path condition is satisfiable. But if line 2 is replaced by `if ((i>10)&&(j<5))`, the resulting path condition

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

is unsatisfiable. Thus line 1 cannot influence line 3 even though `(1)` $\in BS($`(3)`$)$. This example demonstrates how path conditions can make slicing more precise.

Since there may be many assignments to the same variable, and therefore variables may have different values at different program points, all programs are transformed into *static single assignment form* (SSA, [Cytron et al. 1991]) first. In SSA form, there is at most one assignment to every variable. If necessary, we distinguish different SSA-variants of a program variable by additional indices. As an example, consider the fragment and its SSA form

```
(1)    x = a;                 (1)    x1 = a;
(2)    while (x<7) {          (2)    while (x2 = φ(x1,x3),x2<7) {
(3)      x = y+x;             (3)      x3 = y+x2;
(4)      if (x==8)            (4)      if (x3==8)
(5)        p(x);              (5)        p(x3);
       }                             }
```

SSA distinguishes x_1 defined in (1), and x_3 defined in (3).[1] SSA uses ϕ functions [Cytron et al. 1991] to describe situations where different SSA variants of a variable may reach the same program point. In the example, $x_2 = \phi(x_1, x_3)$ just means that $x_2 = x_1 \vee x_2 = x_3$. Statement (5) is only executed if $x_2 < 7$ and $x_3 = 8$, and therefore an information flow along the data dependencies (1) $\rightarrow$ (3) $\rightarrow$ (5) is only possible under that same condition:

$$PC(1,5) \equiv (x_2 < 7) \wedge (x_3 = 8)$$

Without SSA form, the condition would be $PC(1,5) \equiv (x < 7) \wedge (x = 8)$, which is unsatisfiable and not a necessary condition for an influence (1) $\rightarrow^*$ (5).

2.4 Basic formulae for path conditions

Path conditions are defined with respect to chops in the PDG, as $I(y,x) \implies CH(y,x) \neq \emptyset$. Let $CH(y,x)$ consist of 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 by Snelting [1996]:

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

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 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:

$$E(z) = \bigvee_{P_\rho \in CP(z)} \bigwedge_{\nu \rightarrow \mu \in P_\rho} c(\nu \rightarrow \mu) \tag{2}$$

All control conditions $c(\nu \rightarrow \mu)$ 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.

Note that the PDG may contain cycles, so the outer disjunction in equation (1) may run over infinitely many paths. But section 4 will prove that non-overlapping cycles can simply be ignored, and that overlapping cycles can be handled by interval analysis. For the time being, the reader may ignore cycles.

[1] In explanatory examples, we use line numbers as SSA indices; in fact, PDG node numbers are used.

As already mentioned, SSA form must be used to generate correct path conditions. Hence some additional constraints must be generated which represent ϕ-functions. 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 $x_i = x_j \vee x_i = x_k \vee \ldots$. In the above example, the ϕ function $x_2 = \phi(x_1, x_3)$ translates into the additional ϕ constraint $x_2 = x_1 \vee x_2 = x_3$, thus

$$PC(1,3) \equiv (x_2 < 7) \wedge (x_2 = x_1 \vee x_2 = x_3)$$

The set of all ϕ constraints is denoted Φ. For a specific data dependence edge $i \to j$ from a definition of a variable to its use in a ϕ node, the corresponding constraint is written $\Phi(i \to j)$. In the above example, $\Phi(1 \to 2) \equiv x_2 = x_1$, and $\Phi(3 \to 2) \equiv x_2 = x_3$.

The Φ constraints (or at least those Φ constraints relevant for a given path) are always assumed to be part of a path condition, that is, conjunctively added to $PC(y,x)$. Hence equation (1) in fact reads

$$PC(y,x) = \bigvee_{P_\rho \in CH(y,x)} \Big(\bigwedge_{z \in P_\rho} E(z) \wedge \bigwedge_{u \to v \in P_\rho} \Phi(u \to v) \Big)$$

In section 3, we will extend path conditions to programs with procedures, arrays and other data types, pointers, and multi-threaded programs. A simple example of a condition involving array indices has already been given above.

2.5 Weak and strong path conditions

Equation (1) is perhaps not the only way to define path conditions – maybe future authors will present more sophisticated (and still decidable) definitions. In fact, for given (y,x), there might be many different path conditions even for one specific chop, which are all necessary conditions for information flow along that chop. In particular, in a path condition of the form $PC(y,x) = B_1 \wedge B_2 \wedge \ldots \wedge B_n$, any of the B_i is a necessary path condition itself.

We distinguish different path conditions for (y,x) by denoting them $PC_k(y,x)$ (where k is from some – perhaps infinite – index set $K \subseteq \mathbb{N}$). A path condition $PC_k(y,x)$ is *stronger* than another $PC_l(y,x)$ iff $PC_k(y,x) \implies PC_l(y,x)$. The set of all path conditions $\{PC_k(y,x) \mid k \in K\}$ forms a preorder by $\implies$, which can be factored by logical equivalence to obtain a true partial order. Since path conditions as defined in this paper do not contain quantifiers, equivalence is decidable by transforming path conditions into minimal disjunctive normal form.

It is our goal to construct path conditions which are as strong as possible – being not only necessary, but "almost" sufficient. Thus, if such 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 $PC(y,x) \equiv \mathit{false}$, y cannot influence x even though $y \in BS(x)$. If $PC(y,x) \equiv \mathit{true}$, probability is extremely high that $y \in BS(x)$ implies $I(y,x)$. Note that these probabilities cannot be explicitly computed, but can be compared and ordered according to the strength of the path conditions.

Several $PC_k(x,y)$ for the same path can be combined into one stronger condition by building their conjunction. Path conditions are closed under arbitrary conjunctions, because $I(y,x) \implies PC_k(y,x)$ for any $k \in K$ implies $I(y,x) \implies$

$\bigwedge_{k \in K} PC_k(y,x)$. In particular, the strongest path condition is $\overline{PC}(y,x) = \bigwedge\{PC(y,x) \mid I(y,x) \implies PC(y,x)\}$. But this strongest path condition cannot be computed, as it is an infinite conjunction, and $I(y,x) \implies PC(y,x)$ is not decidable. Path conditions as defined in equation (1) 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.

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.

2.6 Solving path conditions

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 quantifier elimination [Weispfenning 1997; 1999] such as Redlog [Dolzmann and Sturm 1997; Sturm and Weispfenning 1996] are particularly suitable. Here is an example for quantifier elimination: if

$$PC(y,x) \equiv \exists c.\ ac^2 + bc + 1 = 0$$

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. Quantifier 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. Using Redlog to solve the path condition $PC(1,3) \equiv (i > 10) \wedge (i+3 = 2j-42)$, eliminating i yields $2j > 55$, while eliminating both i and j yields just $true$. Solving $PC'(1,3) \equiv (i > 10) \wedge (j < 5) \wedge (i+3 = 2j-42)$ yields just $false$.

Note that quantifier elimination is, due to decidability problems, restricted to special 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 [Benhamou and Colmerauer 1993; Marriott and Stuckey 1998].

If path conditions can be solved, the solved conditions act as a *witness* for the path: if input values are provided which satisfy the solved $PC(y,x)$, the statements in the PDG path $y \to^* x$ are usually executed and the – perhaps illegal – influence of y on x becomes visible. This feature might be valuable in legal matters such as a litigation lawsuit against a software vendor.

Note that occasional false alarms are 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, since we can live with rare false alarms, but cannot accept potential misses of illegal influences. In our experiments so far we encountered no false alarms, and predict that

```
 1 int data[100];
 2 int temp[100];
 3
 4 void move (int* fromlist, int first, int last,
 5           int* tolist, int index) {
 6     while (first <= last)
 7         tolist[index++] = fromlist[first++];
 8 }
 9
10 void merge (int first, int mid, int last) {
11     int index, index1, index2;
12
13     index = 0;
14     index1 = first;
15     index2 = mid + 1;
16
17     while ((index1 <= mid) && (index2 <= last))
18     {   if (data[index1] < data[index2])
19             temp[index++] = data[index1++];
20         else
21             temp[index++] = data[index2++];
22     }
23
24     if (index1 > mid)
25         move (data, index2, last, temp, index);
26     else
27         move (data, index1, mid, temp, index);
28
29     move(temp, 0, last-first, data, first);
30 }
31
32 void mergesort (int left, int right) {
33     int m;
34     m = (left+right) / 2;
35     if (left < right) {
36         mergesort (left, m);
37         mergesort (m + 1, right);
38         merge (left, m, right);
39     }
30 }
41
42 int main () {
43     int i;
44
45     data[0]=999;
46     data[1]=1;
47     data[2]=23;
48     data[3]=55;
49     data[4]=44;
40
51     mergesort (0, 4);
52
53     for (i=0; i < 5; ++i) {
54         printf ("%d ",data[i]);
55     }
56     printf ("\n");
57
58   return 0;
59 }
```

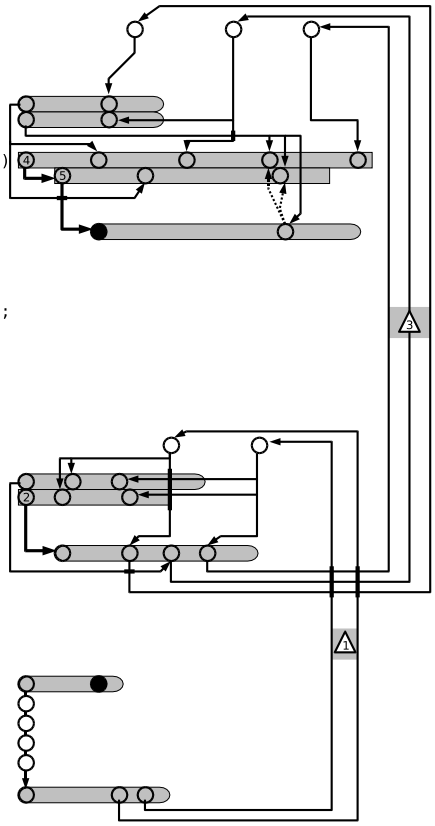


Fig. 1. A mergesort program and part of its PDG.

in practice false alarms should be rare.

2.7 An example

Figure 1 presents a mergesort program in C. Parts of the PDG are also presented, in particular essential dependences from the chop $CH(45, 21)$ between constant 999 in line 45 and array `temp` in line 21. Rounded boxes represent statement nodes in the SDG, while rectangular boxes represent control conditions; for both, their

internal structure is indicated.[2] Normal arcs represent data dependences; boldface arcs represent control dependences, and dashed arcs represent loop-carried data dependences. The example contains arrays as well as procedures. The full details for arrays and procedures are presented only in section 3. Therefore this example treats arrays as scalar variables, and recursive calls are ignored: it serves to explain the basic machinery, before we proceed to more complex language constructs.

Let us now compute $PC(45, 21)$, that is the path condition from the start node to the end node in figure 1. As recursive calls are ignored, we compute the path condition for one specific path in $CH(45, 21)$ and not for the whole chop: we leave out the dependencies from the recursive calls in line 36/37 (full treatment of interprocedural conditions is presented in section 3.2). Thus we compute the path condition for data flow along lines $49 \to 51 \to 32 \to 38 \to 10 \to 21$. Equations (1) and (2) also apply for parts of a chop such as a specific path, but does of course not deliver a general condition for information flow between two program points, but a specific, necessary condition for flow along that specific path or subchop. Compared to the full chop, conditions for a specific path or subchop are stronger; if they are satisfiable, then the path condition for the full chop is satisfiable as well.

To generate the condition, the execution conditions for statements on the path and their constituent control conditions must be generated. For example,

$$c(17 \to 18) \equiv (index1_{17} \leq mid_{10}) \wedge (index2_{17} \leq last_{10})$$

(condition 4 in figure 1), and

$$c(18 \to 21) \equiv data_{49}[index1_{17}] \geq data_{49}[index2_{17}]$$

(condition 5). Next, Φ constraints must be considered. As we will see later in more detail, Φ constraints as described above can be improved by substituting the right hand side from assignments. A precise definition of this technique will be given in section 3.6 (equation 8); right now we substitute the right hand side of the assignment to m (line 34) in line 36, yielding

$$\Phi(34 \to 36) \equiv m_{34} = (left_{32} + right_{32})/2$$

In the example, there is just one assignment to `m`, hence the Φ constraints do not contain any disjunctions: de facto they act like constant propagation. In fact, full constant propagation is automatically built into path conditions, making the conditions much stronger.

Substitution of right hand sides cannot only be applied for assignments, but also for value parameters. Exploiting such formal/actual Φ-constraints for the calls in line 51 and 38 (parameter dependences marked ⚠1 and ⚠2), we obtain $mid_{10} = m_{34} = 2, last_{10} = right_{32} = 4$. From these fragments, we obtain

$$E(21) \equiv (index1_{17} \leq 2) \wedge (index2_{17} \leq 4) \wedge (data_{49}[index1_{17}] \geq data_{49}[index2_{17}])$$

Similarly, $E(38)$ is determined to be $left_{32} < right_{32}$ (condition 2), which via Φ constraints for the mergesort call in line 51 simplifies to $E(38) \equiv 0 < 4 \equiv true$.

[2]ValSoft uses a fine-grained PDG, where subexpressions have their own PDG nodes; this has advantages for some applications [Krinke 2003a].

The initial path condition thus is

$$\begin{aligned} PC(45,21) &\equiv E(21) \wedge E(38) \\ &\equiv (index2_{17} \leq 4) \wedge (index1_{17} \leq 2) \wedge (data_{49}[index1_{17}] \geq data_{49}[index2_{17}]) \end{aligned}$$

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

Automatic simplification generates *true*, which is quite obvious: of course we can always find values for `index1`, `index2`, and `data[]` such that $PC(45,21)$ becomes true. Were `index1`, `index2`, and `data[]` input variables, we could use any values for these input variables as a witness which makes the data flow $45 \rightarrow 21$ visible (in fact, the program as given in figure 1 does not have input variables).

Hence there is high probability that there is information flow from line 45 to line 21, even without a recursive `mergesort` call: the value 999 is eventually assigned to the `temp` array. But note that due to the coarse-grained array treatment, the path condition is too weak. That means, it is too pessimistic — the path condition is overly suggestive of the existence of an influence. On the other hand, the still missing recursive calls would make the path condition even weaker. This observation is the motivation for extensions which handle specific language constructs and will be described in the following section.

3. ANALYZING REAL LANGUAGES

The basic formulae for path conditions, as described in the last section, cover intraprocedural analysis of sequential programs without data structures only. In this section, we introduce extensions for interprocedural analysis, arrays, pointers, abstract data types, assertions, and multi-threaded-programs. These extensions make path conditions much stronger. As the section is rather technical, application-oriented readers may wish to skip to section 4 and come back to this section later.

3.1 Interprocedural slicing and chopping

First, we summarize the fundamentals of interprocedural slicing. Following Horwitz et al. [1990], the PDG is extended: for every procedure f a procedure dependence graph is constructed, which is basically a PDG with start node $START_f \in C$ and with formal-in and -out nodes for every formal parameter of f. A procedure call is represented by a call node and actual-in and -out nodes for each actual parameter. The call node is connected to the start node by a call edge, the actual-in nodes are connected to their matching formal-in nodes via parameter-in edges and the actual-out nodes are connected to their matching formal-out nodes via parameter-out edges. Such a graph is called Interprocedural Program Dependence Graph (IPDG). The System Dependence Graph (SDG) is an IPDG, where summary edges between actual-in and actual-out have been added representing transitive dependence due to calls. Furthermore, parameter passing is assumed to be value-result, which is modeled by additional assignments [Horwitz et al. 1990].

For interprocedural slicing, it is not enough to perform a reachability analysis on IPDGs or SDGs. The resulting slices are not accurate as the calling context is not preserved: the algorithm may traverse a parameter-in edge coming from a call site into a procedure, and leave the procedure through a parameter-out edge going to a

different call site. Such a path is an *unrealizable path* because it is impossible in a program for a called procedure not to return to its call site. We consider an interprocedural slice to be precise if all nodes in the slice are reachable by a *realizable* path. Reps et al. presented algorithms for precise interprocedural slicing and chopping, which keep track of the calling context when following dependencies through a procedure body [Reps et al. 1994b; Reps and Rosay 1995]. These algorithms can be understood as an instance of context-free language reachability [Reps 1998]. Reps' technique constrains the possible paths in a slice or chop to capture correct context for procedure calls. A comparison of various context-sensitive slicing and chopping algorithms has been done by Krinke [2002].

3.2 Interprocedural path conditions

Similar to interprocedural slices and chops, interprocedural path conditions are restricted to realizable paths. To compute an interprocedural path condition $PC(x, y)$, we first consider the case that start and end point x and y are in the same procedure. We compute the interprocedural chop $CH(x, y)$ ignoring call, parameter-in and -out edges (but obeying summary edges). For the realizable paths in $CH(x, y)$, the conditions itself are generated using the standard formula (1). If $CH(x, y)$ contains a summary edge $a \to b$ (which represents transitive dependencies through the procedure body), $PC(a, b)$ as computed for the procedure body is used: a must be an actual-in node for a variable v and b must be an actual-out node for a variable w. Let a' and b' be the corresponding formal-in and -out nodes and v' and w' the corresponding formal parameters. Then

$$PC(a, b) = E(a) \wedge (v = v') \wedge PC(a', b') \wedge E(b) \wedge (w = w') \tag{3}$$

The constraints that bind actual and formal parameters together can be seen as additional Φ constraints for the parameter edges $a \to a'$ and $b \to b'$: $\Phi(a \to a') \equiv v = v'$, $\Phi(b' \to b) \equiv w = w'$. In fact, the summary condition is used like a Φ constraint as well: $\Phi(a \to b) = PC(a, b)$.

The path conditions between formal parameters must be computed only once and can be stored and reused for other calling contexts. When $PC(a', b')$ is needed in another context where parameter edges $a'' \to a', b' \to b''$ bind the actual parameters v'' and w'' to v' and w', the previously computed $PC(a', a')$ can be reused with different Φ constraints:

$$PC(a'', b'') = E(a'') \wedge (v'' = v') \wedge PC(a', b') \wedge E(b'') \wedge (w'' = w')$$

In the presence of recursion the generated path conditions can be recursive themselves, as equation (3) amounts to an unfolding of path conditions for procedure bodies. This problem can be circumvented by lowering the precision and setting $PC(x, y) = \mathit{true}$ for summary edges which are due to nested (recursive) calls. More generally, $PC(x, y) = \mathit{true}$ can be used once the unfolding of procedure calls exceeds a certain depth. In effect, procedure unfolding is cut off at a certain depth of calling context.

As described above, an interprocedural path condition $PC(x, y)$ requires the start and end point x and y to be in the same procedure, and such path conditions are called *same-level*. The machinery to compute non-same-level path conditions is omitted and can be found in [Krinke 2003a].

3.3 Precise loop handling

Even in SSA form, a variable has several values during runtime. This must sometimes be respected when generating path conditions. As an example, consider the fragment

```
(1)   a = u();
(2)   while (n>0) {
(3)     x = v();
(4)     if (x>0)
(5)       b = a;
(6)     else
(7)       c = b;
      }
(8)   z = c;
```

In order to compute $PC(1,8)$, we observe that there is exactly one cycle-free path in the SDG from (1) to (8), namely $1 \to 5 \to 7 \to 8$. All statements on this path must at least be executable, thus $E(5) \equiv (n > 0) \wedge (x > 0)$ as well as $E(7) \equiv (n > 0) \wedge \neg(x > 0)$ must be satisfiable. Applying equation (1), we obtain

$$PC(1,8) \equiv E(1) \wedge E(5) \wedge E(7) \wedge E(8) \equiv (n > 0) \wedge (x > 0) \wedge (n > 0) \wedge \neg(x > 0) \equiv \mathit{false}$$

which is clearly incorrect even though the example is already in SSA form.

The reason is that $5 \to 7$ is a loop carried dependency: the value for `b` is used for `c` only one loop iteration later – when `x` may already have a new value. Thus two values for `x` must be distinguished: one for the path fragment before the loop-carried dependency, and one for the path fragment after it.

Earlier versions of the path condition generator simply replaced control conditions containing the same variable connected by a loop-carried dependency by *true*. The resulting path conditions are still necessary conditions, but weaker than those respecting loop-carried variable distinctions.

For increased precision, we now use additional SSA indices (and ϕ functions) to distinguish between variable instances connected by a loop-carried dependency. For the above example, we thus obtain

$$E(5) = (n > 0) \wedge (x_1 > 0), \qquad E(7) = (n > 0) \wedge \neg(x_2 > 0)$$

In general, such additional indices have to be provided for all path segments of a chop which are connected by a loop-carried dependency. Henceforth we assume that the SSA indices respect such loop-carried distinctions of variable instances. Thus path conditions can now express that a variable may have different values during loop iterations. The details of this technique can be found in [Krinke 2003a].

3.4 Arrays

If array elements are distinguished, additional constraints for index expressions are 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 (1) thus becomes

$$PC(x,y) = \bigvee_{P_\rho \in CH(x,y)} \Big(\bigwedge_{z \in P_\rho} E(z) \Big) \wedge \Big(\bigwedge_{z \to z' \in P_\rho} \delta(z \to z') \Big) \qquad (4)$$

where

$$\begin{aligned} \delta(z \to z') &\equiv true && \text{if } z \to z' \text{ is not an array dependence edge} \\ \delta(a[e_1] \to a[e_2]) &\equiv e_1 = e_2 && \text{otherwise} \end{aligned}$$

The δ conditions can be seen as generalization of Φ conditions, and are in fact treated the same way. 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 (e.g. [Pugh and Wonnacott 1998], this is just one example of the many decision procedures that can analyse Presburger arithmetic).

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 to take care of possible aliases. As an example consider

```
(1) a[i] = x;
(2) a[j] = y;
(3) z = a[k];
```

The standard approach [Agrawal et al. 1991] is to assume that assignments to array elements are *non-killing definitions* (i.e. no previous assignment is killed). This generates data dependence edges $1 \to 3$ and $2 \to 3$, and path condition $PC(1,3) \equiv \delta(1 \to 3) \equiv (i = k)$. However, this is problematic for path conditions, as knowledge about the execution order of (1) and (2) is not available from the control and data dependences alone, and thus there is no knowledge that (2) may kill the definition at (1). In fact, if $j = i$ then definition (1) is killed in (2) and $PC(1,3)$ should evaluate to *false* instead of $i = k$ – the latter condition is too weak.

To make path conditions as strong as possible, we assume assignments to array elements to be *killing modifications*: an assignment to an array element uses the (complete) array before the specified element is defined. This approach does not change $BS(x)$ for any x except where x is an assignment to an array element. In the example we have data dependence edges $1 \to 2$ and $2 \to 3$, where edge $1 \to 2$ is going from a definition to a definition and $2 \to 3$ is going from a definition to a use. The direct dependence $1 \to 3$ has disappeared, but slicing is still correct due to the newly created chain of dependencies. Note that an array definition has no incoming data dependencies from other array definitions if it is a non-killing definition.

In order to construct a necessary condition for information flow along edges from array definitions to array definitions, the δ constraints for edges going from a definition to a definition are *negations* of the δ conditions from equation (4). The reason is that the array element defined at the edge start is only used at the edge end point if the array indices are *not* the same.

Thus equation (4) is only valid for the standard approach. For our more precise variant, the δ constraints are more complex and need some additional notation: we write $i \xrightarrow{dd} j$ for an SDG edge connecting two array definitions, we write $i \xrightarrow{du} j$ for an

```
 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 }
```

Fig. 2. Some array uses.

SDG edge connecting an array definition and an array use, and we write $A(i) = e$ for the array index of an expression $a[e]$ at node i. δ constraints are only valid for a *specific* path under examination, because we must follow backwards all *dd*-edges between definitions of array elements:

$$PC(x,y) = \bigvee_{P_\rho \in CH(x,y)} \Big(\bigwedge_{z \in P_\rho} E(z) \Big) \wedge \Big(\bigwedge_{z \to z' \in P_\rho} \delta(z \to z', P_\rho) \Big) \tag{5}$$

$$\delta(z \to z', P_\rho) \equiv \begin{array}{l} \text{let } z_1 \overset{dd}{\to} z_2 \ldots \overset{dd}{\to} z_k \overset{du}{\to} z' \text{ be a maximal subpath in } P_\rho,\ z_k = z \\ \text{in } (A(z_1) = A(z')) \wedge \big(\bigwedge_{i=2..k} A(z_i) \neq A(z')\big) \end{array} \tag{6}$$

Note that the maximal subpath in this equation is uniquely determined. If there are no *dd*-edges, equations (5) and (6) reduce to equation (4).

For the above example, equations (5) and (6) generate

$$PC(1,3) \equiv \delta(2 \to 3, 1 \to 2 \to 3) \equiv (i = k) \wedge (j \neq k)$$

which makes clear that information flows from (1) to (3) only if (2) does not kill the definition at (1), and is stronger than the condition $i = k$ which was obtained above using the standard approach.

Let us now consider the example in figure 2. For the chop between lines 8 and 21 in figure 2, ValSoft generates a big path condition which heavily relies on equations (4), (5), (6), as well as Φ constraints and substitutions of right hand sides. It is simplified by constraint solving to

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

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 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 $i = 53$.

3.5 Pointers

Until now – with the exception of array dependencies – a data dependence $i \to j$ always leads from the definition i of a variable x and its usage j. In the presence of pointers, i and j may contain complex pointer expressions. Consider the following fragment and its corresponding SSA form:

```
(1) *p = x;                    (1) *p0 = x;
(2) *q = y;                    (2) *q = y;
(3) if (a > 7)                 (3) if (a > 7)
(4)   p = q;                   (4)   p1 = q;
(5) z = *p;                    (5) p2 = φ(p0,p1), z = *p2;
```

Besides the data dependence $4 \to 5$, there exist also the data dependences $1 \to 5$ and $2 \to 5$ due to pointer dereferencing. To handle such pointer expressions, we introduce a straightforward extension of the Φ constraints: we also allow *non-scalar* Φ constraints such as $\Phi(1 \to 5) \equiv (*p_2) = (*p_0)$ and $\Phi(2 \to 5) \equiv (*p_2) = (*q)$.

To compute data dependences in the presence of pointers (such as $1 \to 5$ or $2 \to 5$), *Points-to* or *Alias* analysis has to be used. Points-to analysis computes for every pointer a set of heap objects (in fact, object creation sites) it may point to at runtime. Many points-to algorithms have been published; in our implementation we use a technique similar to Burke et al. [1995]. In the example, there are two object creation sites P and Q for $*p$ and $*q$ respectively; points-to analysis determines that $pointsto(p) = \{P, Q\}$, $pointsto(q) = \{Q\}$.

We can make the path conditions even stronger by introducing additional δ constraints that represent the additional requirements of aliasing. For example, $\Phi(2 \to 5) \equiv (*p_2) = (*q)$ is only valid if $p_2 = q$. Thus equation (4) also holds for data dependence due to pointer usage with a different δ constraint (let $P(x)$ be the expression at x):

$$\begin{array}{lll} \delta(i \to j) \equiv p = q & \text{if } P(i) = *p, P(j) = *q \\ \delta(i \to j) \equiv \&x = q & \text{if } P(i) = x, P(j) = *q \\ \delta(i \to j) \equiv p = \&x & \text{if } P(i) = *p, P(j) = x \end{array} \tag{7}$$

These equations handle only a subset of possible pointer expressions. As all others are handled by $\delta(i \to j) \equiv \mathit{true}$, the path conditions are still correct (that is, necessary conditions, but not as precise as possible).

3.6 Abstract data types

Most programs rely not only on arrays, but on all kinds of standard datatypes such as lists, stacks, queues etc. Precision of path conditions can be increased even more by taking into account the algebraic semantics of such data types. To apply this technique, we assume that for some datatype in the program, an equational specification is given. To exploit such specifications operationally, we make a standard assumption [Bergstra et al. 1989]: 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.[3] Note that we do not require programmers to provide such equational

[3]If this is not the case, there are normalizing techniques like Knuth-Bendix completion (see the excellent description by Baader and Nipkow [1998]), but that is outside the scope of this paper.

specifications, but assume that they are available for standard data types such as lists, stacks, trees etc.

Equational specifications can be exploited whenever a path condition contains a term t that is built according to the signature of the specification. We write $t \rightsquigarrow u$ if t can be reduced to the (unique) normal form u. A term t containing a variable x is written $t[x]$.

As an example, consider a program using a stack. To see how the standard stack equations (e.g. $top(push(s,x)) = x$, $pop(push(s,x)) = s$) are exploited, consider the code fragment

```
(10) if (b)                          (13)   stack = push (20, stack)
(11)   stack = push (10, stack)      (14) ...
(12) else                            (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 in equation (4):

$$\delta(11 \rightarrow 15) \equiv x = 10, \qquad \delta(13 \rightarrow 15) \equiv x = 20$$

Thus δ conditions are now generalized to arbitrary abstract data types and not just dependencies between array elements. The example illustrates the technique of backsubstituting all possible sources of data dependencies for a variable occurring in a term to be rewritten; we call it *rewriting modulo data dependencies.*

In general, rewriting modulo data dependencies must intertwine backsubstituting and rewriting steps in a way which is similar to rewriting modulo an equational theory [Baader and Nipkow 1998]. It is defined by the following rule:

$$\frac{t[u] \rightsquigarrow v, \quad (u_i = r_j) \in \Phi(i \rightarrow j)}{t[r] \rightsquigarrow v}$$

In this rule, u_i, r_j are SSA-variants of variables u and r occurring in term t. Thus the rule states that during rewriting, Φ constraints may be applied. The resulting normal forms are used as additional δ-constraints:

$$\frac{u_i{=}e; \in \mathit{Assignments}, \ e \rightsquigarrow r, \quad (u_i = v_j) \in \Phi(i \rightarrow j)}{\delta(i \rightarrow j) \equiv v_j = r} \tag{8}$$

which states that not only are assignment right hand sides substituted in Φ constraints as already mentioned in section 2.7, but the right hand side is replaced by the result of rewriting modulo data dependencies. Thus equation 8 describes the substitution of right hand sides in general; the simple case from section 2.7 is subsumed by the empty rewrite system. Treatment of path conditions then proceeds as in equation (4). In case a Φ constraint contains disjunctions, these "multiply through" in the reduction (that is, the above inference rules become nondeterministic) and $\delta(u \rightarrow v)$ also contains disjunctions.

Note that substituting right sides must only be done if no cyclic substitutions are introduced e.g. by cyclic assignments or by recursive function calls. The path

```
1   #define TRUE 1
2   #define CTRL2 0
3   #define PB 0
4   #define PA 1
5   void printf();
6   void main() {
7      int p_ab[2] = {0, 1};
8      int p_cd[1] = {0};
9      char e_puf[8];
10     int u;
11     int idx;
12     float u_kg;
13     float kal_kg = 1.0;

14     // [XASSERT] kal_kg==1.0; idx<10;
15     while(TRUE) {
16       if ((p_ab[CTRL2] & 0x10)==0) {
17         u = ((p_ab[PB] & 0x0f) << 8)
18             + (unsigned int)p_ab[PA];
19         u_kg = (float) u * kal_kg;
       }
20     if ((p_cd[CTRL2] & 0x01) != 0) {
21       for (idx=0;idx<7;idx++) {
22         // [XASSERT] idx >= 0; idx <7;
23         // e_puf[idx] >= 65; e_puf[idx]<=90;
24        e_puf[idx] = (char)p_cd[PA];
25        if ((p_cd[CTRL2] & 0x10) != 0) {
26          if (e_puf[idx] == '+')
27          kal_kg *= 1.01;
28          else if (e_puf[idx] == '-')
29            kal_kg *= 0.99;
          }
         }
         // [END XASSERT]
30       e_puf[idx] = '\0';
       }
32     printf("Artikel: %7.7s\n   %6.2f kg    ",
33         e_puf,u_kg);
     }
     // [END XASSERT]
   }
```

Fig. 3. Simple measurement program including an XASSERT assertion.

condition generator checks this condition and substitute right sides whenever possible, as it usually makes path conditions much stronger. Full details can be found in [Robschink 2005].

3.7 Assertions

In order to improve path conditions by exploiting background knowledge, assertions can be used. Ordinary assert statements in C generate a corresponding control condition, which already makes path conditions stronger. Furthermore, ValSoft provides "XASSERT" which allow more fine-grained control: besides specifying a boolean formula, it also allows specification of the scope of an assertion. Usually the scope is a list of statements within a procedure body; scopes may be nested.

Assertions can be invariants which in principle could be derived from the source code, but more typically are truly additional constraints that cannot be derived from the program. The use of assertions makes path conditions much stronger and can reduce the size of path conditions dramatically. In practice, assertions are often used to focus on a specific region in a chop by providing an assertion which excludes data flow along other paths. Assertions are assumed to be valid for all SSA variants of a variable occurring in its scope, thus acting as a scope invariant.

Figure 3 shows a simple measurement program which was discussed in [Snelting 1996]; this program allows manipulation of the displayed weight value by keyboard input '+' or '-'. The source text contains two nested XASSERTs, both containing variable `idx`. The outer XASSERT affects only idx_{21} (this SSA variant comes from the `for` loop initialization `idx=0;`), whereas the inner assertion acts upon both idx_{21} and idx_{31} (the latter coming from the loop variable incrementation `idx++;`, which in the SDG is part of the loop body). Thus assertions are not just extracted from the source code, but the appropriate SSA indices are added (if more than one SSA variant is affected by an assertion, the assertion is duplicated for every SSA variant). Finally, the assertion is conjunctively combined with the path condition.

In figure 3, the original path condition for the chop between keyboard input (variable `p_cd` in line 8) and displayed value (`printf` statement in line 33) is (without

SSA indices)

$$\begin{aligned} PC(8,33) = \quad & \big((p\_ab[0] \;\&\; 16 = 0) \wedge (p\_cd[0] \;\&\; 1 \neq 0) \wedge (idx < 7) \\ & \wedge (p\_cd[0] \;\&\; 16 \neq 0) \wedge (e\_puf[idx] = '+')\big) \\ \vee \quad & \big((p\_ab[0] \;\&\; 16 = 0) \wedge (p\_cd[0] \;\&\; 1 \neq 0) \wedge (idx < 7) \\ & \wedge (p\_cd[0] \;\&\; 16 \neq 0) \wedge (e\_puf[idx] \neq '+') \wedge (e\_puf[idx] = '-')\big) \end{aligned}$$

indicating the safety violation mentioned above. Now let us assume the engineer knows that the hardware used for keyboard input can only deliver capital letters, but not special characters. This is expressed by the condition $65 \leq e\_puf[idx] \leq 90$ in the second assertion, which is conjunctively added to $PC(8,33)$. The result is *false* – if only primitive hardware is used, the safety violation is not possible. This example shows that assertions can indeed reduce path conditions dramatically, but also demonstrates that erroneous assertions can generate false safety statements.

3.8 Multi-Threaded Programs

Path conditions as described so far can handle only sequential programs. In this section, we describe how path conditions can be generalized to multi-threaded programs. The method we describe here is based on Krinke's slicing algorithm for multi-threaded programs [Krinke 1998; 2003a; 2003b].

The main problem when analyzing multi-threaded programs is the presence of *interference*. Interference is data flow between variables that are shared between parallel executing statements. A node j is called *interference dependent* on node i, if there is a variable v, such that $v \in def(i)$ (v is defined at i) and $v \in ref(j)$ (v is referenced at j) and i and j may potentially be executed in parallel.

A *threaded program dependence graph* (tPDG) is an extended PDG in which interference dependence edges have been added. The technique to calculate interference dependence edges is beyond the scope of the paper; such edges can be calculated using standard algorithms [Knoop et al. 1996]. A straightforward approach assumes the existence of a boolean function $parallel(i,j)$ which returns *true* if it is possible for nodes i and j to execute in parallel (see e.g. [McDowell and Helmbold 1989] for an overview or [Naumovich and Avrunin 1998] for a more recent algorithm). An interference dependence edge $i \xrightarrow{\text{id}} j$ is inserted if there is a variable v that is defined at i, referenced at j, and $parallel(i,j)$ is true.

Interference dependencies are not transitive, which is in striking contrast to normal data dependencies. The transitivity of data and control dependencies is obvious from their definition, hence the composition of PDG paths always results in a path again. But if a statement x is interference dependent on a statement y that is interference dependent on z, x is dependent on z iff there is a possible execution in which these three statement are executed in sequence. If we assume that interference dependencies are transitive, we would in fact introduce the possibility of time travel [Krinke 1998].

To compute path conditions in the tPDG, one might replace all interference dependence edges by normal data dependence edges and compute the path conditions as usual. The resulting path conditions are always correct, because the replacement of interference dependency by data dependency is a conservative approximation, thus the resulting path condition is a necessary condition for information flow.

However, the resulting path conditions are too weak as they allow transitive paths and time travel. For example, consider the following fragment:

```
thread 1:                thread 2:
  (1)    a = b;            (5) if (x>0)
  (2)    c = d;            (6)   d = e;
  (3)    e = a;            (7) if (y>0)
                           (8)   d = a;
```

It is impossible that (2) is executed *after* (3); however, due to the interference dependences $3 \to 6$ and $6 \to 2$, there exists a path from (3) to (2) and – interpreting $(3) \to (6)$ and $(6) \to (2)$ as ordinary data dependency edges – the path condition computes to $PC(3,2) \equiv x > 0$. This weak, but satisfiable path condition indicates that (3) may travel backward in time and be executed before (2).

One possibility to eliminate time travel is to compute more precise chops. Unfortunately, for precise threaded chops it is not enough to compute more precise slices (perhaps with Krinke's technique [Krinke 1998; 2003b]) and use the intersection of a forward and a backward slice (as for sequential intraprocedural programs). In the example the intersection of $FS(1)$ and $BS(2)$ includes (6), but a precise chop would not contain (6), because there is no execution where (6) is influenced by (1) and executed before (2) – (6) is only influenced by (1) if it is executed after (2). Still, the intersection of threaded slices can be used as a basis for path conditions, because in equation 1 time traveling can be excluded by using the notion of a *threaded witness*. A threaded witness is a witness of a possible program execution: it presents a statement execution sequence which is free of time travel and consistent with the execution order in every thread. Formally, a sequence $l = \langle n_1, \ldots, n_k \rangle$ of nodes is a *threaded witness*, iff

$$\forall i \in 1 \ldots k : \forall j \in 1 \ldots i-1 : \neg \mathit{parallel}(i,j) \Rightarrow m_j \longrightarrow^\star m_i \text{ in the CFG.}$$

Hence all nodes in a thread must be reachable from its predecessors if they cannot execute in parallel. The definition assumes a simple model of parallel execution, similar to structured `cobegin/coend` parallelism.[4] Under the assumption that for every path it can be decided whether it is a threaded witness, the general equation 1 becomes:

$$PC(y,x) = \bigvee_{\substack{P_\rho \in CH(y,x), \\ P_\rho \text{ is a threaded witness}}} \bigwedge_{z \in P_\rho} E(z) \tag{9}$$

In fact the definition of a threaded witness implies its decidability. The decompositions from section 4.1 can also be applied to the multi-threaded version. The idea is to decompose a path into interference-edge-free subpaths. For the subpaths the path condition can be generated without checking the threaded witness property. The property only has to be checked at the connecting interference edges.

[4]This definition is different than the one presented by Krinke [1998]: it is more precise and more generally applicable.

4. SCALING UP

Path conditions as introduced in the last two sections do not scale. In practice, SDGs have thousands to tens of 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 sizes.

To overcome these obstacles, we apply several techniques:

(1) New formulae for the recursive decomposition of path conditions are introduced;
(2) Ordered binary decision diagrams (OBDDs) are used to avoid blowup of path conditions;
(3) Interval analysis is performed on the SDG, identifying a hierarchy of reducible loops, irreducible loops, or acyclic subgraphs.
(4) The path condition is generated in a divide-and-conquer-style, exploiting the interval analysis and decomposition results.

4.1 Decomposition of path conditions

Snelting [1996] has shown how to simplify path conditions if only structured control flow is used. Typically, the four levels of nested disjunctions or 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. Note that we leave out global Φ and δ constraints, but that the equations below are still valid if Φ constraints are included. The proofs (see Appendix 2) are essentially the same. The same remark applies to δ constraints for arrays etc. (equations (4) - (9)).

The decomposition formulae rely on the notion of a dominator. A node x is a dominator of node y in a CFG or SDG, if every path from $START$ to y must pass through x [Tarjan 1974]. Within a chop $CH(x,y)$, u dominates v if every path from x to v must pass through u.

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

$$PC(x,y) = PC(x,z) \wedge PC(z,y) \tag{10}$$

If there is not just one z dominating y, but k nodes $z_1,\ldots,z_k$ where any path $x \to^* y$ must contain a z_i, then

$$PC(x,y) = \bigvee_{i=1}^{k} PC(x,z_i) \wedge PC(z_i,y) \tag{11}$$

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 $o_1,\ldots,o_m \in S$. Entry and exit points need not necessarily be disjoint. Then

$$PC(x,y) = \bigvee_{i=1}^{k} \Big(PC(x,e_i) \wedge \big(\bigvee_{j=1}^{m} PC(e_i,o_j) \wedge PC(o_j,y) \big) \Big) \tag{12}$$

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

$$PC(x,y) = E(y) \wedge \bigvee_{z \in pred(y)} PC(x,z) \tag{13}$$

The symmetric formula is valid as well:

$$PC(x,y) = E(x) \wedge \bigvee_{z \in succ(x)} PC(z,y) \tag{14}$$

An important theorem – first proved by Snelting [1996] – states that cycles can be ignored. This makes the set of paths for any chop finite. Let $x \to x_1 \to \ldots \to x_k \to x$ be a cycle. Then

$$PC(x,y) = \bigvee_{\substack{P_\rho \in CH(x,y) \\ x_k \to x \ \notin P_\rho}} \bigwedge_{z \in P_\rho} E(z) \tag{15}$$

This equation is the same as fundamental equation (1), except that the cycle's back edge $x_k \to x$ is excluded and therefore the path through the cycle is left out. Hence the equation states that for the computation of $PC(x,y)$, the cycle at x can be ignored. This theorem is due to the fact that a path through a cycle only makes a path condition stronger, but the stronger subconditions are canceled out in the outer disjunction of equation (1) due to the absorption law ($A \vee (A \wedge B) = A$). Note that equation (15) does only apply to non-overlapping cycles; if cycles overlap, just ignoring them would miss some paths and hence generate a path condition which is too strong and not necessary anymore. Section 4.3 explains how to handle overlapping or nested cycles.

Let us add some remarks on execution conditions. As equation (2) is structurally identical to equation (1), decompositions analogous to equations (10) - (15) can be derived for execution conditions as well. We omit the corresponding equations. Note however that the equations for path conditions are defined with respect to a chop $CH(x,y) \subseteq (N, \to)$, while the corresponding equations for execution condition $E(z)$ are defined with respect to $CP(z) \subseteq (C, \to_C)$. If intraprocedural control flow is structured, $CP(z)$ is a tree and the outer disjunction in equation (2) disappears; in fact, $E(z)$ can then be computed by

$$E(START) \equiv true, \qquad E(z) = E(father(z)) \wedge c(father(z) \to z) \tag{16}$$

4.2 Binary Decision Diagrams

Path conditions typically contain the same execution conditions and control conditions repeated over and over, mounted up to substantial heaps of conjunctions and disjunctions. Binary decision diagrams (BDDs [Bryant 1986]) are the data structure of choice in our situation since they support the compact representation of formulae with many such shared expressions. In particular, exponential blowup of path conditions is avoided, and boolean operations can efficiently be implemented. BDDs have become a standard instrument in model checking, where they allow the compact representation of an automaton's symbolic state space. Several off-the-shelf BDD packages are available today, which in principle all can be used to improve path condition computation.

We chose one particular BDD package BuDDy [Lind-Nielsen 2001], and exploit it as follows. First, control conditions $c(i \to j)$ 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

$E(u)$ are cached at statement u, as $E(u)$ can appear in many path conditions. All conditions are handled in BDD form, and path condition generation as described in equations (1) – (16) is implemented through BDD operations.

The use of BDDs 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: the empirical data in section 5 demonstrates that BDDs are one key factor in making path conditions scale for large programs.

Note that in the current implementation the final BDDs are translated back to minimal disjunctive normal form in order to generate readable output, and generate textual input for subsequent constraint solvers. This can result in exponential blowup of the formula. While exponential blowup is rare, it cannot be avoided completely if output in minimal disjunctive normal form is required. But note that there are textual representations for BDDs, which employ additional variables for shared subterms and avoid any risk of exponential blowup. Such textual representations can be used to generate output as well as input for subsequent constraint solvers.

4.3 Interval analysis

In practice, a chop contains many backward edges, and typically only half of them belong to reducible loops. This is in striking contrast to control flow graphs, where irreducible loops are very rare in practice. We therefore perform interval analysis to obtain a hierarchy of nested cycles.

Interval analysis has been introduced by Tarjan [1974] as a technique to identify nested loops in reducible control flow graphs. It has later been extended by Sreedhar et al. [1996] to cope with irreducible loops. 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.

The SGL 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 algorithm by Lengauer and Tarjan [1979] is used for efficient computation of the dominator tree. Note that the SGL algorithm can be implemented in quasi-linear time [Ramalingam 1999]. Figure 4 presents an example of a SGL decomposition of a typical SDG. Every "egg" is either a strongly connected component (hatched) or a reducible loop (unhatched); in this example, the outermost loop happens to be reducible.

According to equation (15) cycles can be ignored if they do not overlap. In particular, reducible loops in which all back edges go back to the same entry point can be ignored, as equation (15) guarantees that removing the back edges does not change the path condition. However, paths in overlapping cycles cannot be ignored. As an example, consider figure 4 (right part): there are five cycle-free paths from X to Y, namely $XABY$, $XABCDEY$, $XCDEY$, $XCDABY$, $XCDEBY$; the last one is lost if the back edge EB is removed. This demonstrates that in the presence

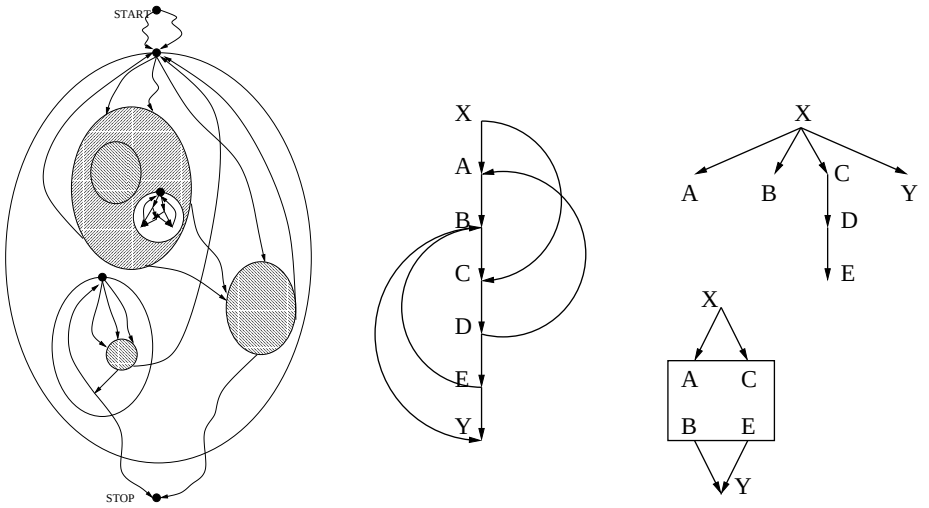


Fig. 4. Left: Example of a Sreedhar-Gao-Lee SDG decomposition. Right: An irreducible graph, its dominator tree, and its strongly connected components.

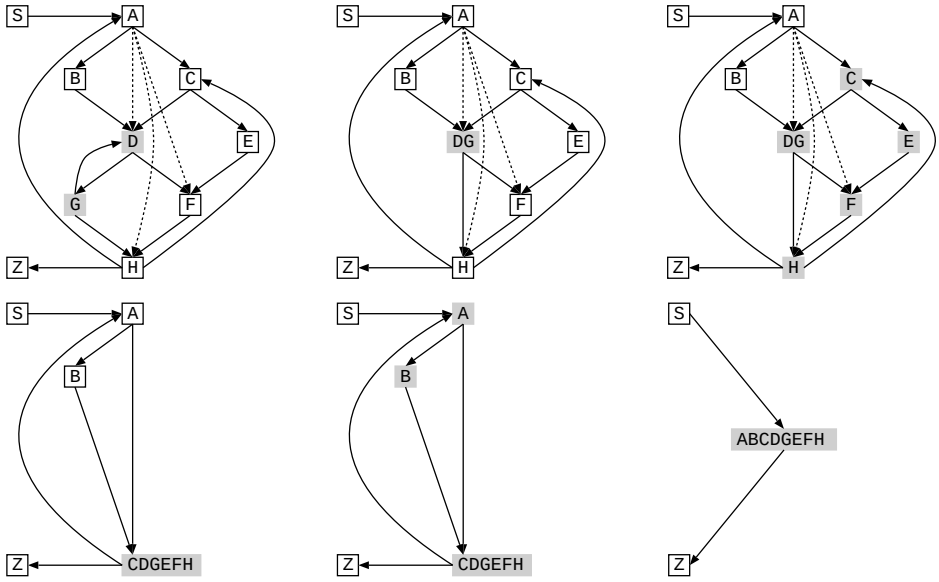


Fig. 5. Bottom-up treatment of nested loops. Dotted arcs are dominator edges.

of overlapping cycles all cycle-free paths between start and target node must be enumerated, which in general generates a number of paths exponential in the size of the graph.

4.4 Exploiting interval analysis

Interval analysis avoids enumerating all paths and thus solves the problem of overlapping cycles. The SGL algorithm determines a hierarchy of reducible as well as irreducible loops. 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. Only in irreducible loops back edges can generate additional path conditions. Thus path enumeration is always limited to an irreducible "egg" (figure 4), therefore the number of paths decreases dramatically. For nested SCCs, the path conditions from inner SCCs are used in outer SCCs according to equation (12). Thus SCCs are processed bottom up. In detail, 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 (15), path conditions can be computed in topological order: the SCC without back edges is cycle-free. For any node $z \in L$, $PC(e, z)$ is computed according to equation (13).
The necessary execution conditions are computed as needed according to equation (2). In case of structured control flow within the SCC, the outer disjunction in this equation becomes redundant, and the control dependencies form a tree. Therefore execution conditions can efficiently be computed according to equation (16). Execution conditions are cached in SDG nodes.
As most of the control flow is structured, the topological ordering of path conditions as well as execution conditions touches each individual $c(x \to y)$, $\delta(x \to y)$ or $\Phi(x \to y)$ only once. Eventually topological order reaches the x_i, thus all $PC(e, x_i)$ can be collected in time $O(|L|)$. Note that this time for constructing path conditions does not include the time for BDD operations; 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 (14) – common prefixes for two paths are thus automatically factored out.
Computation of execution conditions is as in the reducible case. The complexity is $O(p \cdot |L|)$, where p is the number of paths in the SCC (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 (10) or (12). For the purpose of path conditions, SCCs from a lower level are treated as collapsed into one "meganode," where execution and path conditions for this meganode are computed as expressed in equation (12). Note that entry and exit points of SCCs are needed in equation 12 and thus must be propagated up to the next level.
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'.

As an example, consider figure 5, 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 $PC(D,G) = E(D) \wedge E(G)$, $PC(D,D) = E(D)$.[5] 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. According to equation (12),

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

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

$$PC(A,CDGEFH) = E(A) \wedge \big(PC(C,H) \vee E(B) \wedge PC(DG,H)\big)$$

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 infeasible. The empirical data in the next section will illustrate these statements.

4.5 Implementation

The ValSoft system can generate path conditions for full ANSI C (except pointer arithmetic and setjmp/longjmp).[6] The path condition generator was implemented on top of the ValSoft slicer. In order to implemented the algorithm from section 4.3, we implemented the Lengauer/Tarjan fast dominator algorithm [Lengauer and Tarjan 1979] as well as SGL's generalized interval analysis.

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/McCluskey minimizer [Quine 1955] to obtain a minimal disjunctive normal form (MDNF).

[5] In this example, additional δ-constraints are left out.

[6] If programs contain pointer arithmetic or setjmp/longjmp, ValSoft makes very conservative approximations which can easily ruin precision. Fortunately, these constructs are rare; none of our benchmark programs (see section 5) contained pointer arithmetic or setjmp/longjmp.

This MDNF is used for displaying path conditions, and also prevents the subsequent constraint solvers from drowning in huge formulae. Note that computing the MDNF can have exponential time complexity, but our experiments indicate that this poses no problem in practice; in any case, a MDNF is not an absolute requirement (see section 4.2). An interface to the Redlog solver [Dolzmann and Sturm 1997; Sturm and Weispfenning 1996] has been implemented; interfaces to other solvers are in preparation. The solved conditions are displayed to the user in textual form.

The current implementation utilizes the SGL decomposition only in an intraprocedural manner. The dominator tree is computed separately for every procedure, and the SGL "eggs" never cross procedure boundaries. Section 5 will show that even intraprocedural SGL decomposition alone has a very positive effect on performance. It is doubtful that an attempt to determine an interprocedural SGL decomposition will have much additional positive effect for the following reason: if the same procedure is called many times from different places, this will inhibit many interprocedural dominator relationships in the SDG and thus not lead to SCCs which are bigger than a procedure.

In order to introduce an additional control mechanism for precision and performance of path condition generation, we implemented *path length limitation*. For any path in a non-reducible SCC, the number of execution conditions used in equation 1 (and hence the number of path nodes, i.e. the path length) can be limited to $k\%$ of the SCC's node count (where k can be chosen as an analysis parameter, and path nodes are selected in depth-first order in the SCC). The empirical section will show that this has an additional positive effect on performance, while the generated path conditions remain the same in most cases. Note that path length limitation never violates the principle of conservative approximation: according to equation 1, it may generate path conditions which are too weak (see section 2.5), but never generates incorrect (i.e. non-necessary) path conditions.

ValSoft comprises about 75,000 lines of C++, among them 25,000 for path condition generation and simplification (without the BDD package).

5. PERFORMANCE MEASUREMENTS

In this section, we investigate the performance of path conditions based on a set of case studies. In particular, we study the following two questions:

(1) What is the effect of BBDs and interval analysis on performance? Do these techniques scale up?

(2) What is the dominating factor for overall performance: program size, SDG size, SDG structure, chop size, or chop structure?

The experiments in this section are based on 13 programs and 27 chops, thus we do not claim that our results are generally valid. But we think that the experiments display typical behavior of the path condition generator.[7]

Table I presents data for the set of 13 benchmark programs.[8] The criteria for this particular selection of benchmark programs were as follows: (1) The programs

[7] During the final revision of this article, many more experiments and case studies became available, which support the results in this section [Robschink 2005].

[8] *WobbleTableM* is a variant of *WobbleTable* which will be discussed in section 6.

SDG	Nodes	Edges	LOC	Funcs	Calls
mergesort	244	640	59	5	10
calculator	267	716	115	8	15
triple des	5146	20353	1103	22	50
ctags	12958	55290	2933	100	300
assembler	16049	263053	3178	72	523
gnugo	7786	23762	3305	37	293
agrep	22820	79938	3968	87	424
WobbleTable	10590	30210	4563	67	497
WobbleTableM	10695	30641	4571	67	497
flex	48098	688065	7640	119	621
patch	30774	246754	7998	163	856
bison	33158	156926	8313	157	906
larn	171904	1000811	10410	276	2416
moria	364986	1742234	25754	465	4861

Table I. SDG size and structure for various programs.

CHOP	Nodes	Edges	redSCC	irrSCC	Depth	redL	maxN	maxE	irrL	maxN	maxE
mergesort 1	69	156	11	2	14	16	4	7	4	26	53
mergesort 2	99	227	15	3	18	20	4	7	5	26	53
calculator 1	178	487	25	2	10	33	4	6	8	61	123
calculator 2	66	169	7	2	10	10	4	6	5	13	25
triple des 1	897	3265	21	7	12	35	8	14	9	20	49
triple des 2	136	331	1	0	5	1	2	2	0	0	0
ctags 1	4299	16762	381	29	24	566	19	39	118	502	2100
ctags 2	2958	11994	288	7	15	431	55	284	71	502	2100
assembler 1	6169	181470	51	8	12	91	27	56	28	24	49
assembler 2	2704	54900	91	19	18	118	9	16	47	27	56
gnugo 1	4649	14103	582	135	12	659	10	18	232	293	796
gnugo 2	5500	16788	638	139	23	747	10	18	249	293	796
agrep 1	11031	32897	12	2	15	16	98	203	3	94	195
agrep 2	10954	32539	108	4	16	150	336	794	16	316	729
WobbleTable	3466	9865	482	63	21	601	15	22	224	172	564
WobbleTableM	3413	9835	475	68	23	593	174	580	233	173	572
flex 1	6014	43880	63	20	16	81	13	17	39	180	378
flex 2	10482	248099	55	20	9	64	6	10	48	23	46
flex 3	10476	248041	0	0	4	0	0	0	0	0	0
patch 1	14000	128527	1277	283	23	1674	1309	30864	672	1308	30859
patch 2	13943	128271	1273	277	25	1684	1310	30951	657	1309	30864
bison 1	5742	51658	407	130	23	528	97	253	245	440	24154
bison 2	5378	50043	392	110	23	508	97	253	218	440	24154
larn 1	36382	222695	9	0	8	15	5	8	2	11	23
larn 2	36451	222864	10	0	8	16	5	8	2	11	23
moria 1	15376	59571	168	4	15	207	1198	4084	40	1185	4063
moria 2	16644	63194	21	1	13	34	6	10	3	11	18

Table II. Chop structure for various path conditions.

are written in ANSI C, and contain neither pointer arithmetic nor setjmp/longjmp. (2) The programs cover a wide variety of Unix programming styles, not just safety-critical systems (which normally disallow pointers). The first criterion mirrors requirements of the current ValSoft implementation. The second criterion mirrors the fact that path conditions are helpful for any program, not just safety-critical systems. Thus, we included standard Unix programs with heavy pointer usage; the only safety critical system is *WobbleTable*. None of the case studies uses multi-threading. *patch* and *ctags* contain ordinary C asserts, XASSERTs (see Section 3.7)

ChOP	Disj	Conj	Neg	Cond	BddN	BddV	BddVR	Time(s)	Mem(MB)
mergesort 1	2	14	2	17	261	8	6	0	0.9
mergesort 2	4	20	1	25	362	13	10	1	1.0
calculator 1	0	1	0	2	253	9	2	0	1.4
calculator 2	0	2	0	3	209	6	3	0	0.8
triple des 1	3	24	4	28	372	13	8	1	4.6
triple des 2	0	1	1	2	207	4	2	1	3.8
assembler 1	17	177	96	195	373	21	14	40	21.7
assembler 2	41	581	214	623	926	46	15	6	15.9
agrep 1	0	4	4	5	209	5	5	4	20.8
agrep 2	1	12	5	14	416	25	8	17	21.0
WobbleTable	14	323	126	338	4112	132	16	47	11.6
WobbleTableM	11	148	63	160	4535	128	15	38	10.7
flex 2	0	1	2	2	848	59	2	78	40.0
flex 3	379	9515	4200	9895	17185	101	34	1901	357.9
bison 1	2	12	3	15	322	10	6	11	20.3
larn 1	1	9	5	11	426	54	6	148	79.3
larn 2	1	1	1	3	455	55	2	150	79.4
moria 2	0	2	0	3	3102	220	3	63	99.9

Table III. Performance for various path conditions in the standard algorithmic configuration: SGL decomposition and BDDs are active. The data for *ctags 1, ctags 2, gnugo 1, gnugo 2, flex 1, patch 1, patch 2, bison 2, moria 1* are not shown as the run times were more than one hour.

	−BDD +SCC		+BDD +SCC		+BDD +S.G.L.	
CHOP	Time(s)	Mem(MB)	Time(s)	Mem(MB)	Time(s)	Mem(MB)
mergesort 1	0	0.9	0	0.9	0	0.9
mergesort 2	0	1.0	1	1.0	1	1.0
calculator 1	227	208.7	1	1.3	0	1.4
calculator 2	0	0.8	0	0.8	0	0.8
triple des 1	0	4.6	1	4.6	1	4.6
triple des 2	0	3.8	0	3.8	1	3.8
assembler 1	∞	∞	50	21.7	40	21.7
assembler 2	∞	∞	12	15.9	6	15.9
agrep 1	7	20.8	7	20.8	4	20.8
agrep 2	∞	∞	∞	∞	17	21.0
WobbleTable	∞	∞	3332	10.1	47	11.6
WobbleTableM	∞	∞	339	10.2	38	10.7
flex 2	∞	∞	95	40.0	78	40.0
flex 3	∞	∞	∞	∞	1901	357.9
bison 1	26	26.2	14	20.4	11	20.3
larn 1	208	79.3	167	79.3	148	79.3
larn 2	207	79.4	167	79.4	150	79.4
moria 2	86	99.9	69	99.9	63	99.9

Table IV. Performance for some algorithmic variants: only top level SCC decomposition, without and with BDDs; vs. standard configuration with active decomposition.

have not been used.

The table provides data about SDG size, program size, and number of function definitions and calls. Table II presents information about 27 chops which have been selected for computing path conditions.[9] All measurements were done on a standard 1GHz PC with 2GB of RAM. For every program, two or three chops were randomly chosen and then selected according to the following criteria:

[9]The numbers presented in this section differ from the numbers presented by Robschink and Snelting [2002], as they were computed with points-to analysis activated and the improved interprocedural chopping algorithm from Krinke [2002].

CHOP	Disj	Conj	Neg	Cond	BddN	BddV	BddVR	Time(s)	Mem (MB)	k(%)
mergesort 1	2	14	2	17	255	8	6	0	0.8	5.0
	2	14	2	17	255	8	6	0	0.8	0.5
mergesort 2	4	20	1	25	358	13	10	0	0.9	5.0
	4	20	1	25	358	13	10	0	0.9	0.5
calculator 1	0	1	0	2	225	9	2	0	1.0	5.0
	0	1	0	2	217	9	2	1	1.0	0.5
calculator 2	0	2	0	3	209	6	3	0	0.8	5.0
	0	2	0	3	209	6	3	0	0.8	0.5
triple des 1	3	24	4	28	363	13	8	1	4.6	5.0
	3	24	4	28	363	13	8	0	4.6	0.5
triple des 2	0	1	1	2	207	4	2	0	3.8	5.0
	0	1	1	2	207	4	2	0	3.8	0.5
ctags 1	1	1	2	3	547	81	2	11	11.2	5.0
	1	1	2	3	1056	81	2	10	11.2	0.5
ctags 2	1	15	3	17	391	36	9	5	9.6	5.0
	1	11	3	13	370	36	7	6	9.6	0.5
assembler 1	17	177	96	195	373	21	14	40	21.7	5.0
	17	177	96	195	373	21	14	40	21.7	0.5
assembler 2	41	581	214	623	811	39	15	6	15.9	5.0
	41	581	214	623	811	39	15	6	15.9	0.5
gnugo 1	0	1	0	2	83855	322	2	437	10.2	5.0
	0	1	0	2	18090	316	2	7	10.2	0.5
gnugo 2	0	9	2	10	83797	354	10	329	11.1	5.0
	0	9	2	10	83797	354	10	329	11.1	0.5
agrep 1	0	4	4	5	209	5	5	5	20.8	5.0
	0	4	4	5	209	5	5	4	20.8	0.5
agrep 2	1	12	5	14	384	25	8	18	21.0	5.0
	1	12	5	14	437	25	8	4	21.0	0.5
WobbleTable	14	323	126	338	5237	129	16	39	11.0	8.5
	11	175	77	187	5892	129	23	19	12.8	5.0
	0	4	3	5	348	21	5	2	9.4	0.5
WobbleTableM	11	148	63	160	5889	125	15	29	10.4	5.0
	11	148	63	160	1248	60	15	3	9.6	0.5
flex 1	21	191	85	213	1415	27	9	14	30.1	5.0
	21	191	85	213	757	27	9	8	30.1	0.5
flex 2	0	1	2	2	848	59	2	74	40.0	5.0
	0	1	2	2	848	59	2	74	40.0	0.5
flex 3	379	9515	4200	9895	17185	101	34	1918	357.9	5.0
	379	9515	4200	9895	17185	101	34	1898	357.9	0.5
patch 1	103	1734	967	1838	906	25	16	23	33.6	0.1
	103	1734	967	1838	906	25	16	24	33.6	0.05
patch 2	71	1348	404	1420	1036	155	27	139	36.9	0.1
	71	1348	404	1420	1004	154	27	133	36.8	0.05
bison 1	2	12	3	15	322	10	6	11	20.3	5.0
	2	12	3	15	322	10	6	11	20.3	0.5
bison 2	0	9	3	10	181707	344	10	1337	25.0	2.5
	0	9	3	10	188652	341	10	134	24.7	0.5
larn 1	1	9	5	11	426	54	6	148	79.3	5.0
	1	9	5	11	426	54	6	147	79.3	0.5
larn 2	1	1	1	3	455	55	2	150	79.4	5.0
	1	1	1	3	455	55	2	149	79.4	0.5
moria 1	0	0	0	1	202	2	0	27	99.4	0.1
	0	0	0	1	202	2	0	27	99.4	0.05
moria 2	0	2	0	3	3102	220	3	63	99.9	5.0
	0	2	0	3	3102	220	3	64	99.9	0.5

Table V. Effects of limiting the path length to k% of SCC node count (SGL and BDDs active).

—chops should not be too small compared to the SDG size, i.e. the number of chop nodes is at least 5% of the number of SDG nodes;

—chops with obvious simple path conditions have been excluded, therefore start and end node should be deeply nested in the control flow.

The table provides the number of chop nodes and edges as well as structural information. First, the top-level SCCs were determined; *redSCC* is the number of top level reducible SCCs, and *irrSCC* is the number of irreducible top-level SCCs. The rest of the columns are concerned with the SGL decomposition, which determines

not only top level SCCs, but nested SCCs as explained in section 4.3. Columns *redL* and *irrL* give the numbers of nested reducible resp. irreducible SCCs, and of course $redSCC \leq redL$ and $irrSCC \leq irrL$ always hold. Column *Depth* displays the maximal depth over all intraprocedural dominator trees (remember that dominators and SGL decompositions are computed only intraprocedurally). Columns *maxN* and *maxE* present the number of nodes and edges in the biggest reducible resp. irreducible SCC in the SGL decomposition.

For the 27 chops, the number of SCCs varies widely. While the only safety-critical program, *WobbleTable* has a large number of small decomposable SCCs, the *flex 3* chop is not decomposable at all, and the chops for *patch* have very big SCCs, some with a node/edge ratio of 1:20 or less.[10] The latter two scenarios are indicative of complex program structure, as there are lots of interfering dependencies from unstructured control flow, unstructured data flow, or unstructured pointer usage. For such programs, generation of path conditions is expected to be expensive.

Table III presents running times and memory requirements for the path condition examples. These tables were determined with active BDDs and active SGL decomposition, but without path length limitation. 9 out of 27 path conditions could not be determined within one hour. Among those is *bison 2*, which has – as manual inspection revealed – a very bad decomposition structure compared to *bison 1*, resulting in a huge performance difference. For the other 18 chops, we see times in the range of one minute and moderate space requirements. Comparing the data to the chop structure, the case studies indicate

—Path conditions for decomposable chops with small SCCs are easily determined with reasonable effort; chop size is less important than chop structure.

—Non-decomposable chops (*flex 3*) are difficult to analyze, but even worse is a large number of irreducible SCCs with bad structure (i.e. bad node/edge ratio as in *gnugo* or *ctags*).

The table also contains information about the structure of the path condition and the structure of the BDD. Typical path conditions have less than one hundred or at most a few hundred conjunctions and disjunctions, but the non-decomposable *flex 3* has a few thousand. Some path conditions are very small (e.g. *flex 2*) even though the chop is quite big; this happens whenever path conditions from alternating if-then-else paths cancel each other out. The number of intermediate BDD nodes (*BddN*) and variables (*BddV*) compared to the final number of BDD variables (*BddVR*) shows how the intermediate BDDs collapse for the final path condition.

Table IV demonstrates the effect of BDDs and SGL decomposition. Here, ∞ means that the analysis ran out of memory. Using simple syntax trees instead of BDDs (left columns), many of the examples from table III were not analyzable at all. With BDDs, but using only a simple top-level SCC decomposition (middle columns) the time requirements are much higher than with BDDs and SGL decomposition (right columns, repeated from table III).

Table V demonstrates the effect of path length limitation. Path length is limited to an amount between 0.5% und 8.5% of the chop node count. Compared to table

[10] The nesting structure of the SCCs is not visible in the table, but was manually analyzed for the three examples.

III, another spectacular improvement in runtime behavior is visible, which was to be expected. In particular, all 27 path conditions can now be computed in less than an hour, and most of them in less than a minute. Of course, the 0.5% limitation is faster than the 5% limitation.

Section 4.5 explained that path length limitation can make path conditions less precise, but never incorrect. Surprisingly, the structural information about the path conditions demonstrates that for 25 out of 27 chops, the path conditions remain unchanged. Only for *WobbleTable* and *ctags 2* there is a difference: for *WobbleTable*, the 8.5% limitation results in the same path condition as the unlimited variant from table III, while the 5.0% limitation is slightly less and the 0.5% sharply less precise than the unlimited variant.[11] For the "bad guys" *gnugo*, *ctags* etc., the "unlimited" path conditions are not available, but as there are no differences between 5.0% and 0.5% limitation (except for *ctags 2*, where there is a slight difference), we would be surprised if the "unlimited" path conditions were more precise.

Hence table V demonstrates that in practice, length limitation does not influence precision. The engineer can start with small values for k and increase k for interesting path conditions until the path condition does not change anymore. The combination of BDDs, SGL decomposition and path length limitation guarantees that all programs can be analyzed: path conditions do scale up.

6. 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. *WobbleTable* 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. We chose *WobbleTable* as an example of our methodology, for although *WobbleTable* is not a safety critical system (such as a shutdown system for a nuclear power plant) it exhibits many characteristics of such systems.

The source file is 4563 LOC of ANSI C; computation of the SDG took 15 seconds. For some library functions concerned with camera and motor control, C stubs were provided which simulate the function's behavior with respect to data and control dependencies between parameters and global variables.[12] In our experiment, we wanted to check whether the step motor is influenced by an outside agent, and if so determine witnesses for suspicious behavior.

Figure 6 displays the central loop of the source code. While the ball did not reach the target, the ball position is read from the camera and converted to maze coordinates (function "getDPoint", line 4). The function "pathNPos" computes the distance to the next intermediate ball position, and the function "getEngSteps" uses a

[11] The table only shows that the "limited" conditions are smaller. But as all variants are necessary conditions, "smaller" indicates "less precise". This was manually checked for most of the examples.
[12] Providing stubs is a popular way to deal with libraries, but quite expensive in practice: libraries are big, thus many stubs are required; worse, source code is not always available, forcing the analysis to use imprecise approximations.

```
...
path = getPath();  (line 4)
if (path == 0l) {
  ...
}
oldCenter = path->root->pos;
startPos = path->root->pos;
controlInit();

while (notTarget) {
  ret = getDPoint(center);
  if (ret == 0) {
    notTarget = 0;
    continue;
  }
  if (abs(platte.x) > 250
      || abs(platte.y) > 250) {
    ...
    notTarget = 0;
    continue;
  }
  notTarget = pathNPos(path, center,
                       dist, speed);
  new_target = path->root->pos;
  speed->x = center->x - oldCenter.x;
  speed->y = center->y - oldCenter.y;
  getEngSteps(dist->x, speed->x, table.x,
              dist->y, speed->y,
              table.y, &steps_x, &steps_y);
  vect = calcCntrlVect(steps_x, steps_y);
  if (notTarget) {
    sendEngSteps(vect, 0, vectlen);
    table.x = table.x + steps_x;
    table.y = table.y + steps_y;
  }
  free(vect);
  oldCenter = *center;
}
steps_x = -table.x;
steps_y = -table.y;
vect = calcCntrlVect(steps_x, steps_y);
sendEngSteps(vect, vectlen, 0);   (line 45)
...
```

Fig. 6. Source code of central wobble loop.

neural net to compute the rotation of the maze. Function "calcCtrlVect" transforms this information into a control vector which is sent to the motor ("sendEngSteps", line 45); the maze angles are adjusted accordingly.

Figure 7 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. For all atomic conditions in this path condition, their source file and source line is given.[13] Path conditions are LaTeXed automatically; Φ-constraints without disjunctions (i.e. simple value propagations) are automatically substituted. SSA indices of program variables (italic font) and function return values (bold font) are usually shown; in the implementation, they are used as back links to the source code available on mouse click. The condition was not fed into a constraint solver, as it is already in solved form. Performance data for this chop is given in table III, line 11.

The condition is surprisingly small when compared to the program size and becomes quite clear after a look at the source code (and after determining the source positions of the SSA indices). The first part of the condition requires that the target has not been reached, the ball has a definite position, the vertical angles of the maze in x- and y direction do not exceed a value of 250 "steps", that the next intermediate target for the ball is defined, and that the distance between ball and intermediate target does not exceed a maximum value. The inner disjunction demands that either the ball velocity is bounded and the next intermediate target is 0 (i.e. the target has been reached), 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.

[13] The path conditions presented in this section differ slightly from the path conditions in [Robschink and Snelting 2002], as they were computed with activated points-to analysis and an improved interprocedural chopping algorithm [Krinke 2002], leading to smaller chops and more precise path conditions.

$$
\begin{aligned}
PC(4,45) \equiv\ & \\
& notTarget_{9263} = TRUE \qquad (\text{control.c} : 200) \\
\wedge\ & \mathrm{getDPoint}_{9267}(\mathrm{calloc}_{9093}(1,8)) \neq 0 \\
& \quad (\text{control.c} : 160, 203, 206) \\
\wedge\ & |table_{9289}.x_{9290})| \leq 250 \qquad (\text{control.c} : 213) \\
\wedge\ & |table_{9296}.y_{9297})| \leq 250 \qquad (\text{control.c} : 213) \\
\wedge\ & \mathrm{pathNPos}_{9325}(\mathrm{getPath}_{9132}(\ldots), \mathrm{calloc}_{9093}(1,8), \\
& \quad \mathrm{calloc}_{9100}(1,8), \ldots) = TRUE \qquad (\text{control.c} : 160 \ldots, 177, 226) \\
\wedge\ & \mathrm{sqrt}\Big((\mathrm{getPath}_{9132}(\ldots).root_{7874}.pos_{7876}.x_{7878} - \mathrm{calloc}_{9093}(1,8).x_{7880}) * \\
& \quad (\mathrm{getPath}_{9132}(\ldots).root_{7884}.pos_{7886}.x_{7888} - \mathrm{calloc}_{9093}(1,8).x_{7890}) + \\
& \quad (\mathrm{getPath}_{9132}(\ldots).root_{7895}.pos_{7897}.y_{7899} - \mathrm{calloc}_{9093}(1,8).y_{7901}) * \\
& \quad (\mathrm{getPath}_{9132}(\ldots).root_{7905}.pos_{7907}.y_{7909} - \mathrm{calloc}_{9093}(1,8).y_{7911})\Big) \\
& \quad < MAX\_TARGETDIST_{9330} \qquad (\text{pfad.c} : 71, \text{control.c} : 71 - 74, 160, 177, 226) \\
\wedge\ & i_{208} < |-table_{9570}.x_{9571}| + |-table_{9575}.y_{9576}| \\
& \qquad\qquad (\text{calc.c} : 80, 157, 137, \text{control.c} : 270, 271) \\
\wedge\ & \Big(\\
& \quad |\mathrm{calloc}_{9107}(1,8).x_{7923}| < MAX\_TARGETSPEED_{9331} \\
& \qquad (\text{pfad.c} : 111, \text{control.c} : 162, 226) \\
& \quad \wedge\ |\mathrm{calloc}_{9107}(1,8).y_{7932}| < MAX\_TARGETSPEED_{9331} \\
& \qquad (\text{pfad.c} : 112, \text{control.c} : 162, 226) \\
& \quad \wedge\ \mathrm{getPath}_{9132}(\ldots).root_{7941}.next_{7943} = 0 \\
& \qquad (\text{pfad.c} : 115, \text{control.c} : 177) \\
& \quad \vee \\
& \qquad i_{4518} < noNeurons_{6160} \qquad (\text{neuronal.c} : 1101, 1108) \\
& \quad \wedge\ NeuronsInLayer_{9424}[0] \neq 2 \qquad (\text{neuronal.c} : 1348) \\
& \quad \wedge\ NeuronsInLayer_{9424}[0] \neq 3 \qquad (\text{neuronal.c} : 1390) \\
& \quad \wedge\ NeuronsInLayer_{9424}[0] = 8 \qquad (\text{neuronal.c} : 1434) \\
& \Big)
\end{aligned}
$$

Fig. 7. Path condition for step motor.

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 `key` was already declared and easy to spot. According to the programmers, `key` was used in a debugging version, but all references to `key` were removed later. Indeed, in the existing program there is no SDG path from the initialization $def(\texttt{key})$ (not visible in figure 6) to the motor control call in line 45, thus trivially $PC(def(\texttt{key}), 45) \equiv false$.

After introduction of the manipulation, the path condition $PC(def(\texttt{key}), 45)$ was computed again. The result is no longer $false$, but the condition in figure 8; computation of this path condition took 38 seconds. Thus we already know that there is a possible influence from the keyboard to the step motor. The path condition contains various constraints similar to figure 7, as well as the atomic control condition $ping_{3499}\&128 > 0$. Global Φ-constraints (displayed below the path condition) state that $ping_{3499}$ is the memory cell referred to by $ping_{3498}$, and

$$
\begin{aligned}
PC(def(\texttt{key}),45) \;\equiv\;& \\
& i_{207} < |m\_x_{582}| + |m\_y_{583}| \qquad (\texttt{calc.c} : 80, 157, 137) \\
\wedge\;& \mathrm{Layer}_{3591}(neuron\_id_{3497}) \neq 0 \qquad (\texttt{neuronal.c} : 696, 719) \\
\wedge\;& \underbrace{*ping_{3498}\&128 > 0 \qquad (\texttt{neuronal.c} : 696, 731)}_{\text{possible manipulation}} \\
\wedge\;& notTarget_{9361} = TRUE \qquad (\texttt{control.c} : 202) \\
\wedge\;& \mathrm{getDPoint}_{9365}(\mathrm{calloc}_{9188}(1,8)) \neq 0 \\
& \qquad (\texttt{control.c} : 160, 205, 208) \\
\wedge\;& |table_{9387}.x_{9388})| \leq 250 \qquad (\texttt{control.c} : 215) \\
\wedge\;& |table_{9394}.y_{9395})| \leq 250 \qquad (\texttt{control.c} : 215) \\
\wedge\;& \mathrm{pathNPos}_{9423}(\mathrm{getPath}_{9230}(\ldots), \mathrm{calloc}_{9188}(1,8), \\
& \qquad \mathrm{calloc}_{9195}(1,8),\ldots) = TRUE \qquad (\texttt{control.c} : 160\ldots, 179, 228) \\
\wedge\;& \big(\\
& \quad NeuronsInLayer_{5298}[0] = 2 \qquad (\texttt{neuronal.c} : 1338) \\
& \vee\; NeuronsInLayer_{5298}[0] = 3 \qquad (\texttt{neuronal.c} : 1338) \\
& \vee\; NeuronsInLayer_{5298}[0] = 8 \qquad (\texttt{neuronal.c} : 1338) \\
& \big)
\end{aligned}
$$

$$
\begin{aligned}
\Phi \;\equiv\;& m\_x_{582} = -table_{9671}.x_{9672}, \\
& m\_y_{583} = -table_{9676}.y_{9677}, \qquad (\texttt{calc.c} : 137, \texttt{control.c} : 272, 273) \\
& neuron\_id_{3497} = \mathrm{firstNeuron}_{5373}(noLayers_{9523} - 1), \\
& \qquad (\texttt{neuronal.c} : 696, 1448) \\
& ping_{3498} = ping_{5492} = ping_{5291}, \\
& ping_{5291} = ping_{9517} = key_{9224} = key_{9184} \text{ (real definition)} \\
& \qquad (\texttt{control.c} : 158, 171, 240, \texttt{neuronal.c} : 696, 1316, 1363)
\end{aligned}
$$

Fig. 8. Path condition revealing a safety violation.

that there is a pointer chain which ultimately states that this pointer is equal to key_{9184}. SSA indices can be used as back references to the source text: key_{9184} is indeed *def*(`key`), $ping_{5291}$ is a formal parameter of "getEngSteps". Thus we see that the step motor is manipulated by the keyboard input via variable `ping`.

Following the back links to the source code, we immediately see 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 `control.c` they added the statement `ping = (int*) key;`. Deeply hidden inside `neuronal.c` they added the statement

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

which increases the scale factor in the neural net by 20% if the 8th bit of `*ping` (that is, `key`) is set. Interestingly, the variable `val` does not occur in the path condition, as it is never used in any control condition. But the SSA index $ping_{3499}$ 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 discovering such a manipulation!

Summarizing this case study, we would like to point out the following facts:

—Understanding path conditions requires some, but not deep understanding of the source code. Indeed the engineer can use a debugger, provide values for the variables as expressed in the path condition, and the safety violation becomes visible: the path condition acts as a witness.

—Witnesses make visible the reason for an influence, and allow the engineer to decide whether the influence is legal or not. Manipulating the step motor in *WobbleTable* is certainly illegal, while manipulating a weight value as in figure 3 may be legal if it just serves to switch between grams and ounces.

—Witnesses are useful in legal matters, e.g. a lawsuit against a software vendor. Using the witness, one can see the safety violation directly and must not understand artifacts like model checking counterexamples or type mismatches in eclectic type systems (see also section 7).

7. RELATED WORK

Our work is similar in spirit to constraint-based test data generation (e.g. [Gotlieb et al. 1998; Gupta et al. 1998; Goldberg et al. 1994; deMillo and Offut 1991]). All such methods 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 information flow control. Most methods (e.g. [Gotlieb et al. 1998; Gupta et al. 1998]) have only been applied to small programs, while we emphasize scaling up. Some (e.g. [deMillo and Offut 1991]) do not obey the principle of conservative approximation required for safety analysis. Some are restricted to specialized domains (e.g. [Gupta et al. 1998]); our approach provides a general path condition generator which can then be connected to specialised solvers.

Parametric program slicing [Field et al. 1995] generalizes static and dynamic slicing by allowing the specification of arbitrary constraints over input variables (similar to our assertions from section 3.7). A parametric slice is valid for all inputs satisfying the constraints. Parametric slicing requires that the language semantics is defined in terms of rewrite rules, augments these rules by the given constraints and determines the slice during rewriting. Unfortunately, applications to realistic programs have not been reported. Conditioned slicing [Canfora et al. 1998], a very similar technique, also shares the same problems with realistic applications.

PREfix [Bush et al. 2000] analyzes the control flow and builds a memory model to discover bugs like memory leaks or dangling references. It can also generate simple path conditions for such bugs – but again, these are based on control flow rather than on data flow. ESC/Modula3 [Rustan et al. 1998] finds similar bugs by applying verification technology, but requires that the programmer adds assertions to the program. Our approach does not require assertions and is aimed at information flow control rather than detection of low-level bugs.

Pugh and Wonnacott [1998] use 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 [2000] also investigated the use of abstract data types in dependence graphs. He extends his technique of context-free language reachability to model equations for abstract types. It turns out that interprocedural data dependence becomes undecidable. Our approach, on the other hand, is based on rewriting modulo data dependencies, which is a mechanism completely orthogonal to dependence analysis. While perhaps less precise, it avoids any decidability problems and is completely decoupled from the rest of the path condition generator.

Smith and Volpano [1998] presented a type system for an imperative language with threads which can be used to check the Bell/La Padula condition. It is a type-based implementation of the approach by Denning and Denning [1977], who assumed that security domains form a lattice and presented a non-standard semantics for a simple language to determine information flow between different security levels. Compared to slicing and path conditions, Denning's original approach and the Smith/Volpano method are flow-insensitive and hence miss some of the information present in slices and path conditions.

Another type-based approach is the CQual system [Foster et al. 2002]. CQual is flow-sensitive, as flow information is coded into types. It has successfully been used to detect locking bugs in the Linux kernel. While CQual requires annotations, it may be that it can be used to improve the Smith/Volpano method (to our knowledge, this has not been done). However, illegal information flow shows up as a type error in a nonstandard type system, while path conditions can be used as witnesses which make illegal behavior visible directly. Considering the usefulnes of program analysis, for example in a lawsuit, we believe that our "witnesses" are more convincing to the judge than an abstract type mismatch.

The recent overview article [Sabelfeld and Myers 2003] presents even more work on information flow control based on program analysis. The focus is again on type-based methods, and while data flow analysis and program slicing are mentioned, the true value of these techniques (and improvements such as path conditions) for safety analysis has in our opinion not yet been recognized.

There are several generators of static analysers such as PAG [Martin 1998], TVLA [Lev-Ami and Sagiv 2000], or Lande [Metayer and Schmidt 1996]. In principle, these could be used to implement path conditions. This requires a formal semantics for full ANSI C, which, to our knowledge, has not been constructed for any of these systems. Furthermore, we suspect that the generated systems do not scale up, as the algorithmic techniques from chapter 4 are not available.

Recently, model checking has gained popularity as a device to check certain safety properties of programs. The Bandera project [Corbett et al. 2000] as well as SLAM [Ball and Rajamani 2002] extract finite models from software, which can then automatically be checked against specifications in temporal logics. While model checking is certainly a most useful instrument, ordinary model checking cannot be used for manipulation detection: during model extraction, illegal information flow might get abstracted away. Note however that an SDG can be seen as a nonstandard finite model, on which LTL formulae can be checked – an idea we plan to explore in the future.

8. CONCLUSIONS 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 concentrates on fundamental techniques as well as practical possibilities of path conditions. Our results can be summarized as follows:

(1) Path conditions can reduce the imprecision of slicing, and can demonstrate that some slices are in fact impossible;
(2) Subsequent constraint solving generates witnesses for specific information flow, in particular for illegal influences to safety-critical computations;
(3) Naive generation of path conditions does not scale, interval analysis and BDDs are the key devices for taming complexity;
(4) The improved path condition generator produced a witness for a safety violation in a medium-sized C program in less than a minute.
(5) In contrast to other methods for information flow analysis, our witnesses for illegal information flow are a natural device understandable by non-experts.

Of course, our work is not finished at this point. In particular,

—We want to make path conditions work for chops with more than 10^6 edges in minutes instead hours. The first step towards even better scale-up will be to improve the Sreedhar decomposition, perhaps by using the new algorithm from Ramalingam [1999].
—We want to compare the behavior of various solvers, such as Redlog [Dolzmann and Sturm 1997], Mathematica [Wolfram 1999], Pugh's Omega test [Pugh and Wonnacott 1998], and constraint logic programming. In fact path conditions can be classified according to syntactic criteria and "the best" solver can be selected automatically.
—We want to generalize path conditions by feeding in external information such as constraints from formal specifications, dynamic input values or traces (similar in spirit to dynamic slicing), or slicing barriers [Krinke 2004].
—We want to adapt and extend ValSoft for Java; this requires static approximation of dynamic lookup behavior for slicing, and generation of corresponding path conditions.

Applying path conditions in dependence graphs to more case studies is our highest priority. 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 up would not have been possible without earlier work in slicing algorithms, points-to analysis, interval analysis, BDDs, and constraint solvers; we collectively thank all the researchers involved in these topics. We also would like to thank the reviewers for their substantial and helpful comments on an earlier version of this article.

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

REFERENCES

Agrawal, H., DeMillo, R. A., and Spafford, E. H. 1991. Dynamic slicing in the presence of pointers, arrays and records. In *Proceedings of the ACM Fourth Symposium on Testing, Analysis and Verification (TAV4)*. ACM Press, New York, 60–73.

Baader, F. and Nipkow, T. 1998. *Term rewriting and All That*. Cambridge University Press.

Ball, T. and Rajamani, S. K. 2002. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*. ACM Press, 1–4.

Bell, D. and La Padula, L. 1973. Secure computer systems: mathematical foundations. *MITRE Technical Report 2547*.

Benhamou, F. and Colmerauer, A. 1993. *Constraint Logic Programming: Selected Research*. MIT Press.

Bent, L., Atkinson, D. C., and Griswold, W. G. 2000. A comparative study of two whole program slicers for C. Tech. Rep. CS2000-0643, University of California, San Diego, Computer Science and Engineering.

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

Berzins, V. 1995. *Software Merging and Slicing*. IEEE Computer Society Press, Los Alamitos.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 677–691.

Burke, M., Carini, P., Choi, J.-D., and Hind, M. 1995. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Lecture Notes in Computer Science, 892*, D. Gelertner, A. Nicolau, and D. Padua, Eds. Springer-Verlag.

Bush, W., Pincus, J., and Sielaff, D. 2000. A static analyzer for finding dynamic programming errors. *Software Practice and Experience 30*, 775–802.

Canfora, G., Cimitile, A., a. De Lucia, and Lucca, G. 1998. Conditioned program slicing. *Information and Software Technology 40*, 595–607. Special Issue on Program Slicing.

Common Criteria Project Sponsoring Organizations. 2004. Common criteria for information technology security evaluation. CCIMB-2004-01-001, Version 2.2, Revision 2561 (Jan.).

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, and Zheng, H. 2000. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*. ACM Press, 439–448.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 451–490.

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

Denning, D. and Denning, P. 1977. Certification of programs for secure information flow. *Communications of the ACM 20(7)*, 504–513.

Dolzmann, A. and Sturm, T. 1997. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin 31(2)*, 2–9.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9,* 3 (July), 319–349.

Field, J., Ramalingam, G., and Tip, F. 1995. Parametric program slicing. In *Proc. 22nd Symposium on Principles of Programming Languages (POPL'95)*. ACM SIGPLAN-SIGACT, 379–392.

Foster, J., Terauchi, T., and Aiken, A. 2002. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation 2002*. ACM Press.

Goguen, J. and Meseguer, J. 1984. Interference control and unwinding. In *Proc. Symposium on Security and Privacy*. IEEE, 75–86.

Goldberg, A., Wang, T. C., and Zimmerman, D. 1994. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*. ACM Press, 80–94.

Gotlieb, A., Botella, B., and Rueher, M. 1998. Automatic test data generation using constraint solving techniques. In *Proc. International Symposium on Software Testing and Analysis*. ACM, 53–62.

Gupta, N., Mathur, A., and Soffa, M. L. 1998. Automated test data generation using an iterative relaxation model. In *Proc. International Symposium on Foundations of Software Engineering*. ACM, 231–244.

Horwitz, S. B., Reps, T. W., and Binkley, D. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst. 12,* 1 (Jan.), 26–60.

Knoop, J., Steffen, B., and Vollmer, J. 1996. Parallelism for free: Efficient and optimal bitvector analyses for parallel programms. *ACM Trans. Program. Lang. Syst. 18,* 3.

Krinke, J. 1998. Static slicing of threaded programs. In *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 35–42.

Krinke, J. 2002. Evaluating context-sensitive slicing and chopping. In *Proceedings of the International Conference on Software Maintenance*. IEEE, 22–31.

Krinke, J. 2003a. Advanced slicing of sequential and concurrent programs. Ph.D. thesis, Universität Passau.

Krinke, J. 2003b. Context-sensitive slicing of concurrent programs. In *Proc. FSE/ESEC*. ACM Press, 178–187.

Krinke, J. 2004. Slicing, chopping and path conditions with barriers. *Software Quality Journal 12,* 4.

Krinke, J. and Snelting, G. 1998. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, 661–675. Special issue on Program Slicing.

Lengauer, T. and Tarjan, R. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transaction on Programming Languages and Systems*, 121–141.

Lev-Ami, T. and Sagiv, S. 2000. TVLA: A system for implementing static analyses. In *Static Analysis Symposium*. 280–301.

Lind-Nielsen, J. 2001. BuDDy - a binary decision diagram package. Tech. rep., University of Copenhagen. http://www.itu.dk/reserach/buddy.

Mantel, H., Stephan, W., Ullmann, M., and Vogt, R. 2000. Leitfaden für die Erstellung und Prüfung formaler Sicherheitsmodelle im Rahmen von ITSEC und Common Criteria. Tech. rep., Bundesamt für Sicherheit in der Informationstechnik und Deutsches Forschungszentrum für Künstliche Intelligenz. Version 0.8.

Marriott, K. and Stuckey, P. 1998. *Programming with Constraints*. MIT Press.

Martin, F. 1998. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer 2,* 1, 46–67.

McDowell, C. E. and Helmbold, D. P. 1989. Debugging concurrent programs. *ACM Computing Surveys 21,* 4.

Metayer, D. L. and Schmidt, D. 1996. Structural operational semantics as a basis for static program analysis. *ACM Computing Surveys 2,* 2 (june), 340–343.

Naumovich, G. and Avrunin, G. S. 1998. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceddings of the Sixth ACM Symposium on Foundations of Software Engineering (FSE'98)*. 24–34.

Ottenstein, K. J. and Ottenstein, L. M. 1984. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. Vol. 19(5). 177–184.

Pugh, W. and Wonnacott, D. 1998. Constraint-based array dependency analysis. *ACM Transaction on Programming Languages and Systems*, 1248–1278.

Quine, W. 1955. A way to simplify truth functions. *American Mathematical Society 62*, 627–631.

Ramalingam, G. 1999. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 175–188.

REPS, T. 1998. Program analysis via graph reachability. *Information and Software Technology*, 701–726. Special issue on program slicing.

REPS, T. 2000. Undecideability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems*, 162–186.

REPS, T., HORWITZ, S., SAGIV, M., AND ROSAY, G. 1994a. Speeding up slicing. In *Proc. Foundations of Software Engineering*. ACM, 11–20.

REPS, T. W., HORWITZ, S. B., SAGIV, M., AND ROSAY, G. 1994b. Speeding up slicing. In *SIGSOFT'94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, New York, 11–20. 19(5).

REPS, T. W. AND ROSAY, G. 1995. Precise interprocedural chopping. In *SIGSOFT'95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Washington, DC.

ROBSCHINK, T. 2005. Pfadbedingungen in Abhängigkeitsgraphen und ihre Anwendung in der Softwaresicherheitstechnik. Ph.D. thesis, Universität Passau. in German.

ROBSCHINK, T. AND SNELTING, G. 2002. Efficient path conditions in dependence graphs. In *Proceedings International ACM/IEEE Conference on Software Engineering (ICSE'02)*. Orlando, FL, 478–488.

RUSTAN, K., LEINO, M., AND NELSON, G. 1998. An extended static checker for Modula-3. In *Compiler Construction: 7th International Conference (CC'98)*. Lecture Notes in Computer Science, vol. 1383. Springer Verlag, 302–305.

SABELFELD, A. AND MYERS, A. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21,* 1 (January).

SMITH, G. AND VOLPANO, D. 1998. Secure information flow in a multi-threaded imperative language. In *Proceedings of the Twenty-Fifth ACM Symposium on Principles of Programming Languages*. San Diego, CA, 355–364.

SNELTING, G. 1996. Combining slicing and constraint solving for validation of measurement software. In *Proc. Static Analysis Symposium*. LNCS, vol. 1145. 332–348.

SREEDHAR, V. C., GAO, G. R., AND LEE, Y.-F. 1996. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 649–658.

STURM, T. AND WEISPFENNING, V. 1996. Computational geometry problems in REDLOG. In *Automated Deduction in Geometry*. 58–86.

TARJAN, R. E. 1974. Testing flow graph reducibility. *Journal of Computer and System Sciences 9*, 355–365.

TEITELBAUM, T. 2001. Code surfer user guide and reference. Tech. rep., Gramma Tech Product Documentation. http://www.grammatech.com/csurf-doc/manual.html.

TIP, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages 3,* 3 (Sept.), 121–189.

WEISER, M. 1984. Program slicing. *IEEE Trans. Softw. Eng. 10,* 4 (July), 352–357. republished in [Berzins 1995].

WEISPFENNING, V. 1997. Simulation and optimization by quantifier elimination. *Journal of Symbolic Computation 24,* 2, 189–208.

WEISPFENNING, V. 1999. Mixed real-integer linear quantifier elimination. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*. 129–136.

WOLFRAM, S. 1999. *The Mathematica Book*. Wolfram Research.

APPENDIX 1: Slicing and Goguen/Meseguer noninterference

In this appendix, we rely on the notation from Goguen and Meseguer [1984] and Mantel et al. [2000]. We assume a given, finite set of security domains $D = \{D_1, \ldots, D_n\}$ together with a given noninterference relation $\rightsquigarrow$. We use SDGs and path conditions to check whether the noninterference criterion for $(D, \rightsquigarrow)$ is fulfilled for the program.

Noninterference is defined with respect to an abstract automaton $\mathcal{A} = (Z, run, output, z_0)$ where $run : Z \times A \to Z$ is the state transition function and $output : Z \times A \to O$ is the output function. Z, the set of program states, is usually infinite; $z_0 \in Z$ is the start state. The set of actions A is in our case the set of program statements or expressions, or more precisely the set of nodes N in the SDG. run is extended to A^* as usual: $run(z, \epsilon) = z$; $run(z, a^\wedge x) = run(run(z, a), x)$.

The security domain of action $a \in A$ is $dom(a) \in D$. The noninterference relation $\not\rightsquigarrow \subseteq D \times D$ specifies which security domains must not influence each other. The complement of $\not\rightsquigarrow$, namely the interference relation $\rightsquigarrow$ is assumed to be reflexive and transitive. Given a statement sequence x and a security domain d, the function $purge : A^* \times D \to A^*$ removes from x all statements which must not influence security level d: $purge(x, d) = \langle a \in x \mid \neg(dom(a) \not\rightsquigarrow d)\rangle = \langle a \in x \mid dom(a) \rightsquigarrow d\rangle$.

A system is considered safe according to the Goguen/Meseguer noninterference criterion, if for all possible statement sequences x and all final statements a

$$output(run(z_0, x), a) = output(run(z_0, purge(x, dom(a))), a)$$

That is, the final program output is unchanged if any statement which must not influence the last action according to its security level is deleted. If the condition is not satisfied, there might be some action which produces a different output on an actual run than on a run with all supposedly noninfluential statements removed – that is, there is an influence from a statement s in x to a even though this is forbidden due to $dom(s) \not\rightsquigarrow dom(a)$. We see that the notion of safety is based on observational behavior and not on the source code.

The following theorem and corollary demonstrate how slices and path conditions can be used to check for noninterference.

Theorem. If

$$s \in BS(a) \implies dom(s) \rightsquigarrow dom(a)$$

then the noninterference criterion is satisfied for a.

Proof.[14] By definition of $purge$, we have

$$\begin{aligned} & output(run(z_0, purge(x, dom(a))), a) \\ = \; & output(run(z_0, \langle s \in x \mid dom(s) \rightsquigarrow dom(a)\rangle), a) \end{aligned}$$

For every $s \in \langle s \in x \mid dom(s) \rightsquigarrow dom(a)\rangle$, either $s \in BS(a)$ or $s \notin BS(a)$ holds. In the latter case, we may conclude $\neg I(s, a)$ as $I(s, a) \implies s \in BS(a)$ (see section 2.1), and s can be ignored as it cannot influence the final output. Thus we may assume $s \in BS(a)$, hence

$$\begin{aligned} = \; & output(run(z_0, \langle s \in x \mid dom(s) \rightsquigarrow dom(a)\rangle), a) \\ = \; & output(run(z_0, \langle s \in x \mid dom(s) \rightsquigarrow dom(a) \wedge s \in BS(a)\rangle), a) \end{aligned}$$

By assumption, $s \in BS(a) \implies dom(s) \rightsquigarrow dom(a)$, thus $dom(s) \rightsquigarrow dom(a)$ in the

[14] We thank one reviewer for providing this proof, which is simpler than our original inductive proof.

list comprehension is redundant, giving

$$\begin{aligned} & output(run(z_0, purge(x, dom(a))), a) \\ = \; & output(run(z_0, \langle s \in x \mid dom(s) \rightsquigarrow dom(a) \wedge s \in BS(a)\rangle), a) \\ = \; & output(run(z_0, \langle s \in x \mid s \in BS(a)\rangle), a) \\ = \; & output(run(z_0, x), a) \end{aligned}$$

as again those $s \in x$ where $s \notin BS(a)$ cannot influence the final output. QED.

This theorem can be exploited as follows. Typically we want to show that a certain subsystem (or set of statements) S cannot influence a. We thus demand that the noninterference criterion holds under the assumption $dom(s) \not\rightsquigarrow dom(a)$ for all $s \in S$. In order to guarantee this, first we have to check that $\forall s \in BS(a) : dom(s) \rightsquigarrow dom(a)$ (if this is not satisfied, there is a safety violation inside $BS(a)$). Then we check whether $S \cap BS(a) = \emptyset$. If both conditions are satisfied, noninterference holds.

In practice, slicing could be imprecise and $BS(a)$ could contain an s with $dom(s) \not\rightsquigarrow dom(a)$ even though s can never influence a. Besides using better slicing algorithms which avoid including s in $BS(a)$, we can exploit the fact that path conditions make slicing more precise and improve the criterion:

Corollary. If

$$(s \in BS(a) \wedge PC(s, a) \not\equiv false) \implies dom(s) \rightsquigarrow dom(a)$$

then noninterference holds for a.

APPENDIX 2: Proofs of decomposition equations

Proof of equation 10:

As any path from x to y must go through z,

$$\begin{aligned} PC(x,z) \wedge PC(z,y) &= \bigvee_{P \in CH(x,z)} \bigwedge_{u \in P} E(u) \; \wedge \bigvee_{P' \in CH(z,y)} \bigwedge_{u' \in P'} E(u') \\ &= \bigvee_{P \in CH(x,z)} \bigvee_{P' \in CH(z,y)} \Big(\bigwedge_{u \in P} E(u)\Big) \; \wedge \; \Big(\bigwedge_{u' \in P'} E(u')\Big) \\ &= \bigvee_{P \in CH(x,z)} \bigvee_{P' \in CH(z,y)} \bigwedge_{u \in PP'} E(u) \\ &= \bigvee_{P \in CH(x,y)} \bigwedge_{u \in P} E(u) = PC(x,y) \end{aligned}$$

Proof of equation 11: We write $CH(x,y)|z_i$ for the subchop of $CH(x,y)$ containing all paths containing z_i, and $PC(x,y)|z_i$ for the path condition in this subchop (that is, equation (1) is applied to $CH(x,y)|z_i$). Then

$$\begin{aligned} PC(x,y) &= \bigvee_{P \in CH(x,y)} \bigwedge_{u \in P} E(u) \; = \; \bigvee_{i=1}^{k} \bigvee_{P \in CH(x,y)|z_i} \bigwedge_{u \in P} E(u) \\ &= \bigvee_{i=1}^{k} PC(x,y)|z_i \; = \; \bigvee_{i=1}^{k} PC(x,z_i) \wedge PC(z_i,y) \end{aligned}$$

according to equation (10) since z_i is a dominator for y in $CH(x,y)|z_i$.

Proof of equation 12:

$$\begin{aligned} PC(x,y) &= \bigvee_{P\in CH(x,y)} \bigwedge_{u\in P} E(u) = \bigvee_{i=1}^{k} \bigvee_{P\in CH(x,y)|e_i} \bigwedge_{u\in P} E(u) \\ &= \bigvee_{i=1}^{k} PC(x,y)|e_i = \bigvee_{i=1}^{k} PC(x,e_i) \wedge PC(e_i,y) \\ &= \bigvee_{i=1}^{k} PC(x,e_i) \wedge \bigvee_{j=1}^{m} PC(e_i,y)|o_j \\ &= \bigvee_{i=1}^{k} \Big(PC(x,e_i) \wedge \big(\bigvee_{j=1}^{m} PC(e_i,o_j) \wedge PC(o_j,y)\big)\Big) \end{aligned}$$

as o_j is a dominator for y in $CH(e_i,y)|o_j$.

Proof of equation 13:

$$\begin{aligned} PC(x,y) &= \bigvee_{P\in CH(x,y)} \bigwedge_{u\in P} E(u) = \bigvee_{z\in pred(y)} \bigvee_{P\in CH(x,y)|z} \bigwedge_{u\in P} E(u) \\ &= \bigvee_{z\in pred(y)} PC(x,y)|z = \bigvee_{z\in pred(y)} PC(x,z) \wedge PC(z,y) \\ &= \bigvee_{z\in pred(y)} PC(x,z) \wedge E(z) \wedge E(y) = E(y) \wedge \bigvee_{z\in pred(y)} PC(x,z) \end{aligned}$$

as $E(z)$ is already conjunctively added in $PC(x,z)$. The proof of equation 14 is analogous.

Proof of equation 15: Let $x \to x_1 \to \ldots \to x_k \to x$ be the cycle. Then

$$\begin{aligned} PC(x,y) &= \bigvee_{P_\rho\in CH(x,y)} \bigwedge_{z\in P_\rho} E(z) = \Big(\bigvee_{\substack{P_\rho\in CH(x,y)\\ x_1\ldots x_k\notin P_\rho}} \bigwedge_{z\in P_\rho} E(z)\Big) \vee \Big(\bigvee_{\substack{P_\rho\in CH(x,y)\\ x_1\ldots x_k\in P_\rho}} \bigwedge_{z\in P_\rho} E(z)\Big) \\ &= \Big(\bigvee_{\substack{P_\rho\in CH(x,y)\\ x_1\ldots x_k\notin P_\rho}} \bigwedge_{z\in P_\rho} E(z)\Big) \vee \Big(\bigvee_{\substack{P_\rho\in CH(x,y)\\ x_1\ldots x_k\in P_\rho}} \bigwedge_{i=1}^{k} E(x_i) \wedge \bigwedge_{\substack{z\in P_\rho\\ z\notin\{x_1,\ldots,x_k\}}} E(z)\Big) \\ &= \bigvee_{P_\rho\in CH(x,y)} \Big(\Big(\bigwedge_{\substack{z\in P_\rho\\ z\notin\{x_1\ldots x_k\}}} E(z)\Big) \vee \Big(\bigwedge_{i=1}^{k} E(x_i) \wedge \bigwedge_{\substack{z\in P_\rho\\ z\notin\{x_1,\ldots,x_k\}}} E(z)\Big)\Big) \\ &= \bigvee_{P_\rho\in CH(x,y)} \Big(\bigwedge_{\substack{z\in P_\rho\\ z\notin\{x_1\ldots x_k\}}} E(z)\Big) = \bigvee_{\substack{P_\rho\in CH(x,y)\\ x_1\ldots x_k\notin P_\rho}} \bigwedge_{z\in P_\rho} E(z) = \bigvee_{\substack{P_\rho\in CH(x,y)\\ x_k\to x\notin P_\rho}} \bigwedge_{z\in P_\rho} E(z) \end{aligned}$$

as the cycle is excluded from $PC(x,y)$ iff its back edge is excluded.