On Temporal Path Conditions in Dependence Graphs

Andreas Lochbihler*
Lehrstuhl Softwaresysteme
Universität Passau
lochbihl@fim.uni-passau.de

Gregor Snelting
Lehrstuhl Softwaresysteme
Universität Passau
snelting@fim.uni-passau.de

Abstract

Program dependence graphs are a well-established device to represent possible information flow in a program. Path conditions in dependence graphs have been proposed to express more detailed circumstances of a particular flow; they provide precise necessary conditions for information flow along a path or chop in a dependence graph. Ordinary boolean path conditions however cannot express temporal properties, e.g. that for a specific flow it is necessary that some condition holds, and later *another specific condition holds.*

In this contribution, we introduce temporal path conditions, which extend ordinary path conditions by temporal operators in order to express temporal dependencies between conditions for a flow. We present motivating examples, generation and simplification rules, application of model checking to generate witnesses for a specific flow, and a case study. We prove the following soundness property: if a temporal path condition for a path is satisfiable, then the ordinary boolean path condition for the path is satisfiable. The converse does not hold, indicating that temporal path conditions are more precise.

1. Introduction

Program dependence graphs (PDGs) are a well-established device to represent possible information flow in a program. They are used for e.g. program slicing, debugging, reengineering, and security analysis; e.g., information flow control (IFC), a technique for discovering illegal flow from secret variables to public ports, can be based on PDGs, resulting in a more precise analysis than previous type-based approaches [23, 13]. PDGs today can handle medium-sized programs in full C or Java, but can only indicate if an information flow between two program points is possible or definitely impossible. *Path conditions* in PDGs, first proposed by Snelting [22], were to our knowledge the first means to express more detailed circumstances of a particular flow [23]; they provide necessary and precise conditions for information flow along a path in a dependence graph. Path conditions are boolean expressions over program variables, generated from conditions in if- or while-statements, as well as additional constraints extracted from a program. If a path condition cannot be satisfied, no information flow is possible along a path even though the PDG may indicate otherwise. If a path condition can be solved for the input variables (e.g. by using constraint solving techniques), the solved condition represents a *witness* for illegal information flow: if the specific witness values are fed to the program, the illegal flow becomes visible directly; this might be quite useful in law suits.

It is not easy to generate path conditions for medium-sized C or Java programs; for details see [23]. Nevertheless, path conditions have proven useful in realistic case studies. Path conditions as implemented today have, however, one property which may reduce precision: Boolean path conditions cannot express temporal properties, e.g. that for a specific flow it is necessary that a specific condition holds, and *later* another specific condition holds.

In this contribution, we introduce temporal path conditions, which extend ordinary path conditions by temporal operators in order to express temporal dependencies between conditions for a flow. We present motivating examples, generation and simplification rules and a case study. Applying model checking generates witnesses for a specific flow. We prove the following soundness property: if a temporal path condition for a path is satisfiable, then the ordinary boolean path condition for the path is satisfiable, too. The converse does not hold, indicating that temporal path conditions are more precise.

The essence of our work can be summarized as follows: Boolean path conditions can be quite imprecise in the presence of loop-carried dependencies, but temporal path conditions are not that more complicated to generate and simplify, and provide considerably more insight into the detailed conditions for a flow. In this contribution, we present

*This work was supported by DFG grant Sn11/9-1.

their theoretical foundations but we have not yet implemented them fully.

2. Path Conditions in Dependence Graphs

In this contribution, we focus on an imperative while programming language without procedures. In the intraprocedural case, the program dependence graph is simple and straightforward to generate. (For interprocedural PDGs or multithreaded programs, see e.g. [24].) Each program statement corresponds to a graph node, control dependences and data dependences form the edges. If statement t is control dependent on s, i. e. the mere execution of t depends on the evaluation of the conditional expression s (e.g. an if or while statement), there is an edge $s \mathrel{-\!\!\blacktriangleright} t$ labelled by the control condition $\mathrm{c}(s,t)$ for t being executed depending on s, e.g. the while or if condition. A data dependence edge $s \overset{x}{-\!\!\triangleright} t$ models variable x being assigned in s and used in t without being reassigned in between. $s \overset{x}{-\!\!\triangleright} t$ is loop-carried iff there is a while loop node u which execution can reach while x is passed on from s to t such that $u \mathrel{-\!\!\blacktriangleright}^* s$ and $u \mathrel{-\!\!\blacktriangleright}^* t$. We write $s \rightarrow t$ when not distinguishing control and data dependence.

A path $\pi : s \rightarrow^* t$ in the PDG means that information can possibly flow from s to t. Slicing exploits this property: By computing the *backward slice* $\mathrm{BS}\,(t) := \{\, s \,|\, s \rightarrow^* t \,\}$ for t, it conservatively approximates the set of statements that can influence t, i. e. s influencing t implies $s \in \mathrm{BS}\,(t)$. The *forward slice* $\mathrm{FS}\,(s) := \{\, t \,|\, s \rightarrow^* t \,\}$ is the set of all nodes that s can influence. The intersection of forward slice for s and backward slice for t is the *chop* $\mathrm{CH}\,(s,t) = \mathrm{FS}\,(s) \cap \mathrm{BS}\,(t)$ for s and t. For an example program and its PDG, see Fig. 1 where control dependences are drawn with dashed arrows, data dependences with solid ones. However, slicing can be pretty imprecise. Consider e.g. this program fragment:

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

The standard PDG indicates an influence $1 \longrightarrow 3$, but the value of x can only reach y if $i > 10$ and $i + 3 = 2 * j - 42$. Hence $(i > 10) \wedge (i + 3 = 2 * j - 42)$ is a necessary condition over program variables such that line 1 influences line 3. More generally:

```
1 i = j;          1 i_1 = j;
2 while (i<5)     2 while (i_2=Φ(i_1,i_3),
  {                              i_2<5) {
3   i = i+k;      3   i_3 = i_2+k;
4   if (i<=4)     4   if (i_3<=4)
5     x = a;      5     x = a;
6   else          6   else
7     y = x;      7     y = x;
8 }               8 }
9 z = y;          9 z = y;
```

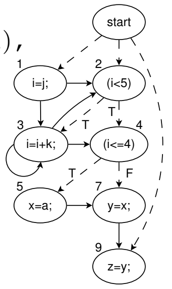

Figure 1. An example program (left), its SSA transform (center) and its PDG (right).

Definition 1 *In a program run r, statement s **influences** statement t (along the PDG path π) if r transports some information generated in s via control and data dependence edges (in the same relative order as in π) to t where it is used. A **path condition** $\mathrm{PC}(\pi)$ is a condition over program variables such that $\mathrm{PC}(\pi)$ is satisfiable if an influence can occur along π, i.e. there is a program run in which s influences t along π. A **path condition** $\mathrm{PC}(s,t)$ for s and t is a condition over program variables that is satisfiable if s influences t in some run r.*

Originally, path conditions are boolean formulae whose variables are implicitly quantified existentially. By definition, an influence between s and t can occur only along a PDG path $\pi : s \rightarrow^* t$ between the two statements s and t. Thus, if we can compute $\mathrm{PC}(\pi)$ for every path π between s and t, we also obtain a path condition for s and t by taking the disjunction of these conditions for all paths between s and t. The core idea for path conditions for a PDG path π is that all nodes on π must be executed if the influence along π occurs.

Definition 2 *An **execution condition** for a PDG node v is a necessary condition over program variables for v being executed.*

For example, we can use control dependence to build execution conditions (and thus a preliminary version for path conditions – $\mathrm{PC_B}$ meaning boolean path condition):

$$\mathrm{E}(v) := \bigvee_{\rho:\mathrm{start} \mathrel{-\!\!\blacktriangleright}^* v} \; \bigwedge_{u \mathrel{-\!\!\blacktriangleright} u' \in \rho} \mathrm{c}(u,u'); \quad \mathrm{PC_B}(\pi) := \bigwedge_{v \text{ node in } \pi} \mathrm{E}(v)$$

For example, the execution condition for line 5 in Fig. 1 is $\mathrm{E}(5) = (i < 5) \wedge (i \leq 4) = (i < 5)$.

Note that an execution condition $\mathrm{E}(v)$ is always finite because cycles in the control dependence graph (CDG) do not contribute to $\mathrm{E}(v)$ due to the absorption law for $\vee$ [22].

However, since there are usually multiple assignments to a variable within a program and thus a variable's value may change during program execution, we must transform a program into static single assignment (SSA) form [7] first: In SSA form, every variable occurs at most once on the left hand side of an assignment. If necessary, we use extra indices to distinguish between different SSA variants of a program variable x: Where control flow meets, we introduce a Φ function that selects the appropriate source for uses of x. For example, reconsider Fig. 1: The program in the center shows the SSA transform of program on the left. Now, the execution condition for line 7 is $\mathrm{E}(7) = (i_2 < 5) \wedge \neg(i_3 \leq 4)$. Without the SSA transformation, $\mathrm{E}(7)$ would be $(i < 5) \wedge \neg(i \leq 4)$, which is not satisfiable and not a necessary condition.

For path conditions, Φ functions are translated into Φ conditions, which are added conjunctively. In Fig. 1, e.g., $\mathtt{i}_2$=$\Phi(\mathtt{i}_1, \mathtt{i}_3)$ becomes $\Phi(i_2; i_1, i_3) := (i_2 = i_1) \vee (i_2 = i_3)$. Besides we also have Φ conditions for data dependence edges $s \overset{x}{-\!\!\triangleright} t$ on a path: Let i be x's SSA index in s and j the x's one in t. Then, the value of x_i must equal x_j's value for the influence to occur. Thus, we obtain the extra constraint $\Phi(s \overset{x}{-\!\!\triangleright} t) = (x_i = x_j)$, which we also add conjunctively.

While SSA form ensures that execution conditions are always correct, it does unfortunately not guarantee an SSA variable being constant during execution because an assignment statement may be executed multiple times, e.g. if loops are present. When we look at the PDG path $\pi := 5 \overset{x}{-\!\!\triangleright} 7$ in Fig. 1, we obtain the path condition $\mathrm{E}(5) \wedge \mathrm{E}(7) = (i_2 < 5) \wedge (i_3 \leq 4) \wedge \neg(i_3 \leq 4)$, which is not satisfiable although the value of a can reach y via x. As the data dependence edge $5 \overset{x}{-\!\!\triangleright} 7$ is loop-carried, $\mathrm{E}(7)$ must hold only one loop iteration later than $\mathrm{E}(5)$, i.e. we actually use the same variable identifier i_3 to refer to two different runtime assignments to i_3. Hence, to obtain correct path conditions, we must distinguish the variables before and after loop-carried data dependences in a path condition (for details, see [17]):

$$\mathrm{PC_B}(\pi) = (i_2 < 5) \wedge (i_3 \leq 4) \wedge (i'_2 < 5) \wedge \neg(i'_3 \leq 4). \quad (1)$$

When we compute $\mathrm{PC_B}(s, t)$, simply taking the disjunction over $\mathrm{PC_B}(\pi)$ for all paths $\pi : s \rightarrow^* t$ might result in an infinite formula, e. g. if the PDG contains a cycle between s and t. Fortunately, we can eliminate cycles: If a cycle does not contain a loop-carried data dependence, we simply ignore the cycle [22]. Otherwise, we do not generate any conditions for the nodes on the cycle and rename the variables in the condition that belong to nodes before the cycle. This way, nodes before and after the cycle are not related. For more details on how to construct boolean path conditions, see [22, 20, 23].

Obviously, boolean conditions as presented above are not the only ones one can imagine as path condition for a specific PDG path π. In order to compare two boolean path conditions, we introduce the concept of *strength*. Since we are interested in constructing path conditions that are as precise as possible, i. e. we want to limit the false witnesses to as few cases as possible, given two boolean path conditions, say $\mathrm{PC_{B1}}$ and $\mathrm{PC_{B2}}$, for the same PDG path π, we say $\mathrm{PC_{B1}}$ is *stronger* than $\mathrm{PC_{B2}}$ iff $\mathrm{PC_{B1}} \Rightarrow \mathrm{PC_{B2}}$ [23]. For example, boolean path conditions with Φ conditions are, in general, stronger than those without.

3. Temporal Path Conditions

Although boolean path conditions are quite strong in practice, they are not able to properly take care of loop-carried dependences. Even worse, they discard the order of the nodes on the path because the $\wedge$ operator is commutative. In particular, we can not express that some condition must hold only *after* some other condition is fulfilled, e.g. data dependences on a PDG path induce a temporal execution order on the nodes on the path. To address these issues, we propose temporal path conditions based on Linear Temporal Logic (LTL). LTL formulae connect boolean expressions over program variables by standard boolean operators and four temporal operators: *always* $\square$, *eventually* $\lozenge$, *next* $\bigcirc$, and *until* $\mathcal{U}$. (For notation, we assume that boolean operators bind stronger than temporal ones.) They are evaluated over infinite sequences of states, i. e. assignments to the variables. For details, see [5].

In this contribution, we restrict ourselves to imperative programs in a While language with scalar-only variables, i. e. we have boolean and integer variables, assignments, if and while statements, but no gotos, no aliasing, no runtime exceptions and no side effects inside expressions.

Whereas we use the PDG to generate LTL path formulae, we obtain the state sequences which can satisfy the LTL formula from program traces, i. e. paths in the control flow graph (CFG) with variables assigned to their values. However, since control statements do not alter the program state, we project these traces to assignment-only statements. As with boolean path conditions, we need to transform the program into SSA form. In case the program trace terminates, we repeat the last state infinitely often to make the sequence artificially infinite. For example, if we run the program in Fig. 1 with initial values $a = 0$, $j = 2$, and $k = 3$, we obtain the state sequence shown in Table 1.

As a motivational example, reconsider the program in Fig. 1, but now, assume that line 4 is replaced by `if`

Table 1. State sequence for the program (cf. Fig. 1) run with initial values $a = 5$, $j = 2$, $k = 3$.

Time	Line	a	i_1	i_2	i_3	j	k	x	y	z
0	`begin`	0	$\perp$	$\perp$	$\perp$	2	3	$\perp$	$\perp$	$\perp$
1	1	0	2	$\perp$	$\perp$	2	3	$\perp$	$\perp$	$\perp$
2	3	0	2	2	5	2	3	$\perp$	$\perp$	$\perp$
3	5	0	2	2	5	2	3	0	$\perp$	$\perp$
4	9	0	2	5	5	2	3	0	$\perp$	$\perp$
5	`end`	0	2	5	5	2	3	0	$\perp$	$\perp$
6	`end`	0	2	5	5	2	3	0	$\perp$	$\perp$
⋮	⋮	⋮								⋮

`!(i<=4)`, i.e. the `if`'s predicate is negated. The boolean path condition for the path $\pi = 5 \overset{x}{-\!\!\triangleright} 7$ (the only path along which 5 influcences 7) is now

$$\mathrm{PC_B}(\pi) = (i_2 < 5) \wedge \neg(i_3 \leq 4) \wedge (i'_2 < 5) \wedge (i'_3 \leq 4), \quad (2)$$

which is equivalent to (1). Clearly, for this influence to occur line 5 must be executed *before* line 7 and in between, execution must not leave the while loop in line 2. This can be expressed by the formula

$$(i_2 < 5) \wedge \neg(i_3 \leq 4) \wedge (i_2 < 5)\, \mathcal{U}\, ((i_2 < 5) \wedge (i_3 \leq 4)). \quad (3)$$

In particular, we can deduce from (3) that i_3 must be increased between line 5 and line 7 being executed. We can use path-insensitive data flow analysis to obtain that $i_3 = i_2 + k$ always holds after having visited node 5, i.e. we can substitute $i_2 + k$ for i_3 in (3). Hence, a constraint solver can deduce that $k < 0$ must hold. Note that we can obtain the same result from (2) with the same techniques. However, in combination with i_3 being necessarily increased, we see that (3) is not satisfiable. Obviously, we can not prove (2) unsatisfiable because (2) $\Leftrightarrow$ (1) and, in fact, the influence in the original example can actually occur. (The LTL path condition for the original example is satisfiable.)

Of course, one can come up with many different LTL path conditions for a specific PDG path. Like in the boolean case, we say θ_1 is *stronger* than θ_2 iff $\theta_1 \Rightarrow \theta_2$. In what follows, we present the construction of LTL path conditions that are reasonably strong.

3.1. Building Blocks for LTL Path Conditions

Clearly, execution conditions for PDG nodes as presented in Sec. 2 are a core concept of LTL path conditions, too. Now, we can, however, also utilize execution conditions for data dependence edges:

Definition 3 *Let $s \overset{x}{-\!\!\triangleright} t$ be a data dependence edge and C the set of assignment nodes which are on CFG paths $\rho : s \rightarrowtail^* t$ from s to t such that x is not redefined on ρ. An* ***execution condition*** *for $s \overset{x}{-\!\!\triangleright} t$ is a necessary condition over program variables for any node in C being executed.*

Obviously, $\mathrm{E}(s \overset{x}{-\!\!\triangleright} t) := \bigvee_{v \in C} \mathrm{E}(v)$ is a correct execution condition for $s \overset{x}{-\!\!\triangleright} t$. For example, $\mathrm{E}(5 \overset{x}{-\!\!\triangleright} 7)$ in Fig. 1 is $(i_2 < 5)$. Since the models of interest for our LTL formulae contain only states for assignment nodes, we have restricted C to assignment nodes.

Yet, execution conditions for statements can be strengthened by loop termination conditions: Suppose a loop predicate node u dominates node v in the CFG, i.e. all CFG paths from the entry node to v contain u, and v is not control dependent on u. When execution reaches v, u must have been executed and the loop must have terminated. Hence, the negated predicate of u holds at v. The *loop termination condition* $\mathrm{L}(v)$ for v is the conjunction thereof over all such u. From now on, we assume that $\mathrm{E}(v)$ contains $\mathrm{L}(v)$ (added conjunctively). For example, in Fig. 1, $\mathrm{E}(9)$ is now $\neg(i_2 < 5)$. Equally, when a data dependence $e = v \overset{x}{-\!\!\triangleright} w$ leaves the loop predicate node u, i.e. u controls v but not w, $\neg u$ must hold at w. The *loop termination condition* $\mathrm{L}(e)$ for e is the conjunction of all such u. In Fig. 1, we have $\mathrm{L}(7 \overset{y}{-\!\!\triangleright} 9) = \neg(i_2 < 5)$.

Apart from execution and loop termination conditions, we include Φ conditions in LTL path conditions. As before, for a data dependence $s \overset{x}{-\!\!\triangleright} t$, we use the constraint $\Phi(s \overset{x}{-\!\!\triangleright} t)$. However, we can no longer capitalize on including Φ constraints from Φ functions because, by definition of Φ functions, every program trace, i.e. every possible model of interest for the LTL path condition, trivially satisfies all corresponding Φ constraints.

3.2. LTL Path Conditions for a Single Path

Given a PDG path $\pi : s \rightarrow^* t$, we want to generate an LTL formula θ such that if there is a run r that carries an influence along π, then the state sequence for r satisfies $\Diamond\, \theta$. In this case, we say that θ is a *correct* path condition for π. Having presented the building blocks for LTL path conditions in Sec. 3.1, we now present how to combine them to obtain a correct path condition. Fig. 2 shows the algorithm for computing the LTL path condition, denoted by $\mathrm{PC_L}(\pi)$, for the path π. It is motivated by the observation that given program run $r = (r_i)_i$ where in $(r_i)_{j \leq i \leq k}$ statement u influences w along π (and π has at least two nodes), then there is an $l \in \{j, \ldots, k\}$ such that in $(r_i)_{j \leq i \leq l}$ u influences some v along the first edge of π and in $(r_i)_{l \leq i \leq k}$ v influences w along the rest of π. Since we do not have states for control nodes, if π starts with a control dependence edge $u \,-\!\!\blacktriangleright v$, we have $l = j$, i.e. there is no need for an $\mathcal{U}$ operator because the execution condition for u must obviously hold in v, too. Thus, there is an $\mathcal{U}$ operator for every data dependence edge in π, which we can often (cf. Sec. 3.4) omit.

For example, in Fig. 1, consider $\pi = e, f, 4 \,-\!\!\blacktriangleright 7$ where $e := 1 \overset{i}{-\!\!\triangleright} 3$ and $f := 3 \overset{i}{-\!\!\triangleright} 4$. The LTL path condition for π is

$$\begin{aligned}
&\mathrm{E}(1) \wedge \mathrm{E}(e)\, \mathcal{U}\, (\mathrm{L}(e) \wedge \Phi\,(e) \wedge \mathrm{E}(3) \wedge \\
&\quad \mathrm{E}(f)\, \mathcal{U}\, (\mathrm{L}(f) \wedge \Phi\,(f) \wedge \mathrm{E}(4) \wedge \mathrm{E}(7))) = \\
&\mathrm{true} \wedge \mathrm{true}\, \mathcal{U}\, (\mathrm{true} \wedge (i_1 = i_2) \wedge (i_2 < 5) \wedge \\
&\quad (i_2 < 5)\, \mathcal{U}\, (\mathrm{true} \wedge (i_3 = i_3) \wedge (i_2 < 5) \wedge \\
&\qquad ((i_2 < 5) \wedge \neg(i_3 \leq 4))))
\end{aligned} \quad (4)$$

which simplifies to (cf. Sec. 3.4)

$$\Diamond((i_1 = i_2) \wedge (i_2 < 5)\, \mathcal{U}\, ((i_2 < 5) \wedge (i_3 > 4))). \quad (5)$$

```
**Input:** $\pi : s \rightarrow^* t$ path in the PDG $G$
**Output:** $\mathrm{PC_L}(\pi)$

PCPath($\pi$)
  if ($\pi$ has only one node $s$) return $\mathrm{E}(s)$
  let $e$,$\pi'$ such that $e, \pi'$=$\pi$
  if ($e$ is a control dependence)
    return ($\mathrm{E}(s) \wedge$ PCPath($\pi'$))
  else
    return ($\mathrm{E}(s) \wedge \mathrm{E}(e)\, \mathcal{U}\, (\mathrm{L}(e) \wedge \Phi(e) \wedge$PCPath($\pi'$)))
```

Figure 2. Algorithm PCPath for LTL path conditions for a single PDG path π.

3.3. Path Conditions for Chops

Like with boolean path conditions, we generate LTL path conditions not only for a single path, but also for two statements s and t. Again, taking the disjunction of $\mathrm{PC_L}(\pi)$ over all paths $\pi : s \rightarrow^* t$ may result in infinite formulae if the PDG contains cycles. Thus, we adapt the algorithm (cf. Fig. 3). Suppose $\pi : s \rightarrow^* t$ is a cycle-free PDG path. Let $P(\pi)$ denote the set of all paths $\rho : s \rightarrow^* t$ such that we obtain π when we remove all cycles from ρ. *PCPath'* computes a path condition $\overline{PC}_{\mathrm{L}}(\pi)$ for π which all runs for all paths in $P(\pi)$ satisfy. Whenever a cycle may be inserted into π, we include an extra $\mathcal{U}$ operator to account for this cycle in the run. There are no states for control dependences in our model, so we can ignore control dependences. θ, the first operand of the new $\mathcal{U}$, is an execution condition which holds for all nodes on all possible cycles which can be inserted into π after the first edge. Note that if there is no such cycle with a data dependence in all recursive steps, the formula generated is equivalent to one without the extra $\mathcal{U}$ operators. *PCChop* computes the disjunction over all cycle-free paths in the PDG G between s and t.

Let us look again at $\pi = 1 \overset{i}{\relbar\!\rhd} 3, 3 \overset{i}{\relbar\!\rhd} 4, 4 \relbar\!\blacktriangleright 7$ in Fig. 1. Only can we add cycles to π after the first edge, namely $3 \overset{i}{\relbar\!\rhd} 3$ and $3 \overset{i}{\relbar\!\rhd} 2, 2 \relbar\!\blacktriangleright 3$. Hence, we include $\mathrm{E}(3) \wedge (\mathrm{E}(3 \overset{i}{\relbar\!\rhd} 3) \vee \mathrm{E}(3 \overset{i}{\relbar\!\rhd} 2))\, \mathcal{U}\, (\mathrm{L}(1 \overset{i}{\relbar\!\rhd} 3)$ in (4):

$$\begin{gathered}\mathrm{E}(1) \wedge \mathrm{E}(e)\, \mathcal{U}\, (\mathrm{L}(e) \wedge \Phi(e) \wedge \\ \mathbf{E(3) \wedge (E(3 \overset{\boldsymbol{i}}{\relbar\!\rhd} 3) \vee E(3 \overset{\boldsymbol{i}}{\relbar\!\rhd} 2))\, \boldsymbol{\mathcal{U}}\, (L(\boldsymbol{e}) \wedge} \\ \mathrm{E}(3) \wedge \mathrm{E}(f)\, \mathcal{U}\, (\Phi(f) \wedge \mathrm{E}(4) \wedge \mathrm{E}(7))))\end{gathered} \tag{6}$$

When we simplify this formula, we obtain again (5).

Note that simplification has removed the extra $\mathcal{U}$ operator in (6) again. This need not always be the case: Consider Fig. 4 and the path $\rho = 5 \overset{a}{\relbar\!\rhd} 6, 6 \overset{y}{\relbar\!\rhd} 7, 7 \overset{x}{\relbar\!\rhd} 6, 6 \overset{y}{\relbar\!\rhd} 9$. Let π be the path we obtain by removing the cycle from ρ. $\mathrm{PC_L}(\pi)$ (without Φ constraints) simplifies to

$$d \wedge (b \wedge c)\, \mathcal{U}\, (b \wedge c \wedge \neg d).$$

```
**Input:** PDG $G = (V, E)$; nodes $s, t$
**Output:** $\overline{PC}_{\mathrm{L}}(s, t)$

PCPath'($G$, $\pi$)
  if ($\pi$ has only one node $s$) return $\mathrm{E}(s)$
  let $e$,$\pi'$ such that $e, \pi'$=$\pi$
  let $v$ be the target node of $e$
  let $A$ be the set of data dependences
    on any cycle through $v$ in $G$
  if ($A \neq \emptyset$) set $\theta$:=$\bigvee_{a \in A} \mathrm{E}(a)$
  else set $\theta$:=$false$
  if ($e = s \relbar\!\blacktriangleright v$)
    return ($\mathrm{E}(s) \wedge \theta\, \mathcal{U}$ PCPath'($G$,$\pi'$))
  else
    return ($\mathrm{E}(s) \wedge \mathrm{E}(e)\, \mathcal{U}\, (\mathrm{L}(e) \wedge \Phi(e) \wedge \mathrm{E}(v) \wedge$
          $\theta\, \mathcal{U}\, (\mathrm{L}(e) \wedge$ PCPath'($G$, $\pi'$))))

PCChop($G$, $s$, $t$)
  let $I$,$(\pi_i)_{i \in I}$ such that $(\pi_i)_{i \in I}$ enumerates all
    cycle-free paths $s \rightarrow^* t$ in $G$
  set $\rho$:=$false$
  for each $i$ in $I$ set $\rho$:=$\rho \vee$ PCPath'($G$, $\pi_i$)
  return $\rho$
```

Figure 3. Algorithm PCChop for LTL path conditions for a chop between s and t in the PDG G.

Note that $\mathrm{PC_L}(\pi)$ is not a correct path condition for ρ because, in fact, line 5 can influence line 9 via lines 6 and 7 some iterations later. The problem is $(b \wedge c)$ being not an execution condition for $e = 7 \overset{x}{\relbar\!\rhd} 6$ because execution leaves the if branch while e is being passed. However, $\overline{PC}_{\mathrm{L}}(\pi)$ is correct and reads after simplification (without Φ constraints) $c \wedge d \wedge b\, \mathcal{U}\, (b \wedge c \wedge \neg d)$. In this case, simplifying the LTL formula with the extra $\mathcal{U}$ operator leads to a formula where the first operand of the $\mathcal{U}$ operator has been weakened.

Lemma 1 *Let $\rho : s \rightarrow^* t$ be a PDG path and $\pi : s \rightarrow^* t$ be ρ with all cycles removed. Then,* $\mathrm{PC_L}(\rho) \Rightarrow \overline{PC}_{\mathrm{L}}(\pi)$.

A proof can be found in [18]. Thus, we are also able to compute correct LTL path conditions for two statements:

Corollary 1 *Let Π be the set of all cycle-free paths π : $s \rightarrow^* t$. Then,* $\mathrm{PC_L}(s, t) := \bigvee_{\pi \in \Pi} \overline{PC}_{\mathrm{L}}(\pi)$ *is a correct path condition for s and t.*

Proof. This follows directly from the correctness of $\mathrm{PC_L}(\pi)$, Lem. 1, and the idempotence of $\vee$.

```
 1 while (b) {
 2   c = !c;
 3   if (c) {
 4     if (d)
 5       a = 5;
 6     y = a+x;
 7     x = y;
 8     if (!d)
 9       z = y;
10   }
11   d = false;
12   y = 0;
13 }
```

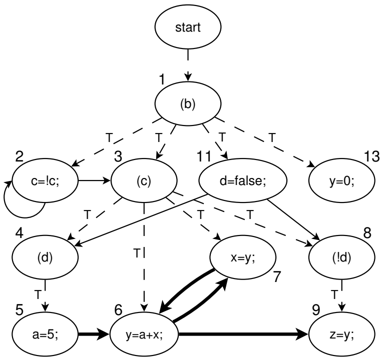

Figure 4. An example program and its PDG to show the extra $\mathcal{U}$ operator for cycles being necessary.

3.4. Simplifying LTL Path Conditions

Path conditions tend to become very long very quickly as programs increase in size. Thus, we must simplify them before examining them further. First, there are numerous equivalences for LTL formulae, i.e., in many cases, we can greatly simplify a formula by applying LTL identities (as we have done so far in examples). Below, we list some that have proven useful in simplifying LTL path conditions. Let A, B, C, D, E, and F be LTL formulae such that $B \Rightarrow D$, $C \Rightarrow D$, and $E \Rightarrow F$. Then, the following holds:

$$\Diamond(A\,\mathcal{U}\,B) \Leftrightarrow A\,\mathcal{U}\,\Diamond B \Leftrightarrow \Diamond B \Leftrightarrow \text{true}\,\mathcal{U}\,B \tag{7}$$

$$B\,\mathcal{U}\,(D\,\mathcal{U}\,A) \Leftrightarrow D\,\mathcal{U}\,A \Leftrightarrow D\,\mathcal{U}\,(B\,\mathcal{U}\,A) \tag{8}$$

$$B \wedge E\,\mathcal{U}\,(D \wedge F\,\mathcal{U}\,A) \Leftrightarrow B \wedge F\,\mathcal{U}\,A \tag{9}$$

$$A\,\mathcal{U}\,B \vee A\,\mathcal{U}\,C \Leftrightarrow A\,\mathcal{U}\,(B \vee C) \tag{10}$$

$$D \wedge B\,\mathcal{U}\,C \Leftrightarrow B\,\mathcal{U}\,C \tag{11}$$

$$\text{false}\,\mathcal{U}\,A \Leftrightarrow A \tag{12}$$

Also, we can often simplify LTL path conditions by slightly weakening them using implications. Since they are only necessary conditions for an influence, this does not affect their correctness. For example, $A\,\mathcal{U} C \vee B\ \mathcal{U}\ C$ implies $(A \vee B)\ \mathcal{U}\ C$, A implies $B\ \mathcal{U}\ A$, and $A\ \mathcal{U}\ (B\ \mathcal{U}\ C)$ implies $(A \vee B)\ \mathcal{U}\ C$. Besides, $\mathcal{U}$ is monotone in both operands w.r.t. to implication. Simplification can be done completely automatically: By definition, the first operand of an $\mathcal{U}$ operator in an LTL path condition is always a boolean formula and at most one operand of any maximal conjunction is not boolean. Hence, standard constraint solvers to decide the implications that must hold for (9), (10), and (11) are sufficient for our simplification purposes. In fact, a SAT solver can decide most implications, for frequently a program predicate occurs multiple times in a formula, i.e. we can treat boolean program expressions like propositional variables. However, when we use these simplification rules as rewrite rules, the order of their application is important. Thus, we have to backtrack when rewriting gets stuck to see if there is a different way to get to an even simpler formula.

Although these rules are a powerful means to make path conditions understandable, instead of greatly simplifying a formula we ought to not create unnecessary formula parts in the first place: By construction, an LTL path condition is a formula of nested $\mathcal{U}$ operators. The argument we used to prove that an SSA transformation is not sufficient for boolean path conditions shows us that we can not completely avoid $\mathcal{U}$ operators. Besides, loop termination conditions require them, too. For example, the $\mathcal{U}$ operator along $7 \xrightarrow{y}\!\triangleright 9$ in Fig. 1 separates the loop predicate $(i_2 < 5)$ from its termination condition $\neg(i_2 < 5)$. If we were to remove this operator, we inevitably would have to drop some of the constraints to preserve correctness. Sometimes, however, we may drop the $\mathcal{U}$ operator introduced by a data dependence:

Lemma 2 *Let $\pi : s \rightarrow^* t$ be a PDG path and $u \xrightarrow{x}\!\triangleright v$ a data dependence in π that is not loop-carried and does not leave a loop. Then, by conjunctively adding all operands of the maximal conjunction that contains the $\mathcal{U}$ operator for $u \xrightarrow{x}\!\triangleright v$ in $\overline{PC}_{\mathrm{L}}(\pi)$, except for the $\mathcal{U}$ term, loop termination conditions, and Φ conditions for loop-carried data dependences, to the $\mathcal{U}$'s second operand, we obtain a correct path condition.*

A slightly stronger lemma is shown in [18]. The key idea is that all SSA variables that occur in the maximal conjunction are defined in nodes which cannot be executed between u and v. For example (Fig. 1), let π be as in (4) and (5). Note that both $3 \xrightarrow{i}\!\triangleright 4$ and $1 \xrightarrow{i}\!\triangleright 3$ are not loop-carried and $3 \xrightarrow{i}\!\triangleright 4$ does not leave a loop. By Lem. 2, we can insert what is set in bold face:

$$\Diamond((i_1 = i_2) \wedge (i_2 < 5)\,\mathcal{U}\,(\boldsymbol{(i_1 = i_2)} \wedge \boldsymbol{(i_2 < 5)} \wedge (i_2 < 5) \wedge ((i_2 < 5) \wedge \neg(i_3 \leq 4))))$$

Now we drop the first occurrence of $(i_1 = i_2)$ and get after simplification $\Diamond((i_1 = i_2) \wedge (i_2 < 5) \wedge \neg(i_3 > 4))$, which contains one temporal operator less than (5).

Lem. 2 seems to make formulae grow, but this is only due to its formulation. In fact, we employ it to move constraints (by dropping the original terms) into the scope of an $\mathcal{U}$ operator nested more deeply and then apply (8) and (7) to properly remove the $\mathcal{U}$ operator.

Regarding path conditions for chops, we may sometimes consider fewer paths by exploiting the disjunctive absorption law: If A implies B then $A \vee B = B$.

Lemma 3 *Let π, ρ be PDG paths. If $\Diamond\overline{PC}_{\mathrm{L}}(\rho)$ is equivalent to $\Diamond\overline{PC}_{\mathrm{L}}(\sigma)$ for some suffix σ of π, then $\Diamond\overline{PC}_{\mathrm{L}}(\pi) \Rightarrow \Diamond\overline{PC}_{\mathrm{L}}(\rho)$. If ρ is a prefix of π, then $\overline{PC}_{\mathrm{L}}(\pi) \Rightarrow \overline{PC}_{\mathrm{L}}(\rho)$.*

Proof. The first claim directly follows by the monotonicity of $\Diamond$ w.r.t. implication, the idempotence law for $\Diamond$ and the implication $A \wedge B \; \mathcal{U} \; C \Rightarrow \Diamond C$. By construction of LTL path conditions, the second claim follows from the monotonicity of $\mathcal{U}$ w.r.t. implication.

Once simplification is done, we look for conjunctions of boolean conditions and feed them to a constraint solver to see whether they are satisfiable at all. If not, we immediately know that there is no program execution for any path that generates this type of constraint. In case we compute the LTL path condition for a chop, we drop this path from the outermost disjunction (cf. Cor. 1), i.e. this path is not enumerated by $(\pi_i)_{i \in I}$ in Fig. 3. To increase chances of showing unsatisfiability we can include extra constraints which can be generated from other data flow analysis techniques. Since there is a single definition for every SSA variable, we can backsubstitute these definitions in the formula until we reach a Φ function definition provided we are not passing along loop-carried data dependences and no definition of the variables introduced by the substitution can possibly be executed along the data dependence edges for the substitution. Unfortunately, this approach interferes with the simplification offered by Lem. 2. Although we have not yet implemented this additional approach, we have already seen it being useful in the motivating example from Sec. 3 (cf. Fig. 1): In (3), we substitute i_3 by $i_2 + k$, so we deduce that $k < 0$. In combination with $\Phi(i_1, i_3) = i_3$ given the data dependence $5 \xrightarrow{x} 7$, we see that i_3 is decreasing whereas (3) requires i_3 being increased, a contradiction. Note that although we can analyze temporal path conditions in this way by combining a number of other static analyses, our focus lies on model checking them (cf. Sec. 5).

3.5. Comparing Boolean and LTL Path Conditions

In the motivating example of Sec. 3, we have seen that temporal path conditions are more precise than boolean path conditions. The next theorem shows that we can derive a satisfying assignment for the boolean path condition from a satisfying program trace of the corresponding LTL path condition:

Theorem 1 (Soundness of LTL Path Conditions) *Let $\pi : s \rightarrow^* t$ be a cycle-free PDG path and let θ denote the path condition for π in which all unnecessary $\mathcal{U}$ operators for data dependences have been dropped (cf. Lem. 2) and $\mathcal{U}$ operators for all cycles that can be inserted into π have been included (cf. Sec. 3.3). Let η denote the boolean path condition for π, in which variables have been separated as necessary. Then, a satisfying assignment for η can be constructed from a satisfying program trace for θ.*

A proof can be found in [18]. Theorem 1 says that if a witness for an influence along a PDG path π is found for the LTL formula then we can obtain a satisfying assignment for the boolean path condition from the witness. Moreover, in the proof, it can be seen that we only have to look at those states that are involved in satisfying execution conditions and Φ constraints, i.e. that correspond to nodes on π.

Apart from temporal operators, loop termination conditions, which we can not easily include in boolean path conditions, make LTL path conditions more precise. For example, consider the following program skeleton where we have a data dependence e w.r.t. x that leaves a loop:

```
if (...)    while (b)   ...   x = ...
if (b)  ...  x ...
```

In this case, the loop termination condition for e gives the constraint $\neg b$, but the execution condition for e's target statement is b, a contradiction, i.e. no information can flow along e.

3.6. LTL vs. CTL

LTL is a natural choice for temporal path conditions when we look at a single path, but for chops, other logics such as CTL may come to mind. We restrict our comparison to LTL and CTL here because these are the most popular temporal logics for which high-performance model checkers are available. When we consider LTL to be a part of CTL* [5], we actually have for a path condition θ that no influence is possible if the program model does *not* satisfy $\mathbf{A} \neg \Diamond \theta$. In LTL, every part of the formula refers to the same program trace whereas in CTL, subformulae under the scope of different path quantifiers may be fulfilled in different program traces. However, due to the specific structure of path conditions, we can prefix every $\mathcal{U}$ subformulae with the existential path quantifier $\mathbf{E}$ to obtain a correct CTL path condition which is equally strong. Yet, path quantifiers impede simplification; e.g., if we want to simplify the path condition θ for a chop more aggressively, we can factor out a common suffix of the influence path: We drop its path condition η in θ and add it conjunctively again: $\theta \wedge \Diamond \eta$. With CTL, this trick is impossible since we can not ensure that $\Diamond \eta$ holds in the same program trace as θ does.

4. Case Study

In this section, we present a short case study in which we have applied LTL path conditions to discover a way of manipulating a weighing scale. [18] contains a more detailed description. Fig. 5 shows a fragment from the measurement software of a fictitious cheese weighing scale. For simplicity, there is just one input port from which both the weight sensor data and the keystrokes are read. In normal mode,

```
 1 sk0 = 65; sk1 = 43;
 2 sk2 = 45; sk3 = 13;
 3 mode=0;
 4 p_keyb=input; p_weigh=input;
 5 while (true) {
 6   if (p_keyb = 27)
 7     mode = 0;
 8   else {
 9     if ((mode = 3) && (p_keyb = sk3)) {
10       mode = 4;
11       p_keyb = input;
12     }
13     if ((mode = 2) && (p_keyb = sk2))
14       mode = 3;
15     if ((mode = 1) && (p_keyb = sk1))
16       mode = 2;
17     if ((mode = 0) && (p_keyb = sk0))
18       mode = 1;
19   }
20   if (mode = 4) {
21     if (p_keyb = 43)
22       kal_kg = kal_kg+1;
23     if (p_keyb = 45)
24       kal_kg = kal_kg-1;
25     p_keyb = input;
26   }
27   if (p_weigh > 0)
28     u_kg = p_weigh*kal_kg;
29   p_keyb = input;
30   p_weigh =input;
31 }
```

Figure 5. A cheese scale measurement software with service mode.

the measurement software computes the weight in kg from the weight sensor data and the calibration factor kal_kg (lines 28-29). In service mode, which is activated by entering a specific sequence of key strokes (lines 9–18), the calibration factor can be adjusted by the keyboard (lines 21-24). We want to check if we can manipulate the weight value u_kg – shown on the display – by the keyboard. The path condition (without SSA indices and Φ constraints, but with the prefixed $\Diamond$) for p_keyb influencing u_kg simplifies to

$$\Diamond((mode = 4) \wedge ((p\_keyb = 43) \vee (p\_keyb = 45)) \wedge \\ \Diamond(p\_weigh > 0)).$$

Thus, we know that one of the calibration keys 43 and 45 must be hit while we are in service mode ($mode = 4$) and later, some weight must be placed on the weight sensor. However, this path condition does not give any information about how to activate the service mode. By applying constant propagation, we can rule out some of the influence paths before taking the disjunction over all cycle-free ones. This way, less simplification with Lem. 3 is done and we obtain

$$\begin{aligned}
&\Diamond((mode{=}0) \wedge (p\_keyb{=}sk0) \wedge \\
&\Diamond((mode{=}1) \wedge (p\_keyb{=}sk1) \wedge \\
&\Diamond((mode{=}2) \wedge (p\_keyb{=}sk2) \wedge \\
&\Diamond((mode{=}3) \wedge (p\_keyb{=}sk3) \wedge \\
&\Diamond((mode{=}4) \wedge ((p\_keyb{=}43) \vee (p\_keyb{=}45) \wedge \\
&\qquad \Diamond(p\_weigh{>}0)))))))
\end{aligned} \tag{13}$$

We see that we must enter the service mode activation keys in the correct order to calibrate the scale. Model checking, however, reveals that the keys need not be hit consecutively.

5. Path Conditions and Model Checking

In the previous section, we have seen that LTL path conditions can become quite complex even for smallish programs. Thus, the larger a program is, the more difficult it becomes to decide without tool support whether a path condition is satisfiable. Boolean path conditions are therefore fed to a constraint solver which simplifies them as far as possible and possibly can solve them for the program's input variables. The equivalent to a constraint solver for boolean path conditions is the model checker for LTL ones: Model checkers are ideal to check LTL formulae for a model. If we transform a program into a model for a model checker, we can use the model checker to find satisfying state sequences for LTL path conditions. More precisely, not only do we have to create a compact model of the program, e.g. using slicing and program abstraction [6, 11], but we must also make sure to use SSA variables. However, it turns out that this does not severely increase the number of reachable states as many distinguished variables have the same value anyway.

If the model checker of our choice tries to find a satisfying state sequence for the LTL formula (as e.g. the explicit state model checker SPIN does with its "never claims" [15]), we simply give the simplified path condition to the model checker and run it on the program model. If the model checker tries to falsify the LTL input specification (as does e.g. the symbolic model checker NuSMV [4]), we have to input the negated LTL path condition. Note that the theory of symbolic model checking [19] also allows to compute the set of initial program states from which satisfying state sequences start, i. e. to solve the LTL path condition for the program's input variables.

For example, let us return to the case study above. Note that the path condition in (13) does not say that we must enter the service mode activation keys consecutively. This is revealed when we apply model checking: We have coded a model for the measurement software for the model checkers SPIN and NuSMV and run both of them on the formula as sketched above. NuSMV generates a trace that directly enters service mode and changes the calibration factor, the keycode sequence being 65, 43, 66, 13, 66, 45, 45, 13, 13. In contrast to that, SPIN reveals that typing almost randomly on the keyboard long enough leads to us accidentally entering service mode (the activation keys are not pressed consecutively) and allows to manipulate the displayed value. Thus, model checking has revealed a witness for the undesired manipulation of the weight on the display.

6. Related Work

The standard approach to IFC is via type systems. See [21] for an overview. While type systems are a fast approach, they usually lack precision. For example, most type systems classify `if (h) l=1; else l=0; l=2;` as insecure (where `h` is the secret variable and `l` is the public variable whose result value must not depend on `h`'s initial value).

Darvas/Hähnle/Sands [8] proposed to use Dynamic Logic and a theorem prover for information flow control. Program variables are classified as public or secret and a formula, which contains the program of interest, is set up to ensure that the initial state of the secret variables has no effect on the result in the public variables. The user then proves the formula using a semi-automatic theorem prover. This approach is not automatic and the user must provide loop invariants.

Hong et al. [16] also use temporal formulae for static program analysis: They use the CFG to construct CTL formulae that express a condition for data flowing from a variable definition to its use. Construction rules for different coverage criteria are provided. These formulae, which are built from predicates for a variable being defined/used in a state and for execution being in a specific state, are fed to the model checker SMV to automatically generate test cases. Their approach ignores control dependence and is not conservative.

In [1], Ammons/Bodík/Larus automatically extract specification automata from dynamic program traces for the correct temporal usage of APIs and ADTs based on the assumption that most usage is correct. Incorrect usage is eliminated from the specification automata while they are simplified. Verification tools such as model checkers are then used to find bugs in programs w.r.t. using the API/ADT. Although they also extract temporal specifications automatically, their extraction is not static and aggressive simplification can not guarantee conservation.

Xie and Chou [25] propose to translate static program analyses into SAT problems. Like us, they use SSA form, but they avoid the problem of repeatedly executing an assignment by heavily abstracting loops in that only the last iteration of every loop is modelled. Thus, loop-carried data dependences cannot be handled properly.

Recently, path-sensitive static program analyses [2, 12, 10] have become popular. However, directly formulating precise IFC conditions in these terms is not as easy: The SLAM project [3] provides a general path-sensitive data flow analysis [2] which operates on boolean programs which are abstracted from C programs. This way, expoiting arithmetic to prove a path unfeasible is not possible.

Fischer et al. [12] from the BLAST project propose dataflow analysis with path predicates: The merge operation does not join dataflow facts from paths if their predicates differ, but keeps track of them separately. If necessary, they can iteratively enlarge the predicate set and thus refine the analysis. Nevertheless, multiple loop iterations are hard to distinguish that way and temporal properties cannot be modelled in the predicate set.

ESP [10, 9] instruments programs to keep track of typestate changes which must satisfy the specification automaton. An interprocedural data flow analyses tries to prove correctness w.r.t. the automaton using property and path simulation [14]. They also have heuristics for when to enlarge the set of predicates of which to keep track. Like BLAST, they must be provided a specification w.r.t. which the program is verified.

Our approach generates such specifications, thus we believe that temporal path conditions can serve as specification input to path-sensitive static analysers when model checking is too time-consuming. A general specification automaton for noninterference would require many refinement iterations whereas our approach would already provide all predicates of interest in a suitable specification form.

7. Conclusion

We have seen that temporal path conditions provide precise time-dependent information about the specific circumstances of an information flow in a PDG. By transforming the program into a compact model that preserves the state sequence semantics, using e.g. slicing and program abstraction [6, 11], we can use model checkers such as SPIN [15] or NuSMV [4] to compute a specific input or state sequence for the information flow if one exists. Otherwise, we know that no information flow is possible, which will turn out to be useful for software safety and security analysis.

In fact, the approach has been developed for a while language with arrays (arrays have been left out in the current paper due to lack of space). But note that the current paper describes only theoretical foundations. Work on an implementation has just started. A lot of work remains until the idea can be applied to realistic programs in full Java.

References

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *POPL'02*, pages 4–16, 2002.

[2] T. Ball and S. K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *PASTE'01*, pages 97–103, 2001.

[3] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL'02*, pages 1–3, 2002.

[4] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *CAV'99*, pages 495–499, 1999.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, London, 2000.

[6] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE'00*, pages 439–448, 2000.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.

[8] Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *SPC'05*, pages 151–171, 2005.

[9] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI'02*, pages 57–68, 2002.

[10] D. Dhurjati, M. Das, and Y. Yang. Path-sensitive dataflow analysis with iterative refinement. In *SAS'06*, pages 425–442, 2006.

[11] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *ICSE'01*, pages 177–187, 2001.

[12] J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *FSE'05*, pages 227–236, 2005.

[13] C. Hammer, J. Krinke, and G. Snelting. Information flow control for Java based on path conditions in dependence graphs. In *ISSSE'06*, pages 87–96, 2006.

[14] H. Hampapuram, Y. Yang, and M. Das. Symbolic path simulation in path-sensitive dataflow analysis. In *PASTE'05*, pages 52–58, 2005.

[15] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[16] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE'03*, pages 232–242, 2003.

[17] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, 2003.

[18] A. Lochbihler. Temporal path conditions in dependence graphs. Master's thesis, Universität Passau, 2006. Available at http://www.infosun.fim.uni-passau.de/st/staff/lochbihl/da.pdf.

[19] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1992.

[20] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *ICSE'02*, pages 478–488, 2002.

[21] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[22] G. Snelting. Combining slicing and constraint solving for validation of measurement software. In *SAS'96*, pages 332–348, 1996.

[23] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM TOSEM*, 15(4):410–457, 2006.

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

[25] Y. Xie and A. Chou. Path sensitive program analysis using boolean satisfiability. Technical report, Stanford University, 2002.