This is the author's version of the article published in the Journal of Automated Software Engineering
The original publication is available at www.springerlink.com (DOI 10.1007/s10515-009-0050-3)

On Temporal Path Conditions in Dependence Graphs

Andreas Lochbihler · **Gregor Snelting**

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.

Keywords program dependence graph · path condition · temporal logic · security analysis

1 Introduction

Program dependence graphs (PDGs) are a well-established device to represent possible information flow in a program. They are used for program slicing, debugging, reengineering, and security analysis, for example. Information flow control (IFC),

An extended abstract of the present article appeared in the 2007 *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation* (SCAM 2007). The research of A. Lochbihler was partially supported by Deutsche Forschungsgemeinschaft, grant Sn11/9-1.

Andreas Lochbihler · Gregor Snelting
Lehrstuhl Programmierparadigmen, Universität Karlsruhe (TH), Karlsruhe, Germany
E-mail: {lochbihl,snelting}@ipd.info.uni-karlsruhe.de

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 (Snelting et al. 2006; Hammer et al. 2006). 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 (Snelting 1996), can express more detailed circumstances of a particular flow (Snelting et al. 2006); 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 feature might be quite useful in law suits.

Making path conditions work for full C and realistic programs required years of theoretical and practical work (Robschink and Snelting 2002; Robschink 2005; Snelting et al. 2006). Today, 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 summarised 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 their theoretical foundations, but we have not completely implemented them yet.

This contribution is an extended version of the one published at SCAM 2007 (Lochbihler and Snelting 2007). Apart from extra examples, definitions and intuition in all major sections, the issues with loop-carried dependences and shortcomings of boolean path conditions with respect to them are now discussed in detail in Sec. 2.2. Sec. 3.6 on how LTL path are automatically simplified has been largely rewritten as to show how this can be done effectively. The last section now also outlines limitations of and other possible areas of applications for temporal path conditions.

2 Path Conditions in Dependence Graphs

Our current work focuses 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. (Tip 1995). 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 $s \dashrightarrow t$ labelled by the control condition $\mathrm{c}(s,t)$ for t being executed depending on s, i.e. the `while` or `if` condition. For example, if s is an `if` statement with condition b and t and t' are in the `then` and `else` branch of s respectively, then $s \dashrightarrow t$ and $s \dashrightarrow t'$ are labelled b and $\neg b$ resp. We assume that the program being analysed always terminates, i.e., only statements inside the body of a `while` loop can be control dependent on the `while` condition. The reflexive and transitive closure of $\dashrightarrow$ is written $\dashrightarrow^*$.

A data dependence edge $s \overset{x}{\dashrightarrow} t$ models variable x being assigned in s and used in t without being reassigned in between. $s \overset{x}{\dashrightarrow} t$ is *loop-carried* iff there is a `while` loop node u such that $u \dashrightarrow^* s$ and $u \dashrightarrow^* t$ which can be executed while x is being passed on from s to t. Similarly, we say $s \overset{x}{\dashrightarrow} t$ *leaves a* `while` *loop* node u if $u \dashrightarrow^* s$ and not $u \dashrightarrow^* t$.

See Fig. 1 for an example program in the upper left corner and its CFG lower right. Its PDG is shown on the right-hand side where control dependences are drawn with dashed arrows, data dependences with solid ones. Instead of the control condition $\mathrm{c}(s,t)$, control dependences $s \dashrightarrow t$ are labelled with the evaluation result of the condition of s, e.g. for an `if` statement s with condition b, the label F instead of $\neg b$ is used for all t in the `else` branch of s. The data dependence $7 \overset{x}{\dashrightarrow} 9$ is loop-carried because both $4 \dashrightarrow^* 7$ and $4 \dashrightarrow^* 9$ and $7 \rightarrowtail 4 \rightarrowtail 5 \rightarrowtail 6 \rightarrowtail 9$ is a CFG path from 7 to 9 where x is not redefined in between. Equally, $5 \overset{i}{\dashrightarrow} 4$ and $5 \overset{i}{\dashrightarrow} 5$ are loop-carried. All other data dependences are not loop-carried. The only data dependence that leaves a loop is $9 \overset{y}{\dashrightarrow} 11$, which leaves the loop at node 4.

PDGs model information flow in a program: Data dependence edges represent direct flows, in which a value computed at some statement is directly used in another statement. Implicit flow, where the execution of a statement depends on the information that reaches some other statement, is captured by control 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 example, in Fig. 1, $\mathrm{BS}(7) = \{\mathrm{start}, 1, 4, 5, 6, 7\}$ and $\mathrm{FS}(1) = \{1, 4, 5, 6, 7, 9, 11\}$, hence $\mathrm{CH}(1,7) = \{1, 4, 5, 6, 7\}$.

Slicing can be used for IFC to capture both direct and implicit flows. If s is not in $\mathrm{BS}(t)$, then s cannot influence t in the sense of noninterference, i.e. the public values computed at t do not depend on the secret inputs at s. Besides, IFC considers a variety of kinds of information channels other than direct and implicit flows, see (Sabelfeld

Example program:

```
 1 i = j;
 2 y = 0;
 3 x = 1;
 4 while (i<5) {

 5   i = i+k;
 6   if (i<=4)
 7     x = a;
 8   else
 9     y = x;
10 }
11 z = y;
```

SSA transform:

```
 1 i1 = j;
 2 y1 = 0;
 3 x1 = 1;
 4 while (i2=Φ(i1,i3),y2=Φ(y1,y3),
          x2=Φ(x1,x3),i2<5) {
 5   i3 = i2+k;
 6   if (i3<=4)
 7     x3 = a;
 8   else
 9     y3 = x;
10 }
11 z = y2;
```

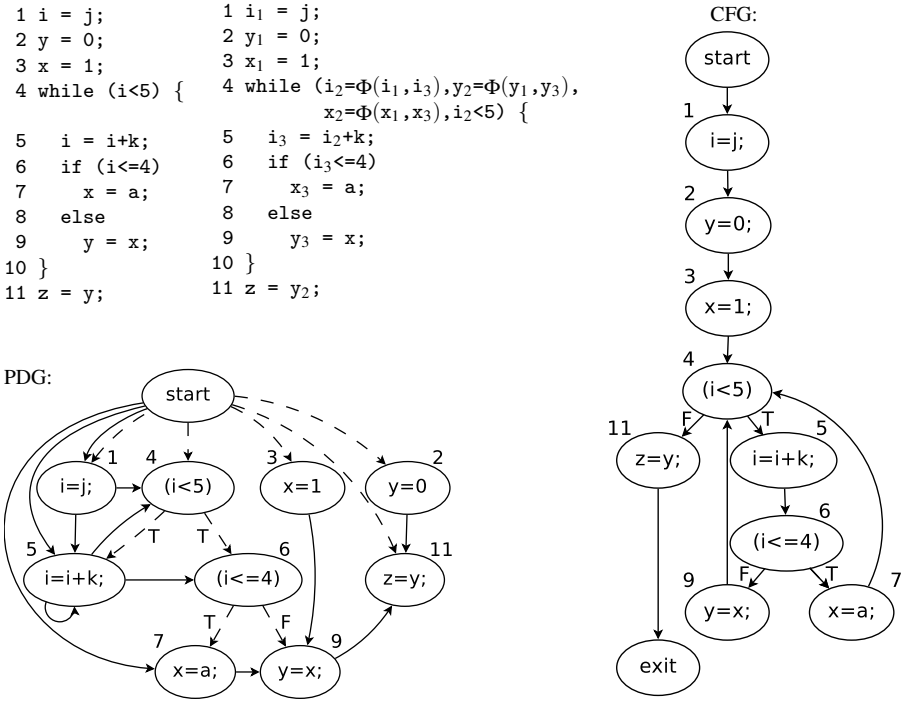

Fig. 1 An example program (upper left) with input variables a, j, k, and output variable z, its SSA transform (upper right), its CFG (lower right), and its PDG (lower left).

and Myers 2003) for an overview. Since we assume that every program execution terminates, we do not consider information flow via termination channels, although this could easily be done by adjusting the notion of control dependence (Ranganath et al. 2007). However, PDGs are not well-suited to discover flow through physical side channels such as timing channels. In what follows, information flow will always refer to direct and implicit flows.

2.1 Boolean Path Conditions

Slicing can be pretty imprecise because not every path in the PDG represents an actual flow of information: Consider, for example, the program fragment:

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

The standard PDG indicates an influence from line 1 to line 3, but the value of x can only reach y if $i > 10$ and $i+3 = 2*j-42$. Hence

$$(i > 10) \wedge (i+3 = 2*j-42) \tag{1}$$

is a necessary condition over program variables such that line 1 influences line 3. More generally:

Definition 1 In a program execution 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 execution 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 execution r.

Originally, path conditions are boolean formulae whose variables are implicitly quantified existentially (Snelting 1996), i.e. satisfiability reduces to tautology. In the following, we always omit to write the existential quantifiers. By definition, every influence between s and t occurs only along some PDG path $\pi : s \to^* t$ 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 get a path condition for s and t by taking the disjunction of these conditions for all paths between s and t. In Fig. 1, we get e.g. $\mathrm{PC}(6,11) = \mathrm{PC}(6 \blacktriangleright 7 \xrightarrow{x} 9 \xrightarrow{y} 11) \vee \mathrm{PC}(6 \blacktriangleright 9 \xrightarrow{y} 11)$.

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.

Execution conditions are built from the control dependence relation: For a statement to be executed, all conditions on at least one control dependence path from the start node to the statement must be fulfilled. Thus, we get a preliminary version for path conditions – $\mathrm{PC_B}$ meaning boolean path condition:

$$\mathrm{E}(v) := \bigvee_{\rho:\mathrm{start} \blacktriangleright^* v} \bigwedge_{u \blacktriangleright u' \in \rho} \mathrm{c}(u,u'); \quad \mathrm{PC_B}(\pi) := \bigwedge_{v \text{ node in } \pi} \mathrm{E}(v) \tag{2}$$

Due to the absorption law for $\vee$, the disjunction in (2) can be restricted to range only over all cycle-free paths ρ, thus also being finite in the presence of unstructured control flow, because cycles in the control dependence graph (CDG) do not contribute to $\mathrm{E}(v)$ (Snelting 1996). As an example, the execution condition for line 6 in Fig. 1 would be

$$\mathrm{E}(7) = c(\mathrm{start},4) \wedge c(4,6) \wedge c(6,7) = (i < 5) \wedge (i \leq 4).$$

Note, however, that the two occurrences of i in $\mathrm{E}(7)$ refer to different runtime instances of the program variable i because i gets reassigned in line 5. In order to distinguish the is in $\mathrm{E}(7)$, we transform the program into static single assignment (SSA) form (Cytron et al. 1991). 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 according to control flow. Thus, we ensure computing only correct execution conditions, i.e., they are satisfiable if the statement can be executed.

For example, reconsider Fig. 1: The program in the lower right corner shows the SSA transform of the program above. Now, the execution condition for line 7 is $\mathrm{E}(7) = (i_2 < 5) \wedge (i_3 \leq 4)$ and for line 9, we get $\mathrm{E}(9) = (i_2 < 5) \wedge \neg(i_3 \leq 4)$. Without SSA transformation, $\mathrm{E}(9)$ would be $(i < 5) \wedge \neg(i \leq 4)$, which is not satisfiable and not a necessary condition. Note that Boolean path conditions use SSA transformation only for generating conditions, not for data flow analysis, i.e. the PDG remains unchanged. In particular, in Fig. 1, node 4 does *not* define variable *i*.

To relate the variables introduced by SSA form in path conditions, Φ functions are translated into Φ conditions, which are conjunctively added to the path condition. In Fig. 1, for example, $\texttt{i}_2\texttt{=}\Phi(\texttt{i}_1\texttt{,i}_3)$ becomes $\Phi(i_2;i_1,i_3) := (i_2 = i_1) \vee (i_2 = i_3)$. Besides, there are also Φ conditions for data dependence edges $s \xrightarrow{x} t$ on a path: Let i be x's SSA index in s and j the 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 $\Phi(s \xrightarrow{x} t) = (x_i = x_j)$, which we also add conjunctively.

Consider the following program with the SSA transform to the right:

```
1 a = true;                    1 a1 = true;
2 if (b) { a = c; }            2 if (b) { a2 = c; }
3 ...                          3 a3 = Φ(a1, a2) ...
4 if (a) { x = i; }            4 if (a3) { x = i; }
```

For `a = c;` in line 2 influencing `x=i` in line 4, the path condition for $\pi =$ `a=c` $\xrightarrow{a}$ `(a)`, `(a)` $\blacktriangleright$ `x=i` with Φ conditions is

$$\mathrm{PC_B}(\pi) = \mathrm{E}(\texttt{a=c}) \wedge \mathrm{E}(\texttt{(a)}) \wedge \mathrm{E}(\texttt{x=i}) \wedge \Phi(\texttt{a=c} \xrightarrow{a} \texttt{(a)}) \wedge \Phi(a_3;a_1,a_2) = (b) \wedge \mathrm{true} \wedge (a_3) \wedge (a_3 = a_2) \wedge ((a_3 = a_1) \vee (a_3 = a_2))$$

Note that $\Phi(a_3;a_1,a_2)$ does not subsume $\Phi(\texttt{a=c} \xrightarrow{a} \texttt{(a)})$: $\mathrm{PC_B}(\pi)$ demands that a_2 be true. With $a_2 = c$, we deduce that `x=i` is only influenced if the program input c is true.

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 may simply ignore the cycle (Snelting 1996). Otherwise, no conditions are generated for the nodes on the cycle and all variables in the condition that belong to nodes before the cycle are renamed. Thus, nodes before and after the cycle are not related. For more details on how to construct boolean path conditions, see (Snelting 1996; Robschink and Snelting 2002; Snelting et al. 2006).

Obviously, one can imagine a variety of boolean path conditions for a specific path π, not only the one as presented above. These can be compared with the concept of *strength*: Since we are interested in constructing path conditions that are as precise as possible, it is desirable to limit the false witnesses to as few cases as possible.

Definition 3 (Snelting et al. 2006) 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 $\mathrm{PC_{B1}} \Rightarrow \mathrm{PC_{B2}}$.

```
1 x = 0;
2 i = 0;
3 while (i<=1) {
4   i = 2;
5   x = 1;
6 }
7 if (i>1) {
8   y = x;
9 }
```

```
1 x1 = 0;
2 i1 = 0;
3 while (i2=Φ(i1,i3),
  x2=Φ(x1,x3),i2<=1) {
4   i3 = 2;
5   x3 = 1;
6 }
7 if (i2>1) {
8   y = x2;
9 }
```

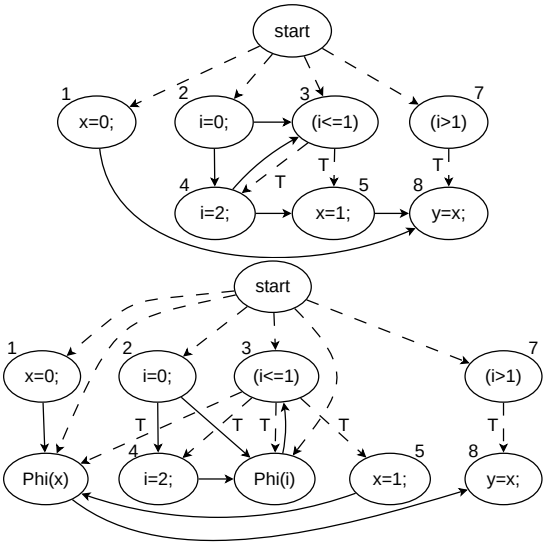

Fig. 2 Hidden loop-carried data dependences through Φ nodes.

In the example at the very beginning of Sec. 2.1, the path condition in (1) for $1 \overset{a}{\dashrightarrow} 3$ is stronger than $\mathrm{PC_B}(1 \overset{a}{\dashrightarrow} 3) = (i > 10)$, which ignores array indices.

2.2 Loop-Carried Data Dependences

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 or Φ node may be executed multiple times, e.g. if loops are present. For the PDG path $\pi := 7 \overset{x}{\dashrightarrow} 9$ in Fig. 1, we obtain the path condition $\mathrm{E}(7) \wedge \mathrm{E}(9) = (i_2 < 5) \wedge (i_3 \leq 4) \wedge \neg(i_3 \leq 4)$, which is not satisfiable although the value of `a` can reach `y` via `x`. As the data dependence edge $7 \overset{x}{\dashrightarrow} 9$ is loop-carried, $\mathrm{E}(9)$ must hold at least one loop iteration later than $\mathrm{E}(7)$, 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, the variables before and after loop-carried data dependences must be distinguished in a path condition. (Krinke 2003; Snelting et al. 2006) rename SSA variables:

$$\mathrm{PC_B}(\pi) = (i_2 < 5) \wedge (i_3 \leq 4) \wedge (i'_2 < 5) \wedge \neg(i'_3 \leq 4). \tag{3}$$

With Φ conditions, $\mathrm{PC_B}(\pi)$ from (3) then fully reads:

$$\begin{aligned}\mathrm{PC_B}(\pi) =& (i_2 < 5) \wedge (i_3 \leq 4) \wedge ((i_2 = i_1) \vee (i_2 = i_3)) \\ & \wedge (i'_2 < 5) \wedge \neg(i'_3 \leq 4) \wedge ((i'_2 = i'_1) \vee (i'_2 = i'_3)) \wedge (x = x')\end{aligned}$$

Unfortunately, separating variable names at loop-carried data dependences is not enough: Consider the program shown in the upper left corner in Fig. 2 and its PDG to

its right. The program below it is its SSA transform and the graph in the lower right corner shows the PDG with Φ nodes for the SSA transform.

Note that the boolean path condition for $5 \xrightarrow{x} 8$ is incorrect:

$$\mathrm{PC_B}(5 \xrightarrow{x} 8) = (i_2 \leq 1) \wedge (i_2 > 1) = false,$$

although the path $5 \xrightarrow{x} 8$ is possible. This is because the edge $5 \xrightarrow{x} 8$, which itself is loop-independent, subsumes the two edges $5 \xrightarrow{x_3} Phi(x)$ and $Phi(x) \xrightarrow{x_2} 8$, the first of which is loop-carried due to node 3, i.e. $5 \xrightarrow{x} 8$ actually hides a loop-carried data dependence. Thus, the two occurrences of variable i_2 in $\mathrm{PC_B}(5 \xrightarrow{x} 8)$ ought to be separated. Such hidden loop-carried data dependences were not considered in (Snelting et al. 2006). Since this problem can only arise when a data dependence edge leaves a loop, we will conservatively assume that variables have also been separated at such edges in the following.

Another problem arises when we consider the assignment to the quantified variables as a witness because such an assignment may be misleading:

```
1 if (i=0)                  1 if (i1=0)
2   while (i<2) {           2   while (i2=Φ(i1,i3), i2<2) {
3     i = i+1;              3     i3 = i2+1;
4     if (i=2)              4     if (i3=2)
5       foo();              5       foo();
6   }                       6   }
```

For instance, in the above program (with its SSA transform to the right), consider the execution condition for line 5 together with the Φ condition $\Phi(i_2; i_1, i_3)$:

$$(i_1 = 0) \wedge (i_2 < 2) \wedge (i_3 = 2) \wedge ((i_2 = i_1) \vee (i_2 = i_3)) \tag{4}$$

This implies that i_2 must be equal to 0. However, whenever line 5 is executed, i_2 cannot be equal to 0. The problem here is that $(i_2 = i_3)$ in the Φ condition relates the current value of i_2 to the value of i_3 in the previous loop iteration, but i_3 in the execution condition refers to the current value.

Nonetheless, from a formal point of view, (4) is not incorrect in the sense of Def. 1 if we consider that all variables are implicitly quantified existentially, i.e. (4) is equivalent to true. In general, boolean path conditions being correct is not affected by this problem because for our programming language, Φ constraints cannot make a path condition contradictory by themselves if there is an influencing program trace. In fact, for practical purposes, this problem is significant because we are not only interested in satisfiability, but also in witnesses, i.e. the assignments to the variables.

3 Temporal Path Conditions

Although boolean path conditions are quite strong in practice, we have seen in Sec. 2.2 that problems arise in the presence of loops and loop-carried data dependences. What is worse, boolean path conditions discard the order of the nodes on the path because the ∧ operator is commutative. In particular, we cannot express that some condition

Operator		Formal semantics	Intuitive meaning
$(\Xi, j) \vDash \Box\,\theta$	iff	$(\Xi, k) \vDash \theta$ for all $k \geq j$	θ holds from j on
$(\Xi, j) \vDash \Diamond\,\theta$	iff	$(\Xi, j) \vDash \theta$ for some $k \geq j$	θ holds some time after j
$(\Xi, j) \vDash \bigcirc\,\theta$	iff	$(\Xi, j+1) \vDash \theta$	θ holds at time $j+1$
$(\Xi, j) \vDash \eta\ \mathscr{U}\ \theta$	iff	$(\Xi, k) \vDash \theta$ for some $k \geq j$ and $(\Xi, l) \vDash \theta$ for all l, $j \leq l < k$	θ holds some time after j and η holds until then

Table 1 Semantics of the temporal operators in LTL.

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).

3.1 Linear Temporal Logic

Formulae in LTL define predicates over infinite sequences of states. Propositional LTL formulae use the standard boolean operators and the four temporal operators *always* $\Box$, *eventually* $\Diamond$, *next* $\bigcirc$, and *until* $\mathscr{U}$ to connect the boolean constants and expressions over program variables of type boolean. An LTL formula is evaluated over an infinite sequence of states, i.e. assignments to the formula's variables. Given such a sequence Ξ and a position $j \in \mathbb{N}$, we write $(\Xi, j) \vDash \theta$ the LTL formula θ holds in Ξ at time j. The boolean connectives behave as usual. For the semantics of the temporal operators, see Table 1. We say Ξ satisfies θ, denoted by $\Xi \vDash \theta$ iff $(\Xi, 0) \vDash \theta$. For notation, we assume that boolean operators bind stronger than temporal ones. For more details, see (Clarke et al. 2000).

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

Whereas LTL path conditions are based on the PDG, we obtain the state sequences which can satisfy the LTL formula from program traces, i.e. executable 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, transformating the program into SSA form is mandatory. In case the program trace terminates, the last state is repeated infinitely often to make the sequence artificially infinite. For example, when we execute the program in Fig. 1 with initial values $a = 2$, $j = 1$, and $k = 3$, we obtain the state sequence shown in Table 2.

3.2 Motivating example

As a motivating example, reconsider the program in Fig. 1, but now, assume that line 6 is replaced by `if !(i<=4)`, i.e. the `if`'s predicate is negated. The boolean path

Time	Line	a	i_1	i_2	i_3	j	k	x_1	x_2	x_3	y_1	y_2	y_3	z
0	`start`	2	$\bot$	$\bot$	$\bot$	1	3	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
1	1	2	1	$\bot$	$\bot$	1	3	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
2	2	2	1	$\bot$	$\bot$	1	3	$\bot$	$\bot$	$\bot$	0	$\bot$	$\bot$	$\bot$
3	3	2	1	$\bot$	$\bot$	1	3	1	$\bot$	$\bot$	0	$\bot$	$\bot$	$\bot$
4	5	2	1	1	4	1	3	1	1	$\bot$	0	0	$\bot$	$\bot$
5	7	2	1	1	4	1	3	1	1	2	0	0	$\bot$	$\bot$
6	5	2	1	4	7	1	3	1	2	2	0	0	$\bot$	$\bot$
7	9	2	1	4	7	1	3	1	2	2	0	0	2	$\bot$
8	11	2	1	7	7	1	3	1	2	2	0	2	2	2
9	`exit`	2	1	7	7	1	3	1	2	2	0	2	2	2
10	`exit`	2	1	7	7	1	3	1	2	2	0	2	2	2
$\vdots$	$\vdots$	$\vdots$												$\vdots$

Table 2 State sequence for the program in Fig. 1 executed with initial values $a = 2$, $j = 1$, and $k = 3$.

condition for the path $\pi = 7 \xrightarrow{x} 9$ (the only path along which 7 influences 9) is now (without Φ conditions)

$$\mathrm{PC}_{\mathrm{B}}(\pi) = (i_2 < 5) \wedge \neg(i_3 \leq 4) \wedge (i_2' < 5) \wedge (i_3' \leq 4), \tag{5}$$

which is equivalent to (3) on p. 7. Clearly, for this influence to occur, line 7 must be executed *before* line 9 and in between, execution must not leave the `while` loop in line 4. This can be expressed by the formula

$$\underbrace{(i_2 < 5) \wedge \neg(i_3 \leq 4)}_{\mathrm{E}(7)} \wedge (i_2 < 5) \mathbin{\mathscr{U}} \underbrace{((i_2 < 5) \wedge (i_3 \leq 4))}_{\mathrm{E}(9)}. \tag{6}$$

In particular, we can deduce from (6) that i_3 must be decreased between line 7 and line 9 being executed. We can use standard data flow analysis to obtain that $i_3 = i_2 + k$ always holds after having visited node 7, i.e. we can substitute $i_2 + k$ for i_3 in (6). Hence, a constraint solver can deduce that $k \geq 0$ must hold. Note that we can obtain the same result from (5) with the same techniques. However, in combination with i_3 being necessarily increased, we see that (6) is not satisfiable. Obviously, we cannot prove (5) unsatisfiable because (5) $\Leftrightarrow$ (3) 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.

3.3 Building Blocks for LTL Path Conditions

Clearly, execution conditions for PDG nodes as presented in Sec. 2 are also a core concept of LTL path conditions. Yet, execution conditions for statements can be strengthened by loop termination conditions: Suppose a node v is not control dependent on a loop predicate node u, but u dominates v in the CFG, i.e. all CFG paths

from the start node to v contain u. When execution reaches v, u must have been executed and the loop terminated. Hence, the negated predicate of u must have held the last time u has been executed. The *loop termination condition* $\mathrm{L}(v)$ for v is the conjunction thereof over all such u.

SSA form allows us to refer to past values of a variable: The assignment Ξ_j at time j to SSA variable x_i gives the value which was computed the last time the statement at which x_i is defined has been executed. Thus, $\mathrm{L}(v)$ must always hold in the LTL model when v is executed. Hence, from now on, we assume that $\mathrm{E}(v)$ contains $\mathrm{L}(v)$ (added conjunctively). For example, in Fig. 1, $\mathrm{E}(11)$ is now $\neg(i_2 < 5)$.

Similarly, when a data dependence $e = v \xrightarrow{x} w$ leaves the `while` loop node u, $\neg u$ must hold at w. The *loop termination condition* $\mathrm{L}(e)$ for e is the conjunction of all such u. If there is no such u, set $\mathrm{L}(e) = \mathit{true}$. In Fig. 1, we have $\mathrm{L}(9 \xrightarrow{y} 11) = \neg(i_2 < 5)$. Note that $\mathrm{L}(s \xrightarrow{x} v)$ and $\mathrm{L}(v)$ need not always coincide: $\mathrm{L}(s \xrightarrow{x} v)$ does not require the loop predicate nodes to dominate v whereas $\mathrm{L}(v)$ is independent of data dependences.

Further, in temporal path conditions, we can also utilise execution conditions for data dependence edges:

Definition 4 Let $s \xrightarrow{x} t$ 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 $s \xrightarrow{x} t$ is a necessary condition over program variables for any node in C being executed.

Obviously, $\mathrm{E}(s \xrightarrow{x} t) := \bigvee_{v \in C} \mathrm{E}(v)$ is a correct execution condition for $s \xrightarrow{x} t$. In Fig. 1, e.g., $\mathrm{E}(7 \xrightarrow{x} 9) = (i_2 < 5)$ and $\mathrm{E}(9 \xrightarrow{y} 11) = (i_2 < 5) \vee \neg(i_2 < 5) = \mathit{true}$. Since the models of interest for our LTL formulae contain only states for assignment nodes, we have restricted C to assignment nodes.

However, we do not include Φ constraints from Φ functions since peculiarities like in the example in Sec. 2.2 would make LTL path conditions incorrect. Yet, we do not lose any precision: For single execution conditions, they never contribute to a contradiction, and the constraints expressible by them for multiple execution conditions are trivially satisfied by every program trace, i.e. every possible model of interest for the LTL path condition. Though, we do include Φ conditions for data dependences in LTL path conditions.

3.4 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 an influence can occur along π, then the state sequence for π 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. 3.3, we now present how to combine them to obtain a correct path condition.

Fig. 3 shows the algorithm for computing the LTL path condition, which we denote by $\mathrm{PC}_{\mathrm{L}}(\pi)$, for the path π: If the path π consists of only a single node s (l. 1), then the path condition only consists of the execution condition for s. Otherwise, let e denote the first edge of the path π with source node s and target node s', and π' the rest of π (ll. 2–3).

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

PCPath(π)
1 **if** (π has only one node s) **return** $\mathrm{E}(s)$
2 **let** e,π' such that $e,\pi'=\pi$
3 **let** s be the source node of e
4 **if** (e is a control dependence)
5 **return** $(\mathrm{E}(s) \wedge \mathit{PCPath}(\pi'))$
6 **else**
7 **return** $(\mathrm{E}(s) \wedge \mathrm{E}(e)\ \mathscr{U}\ (\mathrm{L}(e) \wedge \Phi(e) \wedge \mathit{PCPath}(\pi')))$

Fig. 3 Algorithm for LTL path conditions for a single PDG path π.

If e is a control dependence edge, the path condition for π is the conjunction of the execution condition for s and the path condition for the rest π' of π (ll. 4–5). (Note that $\mathrm{PC_L}(\pi')$ starts with the execution condition for s', i.e. if $\mathrm{E}(s)$ implies $\mathrm{E}(s')$, then $\mathrm{E}(s)$ can be omitted from $\mathrm{PC_L}(\pi)$. While this is always the case in the setting of our while language, this need not hold if control flow is unstructured.)

If e is a data dependence edge for variable x (ll. 6–7), then a necessary condition for s influencing s' via $e = s \overset{x}{\dashrightarrow} s'$ is that

1. s is executed,
2. the nodes on some CFG path from s to s' are executed on which x is not redefined,
3. all loops that enclose s but not s' have terminated.
4. s' is executed, and
5. the value assigned to x at s reaches s'.

This directly gives the constraint $\mathrm{E}(s) \wedge \mathrm{E}(e)\ \mathscr{U}\ (\mathrm{L}(e) \wedge \mathrm{E}(s') \wedge \Phi(e))$. Since $\mathrm{E}(s')$ is already part of the path condition for the remaining path π', *PCPath* drops it in its recursive call (l. 7).

For example, in Fig. 1, consider $\pi = 1 \overset{i}{\dashrightarrow} 5, 5 \overset{i}{\dashrightarrow} 6, 6 \dashrightarrow \blacktriangleright 9$. The LTL path condition $\mathrm{PC_L}(\pi)$ for π is

$$\begin{aligned}
&\mathrm{E}(1) \wedge \mathrm{E}(1 \overset{i}{\dashrightarrow} 5)\ \mathscr{U}\ (\mathrm{L}(1 \overset{i}{\dashrightarrow} 5) \wedge \Phi(1 \overset{i}{\dashrightarrow} 5) \wedge \mathrm{E}(5) \wedge \\
&\qquad \mathrm{E}(5 \overset{i}{\dashrightarrow} 6)\ \mathscr{U}\ (\mathrm{L}(5 \overset{i}{\dashrightarrow} 6) \wedge \Phi(5 \overset{i}{\dashrightarrow} 6) \wedge \mathrm{E}(6) \wedge \mathrm{E}(9))) = \\
&\mathrm{true} \wedge \mathrm{true}\ \mathscr{U}\ (\mathrm{true} \wedge (i_1 = i_2) \wedge (i_2 < 5) \wedge \\
&\qquad (i_2 < 5)\ \mathscr{U}\ (\mathrm{true} \wedge (i_3 = i_3) \wedge (i_2 < 5) \wedge \\
&\qquad\quad ((i_2 < 5) \wedge \neg(i_3 \leq 4))))
\end{aligned} \tag{7}$$

which simplifies to (cf. Sec. 3.6)

$$\Diamond((i_1 = i_2) \wedge (i_2 < 5)\ \mathscr{U}\ ((i_2 < 5) \wedge (i_3 > 4))). \tag{8}$$

There is also a semantic justification for this algorithm: The main idea is that for any program state sequence $\Xi = (\xi_i)_i$, if statement s influences t along π in $(\xi_i)_{j \leq i \leq k}$, then there is an $l \in \{j, \ldots, k\}$ such that in $(\xi_i)_{j \leq i \leq l}$ s influences some s' along e and in $(\xi_i)_{l \leq i \leq k}$ s' influences t along π'. For data dependence edges on π, this gives an $\mathscr{U}$ operator each (l. 7 in Fig. 3). However, we can often omit it (cf. Sec. 3.6). For control dependence edges, there is no need for for an $\mathscr{U}$ operator: Since we do not have states for control nodes, if π starts with a control dependence edge e, we have $l = j$, i.e. the execution condition for s must obviously hold in s', too.

Input: PDG $G = (V, E)$; cycle-free PDG path π
Output: $\overline{PC}_{\mathrm{L}}(\pi)$

PCPath'(G, π)
1 **if** (π has only one node s) **return** $\mathrm{E}(s)$
2 **let** e,π' such that e, π'=π
3 **let** s be the source and s' the target node of e
4 **let** A be the set of data dependences on any cycle through s' in G
5 **if** ($A \neq \emptyset$ and $s' \neq t$) **set** θ:=$\bigvee_{a \in A} \mathrm{E}(a)$ **else set** θ:=$false$
6 **if** ($e = s \blacktriangleright s'$)
7 **return** $(\mathrm{E}(s) \wedge \theta \mathbin{\mathscr{U}} PCPath'(G,\pi'))$
8 **else**
9 **return** $(\mathrm{E}(s) \wedge \mathrm{E}(e) \mathbin{\mathscr{U}} (\mathrm{L}(e) \wedge \Phi(e) \wedge \mathrm{E}(s') \wedge$
10 $\theta \mathbin{\mathscr{U}} (\mathrm{L}(e) \wedge PCPath'(G, \pi'))))$

Fig. 4 Algorithm *PCPath'* for LTL path conditions for cycle-free paths.

3.5 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 $\mathrm{PC}_{\mathrm{L}}(\pi)$ over all paths $\pi : s \rightarrow^* t$ may result in infinite formulae if the PDG contains cycles. Thus, we adapt our algorithm (cf. Fig. 4):

For a cycle-free path $\pi : s \rightarrow^* t$, *PCPath'* computes an LTL path condition $\overline{PC}_{\mathrm{L}}(\pi)$ that is satisfied by every program state sequence in which s influences t along some path π' from which we obtain π again by removing all cycles from π'. The idea is to conservatively ignore what may happen while the program execution is inside such a cycle.

For example, consider the PDG on the right hand side of Fig. 5 and let $\pi = 5 \xrightarrow{a} 6, 6 \xrightarrow{y} 9$. The new path condition for π is supposed to capture not only any information flow along π itself, but also along any other path π' that starts with the edge $5 \xrightarrow{a} 6$ and ends with the edge $6 \xrightarrow{y} 9$, but that goes in between arbitrarily many times through the cycle $6 \xrightarrow{y} 7, 7 \xrightarrow{x} 6$. To this end, we separately generate the constraints for both subpaths of π before and after the cycle, i.e. $5 \xrightarrow{a} 6$ and $6 \xrightarrow{y} 9$. Then, we join them again with an $\mathscr{U}$ operator such that its first operand is satisfied all the time while execution remains inside the dependence cycle $6 \xrightarrow{y} 7, 7 \xrightarrow{x} 6$.

In general (Fig. 4), we insert at every node v ($v \notin \{s,t\}$) of the path π an extra until operator (ll. 7, 10) whose first operand is a necessary execution condition θ for all nodes which control flow can reach when v influences itself via some cycle in the PDG. Let A be the set of data dependences that are part of cycles which contain v (l. 4). Since the PDG is finite, so is A, i.e., $\theta = \bigvee_{a \in A} \mathrm{E}(a)$ is a finite, necessary condition for all states visited while v influences itself. We can ignore control dependence because there are no states for control dependences in our model. If no such cycle exists, this condition θ is false (l. 5), i.e. the extra until operator is removed again by simplification of the formula.

In Fig. 1 (cf. (7)), for example, let us look again at the acyclic path $\pi = 1 \xrightarrow{i} 5$, $5 \xrightarrow{i} 6, 6 \blacktriangleright 9$. Cycles can only be added to π after the first edge, namely $5 \xrightarrow{i} 5$ and $5 \xrightarrow{i} 4, 4 \blacktriangleright 5$, but there is no cycle at node 6. Thus, *PCPath'* computes the following

```
 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 }
```

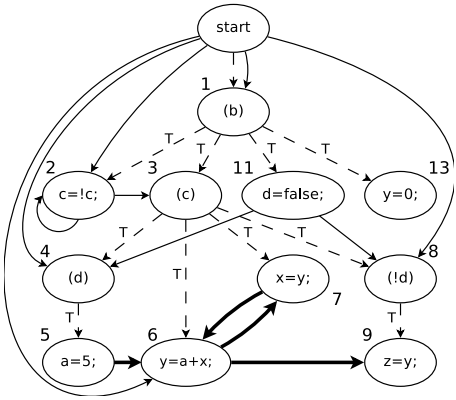

Fig. 5 An example program and its PDG to show the extra $\mathscr{U}$ operator for cycles being necessary.

path condition where the additional parts (when compared to (7)) are set in bold face:

$$\begin{aligned}&\mathrm{E}(1)\wedge\mathrm{E}(1\overset{i}{\dashrightarrow}5)\ \mathscr{U}\ (\mathrm{L}(1\overset{i}{\dashrightarrow}5)\wedge\Phi(1\overset{i}{\dashrightarrow}5)\wedge\\&\mathrm{E}(5)\wedge(\mathrm{E}(5\overset{i}{\dashrightarrow}5)\vee\mathrm{E}(5\overset{i}{\dashrightarrow}4))\ \mathscr{U}\ (\mathrm{L}(1\overset{i}{\dashrightarrow}5)\wedge\\&\mathrm{E}(5)\wedge\mathrm{E}(5\overset{i}{\dashrightarrow}6)\ \mathscr{U}\ (\Phi(5\overset{i}{\dashrightarrow}6)\wedge\mathrm{E}(6)\wedge \mathit{false}\ \mathscr{U}\ (\mathrm{E}(9))))\end{aligned}\tag{9}$$

That is, we have included "$\mathrm{E}(4)\wedge(\mathrm{E}(4\overset{i}{\dashrightarrow}4)\vee\mathrm{E}(4\overset{i}{\dashrightarrow}3))\ \mathscr{U}\ (\mathrm{L}(1\overset{i}{\dashrightarrow}4)$" and "$\mathit{false}\ \mathscr{U}$" in (7). Simplifying (9) removes the extra $\mathscr{U}$ operator again and we obtain (8) once again.

Note that this need not always be the case: Consider Fig. 5 and the path $\rho = 5\overset{a}{\dashrightarrow}6, 6\overset{y}{\dashrightarrow}7, 7\overset{x}{\dashrightarrow}6, 6\overset{y}{\dashrightarrow}9$. By removing the cycle from ρ, we obtain the path $\pi = 5\overset{a}{\dashrightarrow}6, 6\overset{y}{\dashrightarrow}9$. $\mathrm{PC}_\mathrm{L}(\pi)$ (without Φ constraints) simplifies to $d\wedge(b\wedge c)\ \mathscr{U}\ (b\wedge c\wedge\neg d)$. Note that $\mathrm{PC}_\mathrm{L}(\pi)$ is not satisfiable over the program traces.

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

With $\overline{PC}_\mathrm{L}$ and *PCPath'*, we compute a path condition $\mathrm{PC}_\mathrm{L}(s,t)$ for two statements s and t by taking the disjunction over all cycle-free paths in the PDG G between s and t. This is done by the algorithm *PCChop* shown in Fig. 6.

For example, consider again Fig. 1: The chop $\mathrm{CH}(1,9)$ contains eight acyclic paths, namely: $\pi_1 = 1\overset{i}{\dashrightarrow}5\overset{i}{\dashrightarrow}6\blacktriangleright 9$, $\pi_2 = 1\overset{i}{\dashrightarrow}5\overset{i}{\dashrightarrow}6\blacktriangleright 7\overset{i}{\dashrightarrow}9$, $\pi_3 = 1\overset{i}{\dashrightarrow}5\overset{i}{\dashrightarrow}4\blacktriangleright 6\overset{i}{\dashrightarrow}9$, $\pi_4 = 1\overset{i}{\dashrightarrow}5\overset{i}{\dashrightarrow}4\blacktriangleright 6\blacktriangleright 7\overset{i}{\dashrightarrow}9$, $\pi_5 = 1\overset{i}{\dashrightarrow}4\blacktriangleright 6\blacktriangleright 9$, $\pi_6 = 1\overset{i}{\dashrightarrow}4\blacktriangleright 6\blacktriangleright 7\overset{i}{\dashrightarrow}9$, $\pi_7 = 1\overset{i}{\dashrightarrow}4\blacktriangleright 5\overset{i}{\dashrightarrow}6\blacktriangleright 9$, and $\pi_8 = 1\overset{i}{\dashrightarrow}4\blacktriangleright 5\overset{i}{\dashrightarrow}6\blacktriangleright 7\overset{i}{\dashrightarrow}9$. *PCChop* invokes *PCPath'* for each π_i, $1\le i\le 8$. As we have already seen above, *PCPath'*(π_5) simplifies to (8) on p. 12. Simplifying the formulae generated by *PCPath'* for π_1, π_3, and π_7 also yields (8). The

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

PCChop(G, s, t)
1 **let** $I, (\pi_i)_{i\in I}$ such that $(\pi_i)_{i\in I}$ enumerates
2 all cycle-free paths $s \to^* t$ in G
3 **return** $\bigvee_{i\in I} PCPath'(G, \pi_i)$

Fig. 6 Algorithm *PCChop* for LTL path conditions for a chop between s and t in the PDG G.

formulae generated for π_2, π_4, π_6, and π_8 are analogously simplified to

$$\Diamond((i_1 = i_2) \wedge (i_2 < 5)\ \mathscr{U}\ ((i_3 \leq 4) \wedge (i_2 < 5)\ \mathscr{U}\ ((i_2 < 5) \wedge (i_3 > 4)))) \qquad (10)$$

Hence, *PCChop* computes the disjunction of (8) and (10), which is again simplified to yield (8).

The next lemma shows that *PCPath'* computes a path condition $\overline{PC}_{\mathrm{L}}(\pi)$ for a cycle free path which is satisfied by any program trace which carries information along some path π' from which we can obtain π again by removing cycles:

Lemma 1 (Correctness of the algorithm *PCPath'*) *Let $\pi' : s \to^* t$ be a PDG path and let π be the path π' from which all cycles have been removed. Let Ξ be any state sequences such that $\Xi \vDash \Diamond \mathrm{PC}_{\mathrm{L}}(\pi')$. Then $\Xi \vDash \Diamond \overline{PC}_{\mathrm{L}}(\pi)$.*

Proof. Suppose an LTL formula X is of the form

$$A_1\ \mathscr{U}\ (\ldots \wedge A_2\ \mathscr{U}\ (\ldots(\ldots \wedge A_n\ \mathscr{U}\ B))$$

for some LTL formula B and some boolean formulae A_i, $1 \leq i \leq n$. Then, $\Xi, j \vDash X$ implies that $\Xi, j \vDash (\bigvee_{i=1}^{n} A_i)\ \mathscr{U}\ B$ for any state sequence Ξ and time j. Hence, if ρ is a cycle in π', say $\pi' = \pi'_1, \rho, \pi'_2$, then $\mathrm{PC}_{\mathrm{L}}(\rho, \pi'_2)$ is $\mathrm{PC}_{\mathrm{L}}(\rho)$ in which $\mathrm{PC}_{\mathrm{L}}(\pi'_2)$ has been conjunctively added to the execution condition for the last node on ρ. Moreover, the part of $\mathrm{PC}_{\mathrm{L}}(\rho, \pi'_2)$ up to $\mathrm{PC}_{\mathrm{L}}(\pi'_2)$ is of the above form, so if $\Xi, j \vDash \mathrm{PC}_{\mathrm{L}}(\rho, \pi'_2)$, then also $\Xi, j \vDash \theta\ \mathscr{U}\ \mathrm{PC}_{\mathrm{L}}(\pi'_2)$ where $\theta = \bigvee_{u \xrightarrow{x} w \in \rho} \mathrm{E}(u \xrightarrow{x} w)$ for all times j. If π'_1 is empty, then $\Diamond \mathrm{PC}_{\mathrm{L}}(\pi') = \Diamond \mathrm{PC}_{\mathrm{L}}(\rho, \pi'_2)$ and from $\Xi \vDash \Diamond \mathrm{PC}_{\mathrm{L}}(\pi')$ we get $\Xi \vDash \Diamond(\theta\ \mathscr{U}\ \mathrm{PC}_{\mathrm{L}}(\pi'_2))$, which is equivalent to $\Xi \vDash \Diamond \mathrm{PC}_{\mathrm{L}}(\pi'_2)$. If π'_2 is not empty, then let η be $\mathrm{PC}_{\mathrm{L}}(\pi')$ in which $\mathrm{PC}_{\mathrm{L}}(\rho, \pi'_2)$ has been replaced by $\theta\ \mathscr{U}\ \mathrm{PC}_{\mathrm{L}}(\pi'_2)$. As $\mathscr{U}$ is monotone w.r.t. implication in the second operand, from $\Xi \vDash \Diamond \mathrm{PC}_{\mathrm{L}}(\pi')$ follows $\Xi \vDash \Diamond \eta$.

When we apply the above reasoning to all cycles we have removed from π' to obtain π, we obtain a formula F which is structurally isomorphic to $\overline{PC}_{\mathrm{L}}(\pi)$. Since the first operand of any $\mathscr{U}$ operator in $\overline{PC}_{\mathrm{L}}(\pi)$ is implied by the first operand of the corresponding $\mathscr{U}$ in F, and $\mathscr{U}$ is monotone in both operands, we also get $\Xi \vDash \Diamond \overline{PC}_{\mathrm{L}}(\pi)$. □

From this lemma and $\mathrm{PC}_{\mathrm{L}}(\pi')$ being correct for all paths π', it directly follows that *PCChop* is correct:

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

E1: $\text{true} \mathbin{\mathscr{U}} A \rightsquigarrow \Diamond A$

E2: $\text{false} \mathbin{\mathscr{U}} A \rightsquigarrow A$

E3: $A \mathbin{\mathscr{U}} \text{true} \rightsquigarrow \text{true}$

E4: $A \mathbin{\mathscr{U}} \text{false} \rightsquigarrow \text{false}$

E5: $\Diamond(A \mathbin{\mathscr{U}} B) \rightsquigarrow \Diamond B$

E6: $A \mathbin{\mathscr{U}} (\Diamond B) \rightsquigarrow \Diamond B$

E7: $\Diamond\Diamond A \rightsquigarrow \Diamond A$

E8: $\Diamond\text{true} \rightsquigarrow \text{true}$

E9: $\Diamond\text{false} \rightsquigarrow \text{false}$

E10: $A \mathbin{\mathscr{U}} B \vee A \mathbin{\mathscr{U}} C \rightsquigarrow A \mathbin{\mathscr{U}} (B \vee C)$

E11: $$\frac{B \Rightarrow A}{A \mathbin{\mathscr{U}} (B \mathbin{\mathscr{U}} C) \rightsquigarrow A \mathbin{\mathscr{U}} C}$$

E12: $$\frac{A \Rightarrow B}{A \mathbin{\mathscr{U}} (B \mathbin{\mathscr{U}} C) \rightsquigarrow B \mathbin{\mathscr{U}} C}$$

E13: $$\frac{C \Rightarrow A \qquad D \Rightarrow B}{A \mathbin{\mathscr{U}} (B \wedge C \mathbin{\mathscr{U}} D) \rightsquigarrow A \mathbin{\mathscr{U}} D}$$

E14: $$\frac{A \Rightarrow C \qquad B \Rightarrow D}{A \wedge B \mathbin{\mathscr{U}} (C \wedge D \mathbin{\mathscr{U}} E) \rightsquigarrow A \wedge D \mathbin{\mathscr{U}} E}$$

E15: $$\frac{B \Rightarrow A \qquad C \Rightarrow A}{A \wedge B \mathbin{\mathscr{U}} C \rightsquigarrow B \mathbin{\mathscr{U}} C}$$

Fig. 7 LTL equivalences as rewrite rules for simplifying path conditions.

Proof. By Lem. 1, $\mathrm{PC}_\mathrm{L}(\pi') \Rightarrow \overline{PC}_\mathrm{L}(\pi)$ for all paths $\pi' : s \rightarrow^* t$ such that π is π' without the cycles in π', thus $\mathrm{PC}_\mathrm{L}(\pi') \Rightarrow \mathrm{PC}_\mathrm{L}(s,t)$ by definition of $\mathrm{PC}_\mathrm{L}(s,t)$. If s influences t via some path $\rho : s \rightarrow^* t$, we have that $\Xi \vDash \mathrm{PC}_\mathrm{L}(\rho)$ for some program state sequence Ξ, because $\mathrm{PC}_\mathrm{L}(\rho)$ is correct. Thus, also $\Xi \vDash \mathrm{PC}_\mathrm{L}(s,t)$. □

3.6 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). Fig. 7 shows a number of LTL equivalences as rewrite rules which we have found useful for simplifying path conditions. Note that this list is not complete. The rewrite rules E11, E12, E13 and E14 have implications as premises. By construction of path conditions, the first operand of an $\mathscr{U}$ operator is always a boolean formula and at most one operand of any maximal conjunction is not boolean. Hence, the right-hand sides of these rules are always instantiated with boolean formulae. Their antecedent, however, may be instantiated with formulae that contain $\mathscr{U}$ operators, which are never in the scope of a negation operator. Thus, by using the LTL implication $A \mathbin{\mathscr{U}} B \Rightarrow A \vee B$, we can reduce implications in the rules' premises to the boolean level where SAT or SMT solvers can be applied to decide them automatically. In fact, SAT solvers are sufficiently powerful for most of them because frequently a program predicate occurs multiple times in a formula, i.e. we do not interpret boolean program expressions, but treat them like propositional variables.

For example, the path condition in (9) on p. 14 is

$$\text{true} \wedge \text{true} \mathbin{\mathscr{U}} (\text{true} \wedge (i_1 = i_2) \wedge (i_2 < 5) \wedge ((i_2 < 5) \vee (i_2 < 5)) \mathbin{\mathscr{U}} (\text{true} \wedge (i_2 < 5) \wedge (i_2 < 5) \mathbin{\mathscr{U}} ((i_3 = i_3) \wedge (i_2 < 5) \wedge \text{false} \mathbin{\mathscr{U}} ((i_2 < 5) \wedge \neg(i_3 \leq 4)))))$$

After having simplified all boolean trivialities in this formula, we apply to it the simplification rules E1 and E2 and obtain:

$$\Diamond((i_1 = i_2) \wedge (i_2 < 5) \wedge (i_2 < 5)\ \mathcal{U}\ (\underbrace{(i_2 < 5)}_{A} \wedge \underbrace{(i_2 < 5)}_{B}\ \mathcal{U}\ (\underbrace{(i_2 < 5) \wedge (i_3 > 4)}_{C})))$$

to which we apply E15 as indicated:

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

but now, we can directly apply E12, which removes the additional $\mathcal{U}$ operator that has been introduced to account for cycles at node 3:

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

Applying E15 once more yields (8) on p. 12.

Also, we can often simplify LTL path conditions by slightly weakening them using LTL implications instead of equivalences. Since they are only necessary conditions for an influence, this does not affect their correctness. For example, we have the two extra rewrite rules with lower priority than the rules in Fig. 7:

$$\textbf{I1}\colon\ (A\ \mathcal{U}\ C) \vee (B\ \mathcal{U}\ C) \rightsquigarrow (A \vee B)\ \mathcal{U}\ C$$
$$\textbf{I2}\colon\quad A\ \mathcal{U}\ (B\ \mathcal{U}\ C) \rightsquigarrow (A \vee B)\ \mathcal{U}\ C$$

Although all the 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 cannot completely avoid $\mathcal{U}$ operators. Besides, loop termination conditions require them, too. For example, the $\mathcal{U}$ operator along $9 \xrightarrow{y} 11$ 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 $u \xrightarrow{x} v$ 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 $u \xrightarrow{x} v$ in $\mathrm{PC_L}(\pi)$ (or in $\overline{PC}_{\mathrm{L}}(\pi)$ after it having been simplified with E2), 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 (Lochbihler 2006). 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. Hence, they have the same value at both states before and after the $\mathcal{U}$ operator.

For example (Fig. 1), let π be as in (7) and (8) on p. 12. Note that both $5 \xrightarrow{i} 6$ and $1 \xrightarrow{i} 5$ are not loop-carried and $5 \xrightarrow{i} 6$ does not leave a loop. By Lem. 2, we can take

what is underlined and insert these terms (set in bold face) after the $\mathscr{U}$ operator for $5 \overset{i}{\dashrightarrow} 6$ into (7):

$$\text{true} \wedge \text{true}\ \mathscr{U}\ (\underline{\text{true} \wedge (i_1 = i_2) \wedge (i_2 < 5)} \wedge (i_2 < 5)\ \mathscr{U}\ (\text{true} \wedge (i_3 = i_3) \wedge$$
$$(i_2 < 5) \wedge ((i_2 < 5) \wedge \neg(i_3 \leq 4)) \wedge \underline{\text{true} \wedge (i_1 = i_2) \wedge (i_2 < 5)}))$$

Since $\mathscr{U}$ is monotone w.r.t. implication, we can now drop the underlined terms and then apply the above simplification techniques to get

$$\Diamond((i_1 = i_2) \wedge (i_2 < 5) \wedge \neg(i_3 > 4)),$$

which contains one temporal operator less than (8).

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 $\mathscr{U}$ operator nested more deeply and then apply E11, E12 and I2 to properly remove the $\mathscr{U}$ operator.

Regarding path conditions for chops, the next lemma utilises the disjunctive absorption law "If A implies B then $A \vee B \Leftrightarrow B$" such that we only have to consider fewer paths in the outer disjunction (ll. 1–2 in Fig. 6).

Lemma 3 *Let π, ρ be PDG paths. If for some suffix σ of π, $\Diamond \overline{PC}_{\mathrm{L}}(\sigma)$ implies $\Diamond \overline{PC}_{\mathrm{L}}(\rho)$, 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. For the first claim suppose that the state sequence $\Xi = (\xi_i)_{i \in \mathbb{N}}$ satisfies $\Diamond \overline{PC}_{\mathrm{L}}(\pi)$. Let $\pi = \sigma', \sigma$ such that $\Diamond \overline{PC}_{\mathrm{L}}(\sigma) \Rightarrow \Diamond \overline{PC}_{\mathrm{L}}(\rho)$. Then $\overline{PC}_{\mathrm{L}}(\pi)$ is of the form

$$\underbrace{A_0 \wedge A_1\ \mathscr{U}\ (\ldots \wedge A_k\ \mathscr{U}\ (A_{k+1})}_{\text{condition for } \sigma'} \wedge \underbrace{A'_{k+1} \wedge A_{k+2}\ \mathscr{U}\ (\ldots))}_{\overline{PC}_{\mathrm{L}}(\sigma)}) \tag{11}$$

Hence, there is a time j such that $(\Xi, j) \vDash \overline{PC}_{\mathrm{L}}(\sigma)$. Therefore $\Xi \vDash \Diamond \overline{PC}_{\mathrm{L}}(\sigma)$. From this, the claim follows with $\Diamond \overline{PC}_{\mathrm{L}}(\sigma) \Rightarrow \Diamond \overline{PC}_{\mathrm{L}}(\rho)$.

For the second claim, let $\sigma' = \rho$ and $\pi = \sigma', \sigma$. Then, $\overline{PC}_{\mathrm{L}}(\pi)$ is of the form shown in (11). Let θ be $\overline{PC}_{\mathrm{L}}(\pi)$ with $\overline{PC}_{\mathrm{L}}(\sigma)$ replaced by true. Since $\mathscr{U}$ is monotone w.r.t. implication, if we have $\overline{PC}_{\mathrm{L}}(\pi) \Rightarrow \theta$ and $\theta \Leftrightarrow \overline{PC}_{\mathrm{L}}(\rho)$. □

Once simplification is done, we look for large 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 these paths from the outermost disjunction (cf. Cor. 1, ll. 1–2 in Fig. 6).

To increase chances of showing unsatisfiability we can include extra constraints which can be generated from other analysis techniques for data and control flow. For example, there is only a single definition for every SSA variable. Hence, we can back-substitute these definitions in the formula until we reach a Φ function definition, provided we do not pass along a loop-carried data dependence and no definition of the variables introduced by the substitution can possibly be executed along the data dependence edges for the substitution. Similarly, when control flow passes along a data dependence edge, we often know how a Φ function must evaluate because we can rule out some other CFG paths. In the motivating example from Sec. 3.2 (cf.

```
1 i1 = 0;
2 while (i2=Φ(i1,i3);b) {
3   x = z;
4   i3=i2+1;
5   if (i3=2) {
6     y = x;
7   }
8   z = a;
9 }
```

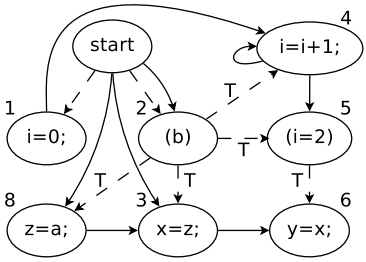

Fig. 8 Example program in SSA form to show that Lem. 2 and other analysis techniques are not orthogonal.

Fig. 1), we know that $i_3 = i_2 + k$ when we are at node 7, i.e. in (6) on p. 10, we substitute i_3 by $i_2 + k$ in $\mathrm{E}(7)$, so we deduce that $k \geq 0$. Moreover, since we must have executed 7 beforehand in a previous loop iteration, the Φ function $\Phi(i_1, i_3)$ must have evaluated to i_3. With this, (6) gives us that i_3 must be decreased whereas $k \geq 0$ makes i_3 being increased, a contradiction.

Unfortunately, this approach interferes with the simplification offered by Lem. 2. Consider the program in SSA form in Fig. 8. The path condition $\overline{PC}_{\mathrm{L}}(\pi)$ for $\pi = 8 \xrightarrow{z} 3, 3 \xrightarrow{y} 6$ is after simplification with E2

$$(b) \wedge (b)\ \mathscr{U}\ ((b) \wedge (b)\ \mathscr{U}\ ((b) \wedge (i_3 = 2))).$$

Now, we add $i_3 = i_2 + 1$, which must hold at line 6, and get

$$(b) \wedge (b)\ \mathscr{U}\ ((b) \wedge (b)\ \mathscr{U}\ ((b) \wedge (i_3 = 2) \wedge (i_3 = i_2 + 1))).$$

From $8 \xrightarrow{z} 3$, we know that $i_2 = i_3$ must hold at line 3. Thus, we add this, too:

$$(b) \wedge (b)\ \mathscr{U}\ ((b) \wedge (i_2 = i_3) \wedge (b)\ \mathscr{U}\ ((b) \wedge (i_3 = 2) \wedge (i_3 = i_2 + 1)))$$

If we were now to apply the ideas of Lem. 2 to $3 \xrightarrow{y} 6$, we would no longer be conservative, because

$$(b) \wedge (b)\ \mathscr{U}\ ((b) \wedge (i_2 = i_3) \wedge (b)\ \mathscr{U}\ ((i_2 = i_3) \wedge (b) \wedge (i_3 = 2) \wedge (i_3 = i_2 + 1)))$$

is equivalent to false, even though in every program execution in which the loop is entered the value of `a` reaches `y` via `z` and `y`.

Due to the intricacies involved with simplification, we have not yet implemented this additional approach. Note that although we can analyse temporal path conditions in this way by combining a number of other static analyses, our focus lies on model checking them (cf. Sec. 5).

3.7 Comparing Boolean and LTL Path Conditions

In the motivating example of Sec. 3.2, 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 (and all paths from which we can obtain π by removing cycles) in which all unnecessary $\mathscr{U}$ operators for data dependences have been dropped (cf. Lem. 2) and $\mathscr{U}$ operators for all cycles that must be inserted into π have been included (cf. Sec. 3.5). 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 (Lochbihler 2006). 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. The idea is that separation of variables in η corresponds to $\mathscr{U}$ operators that have not been removed in θ, i.e., every maximal boolean conjunction in θ corresponds to a conjunction of atomic formulae in η, which are not related to other parts of η, and vice versa. These atomic formulae are all contained in one maximal conjunction of θ except for Φ constraints from Φ functions. However, these Φ constraints cannot turn the conjunction of atomic formulae unsatisfiable if there is a witness for the influence. Since all atomic formulae of η are covered by conjunctions in θ, we can construct a satisfying assignment from the witness for π.

Apart from temporal operators, loop termination conditions, which we cannot 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*.

3.8 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. Considering LTL to be a part of CTL* (Clarke et al. 2000), 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 $\mathscr{U}$ 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 cannot ensure that $\Diamond \eta$ holds in the same program trace as θ does.

In general, model checking for LTL has a worst-case complexity that is exponential in the size of the formula whereas for CTL the worst case complexity is linear in the length of the formula. It is linear in the size of the program for both of them (Clarke et al. 2000). In light of the rapid growth in size for path condition, this seems to be a telling argument for CTL.

But, in fact, we only use a small subset of LTL: At most one operand in every maximal conjunction of an LTL path condition is not a boolean formula. Hence, when the LTL formula is converted into a Büchi automaton for model checking, this can be done in time linear in the size of the formula and the automaton's states set has linear size, too. Thus, model checking an LTL path condition is also linear both in the state space and the formula length.

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. A more detailed description can be found in (Lochbihler 2006). Fig. 9 shows a fragment from the measurement software of a fictitious cheese weighing scale. There are two input ports `input1` and `input2` from which the keystrokes and the weight sensor data, respectively, are read. In normal mode, the measurement software computes the weight in kg from the weight sensor data and the calibration factor `kal_kg` (ll. 28–29). In service mode, which is activated by entering a specific sequence of key strokes (ll. 8–19), the calibration factor can be adjusted by the keyboard (ll. 21-27). We want to check if we can manipulate the weight value `u_kg` – shown on the display – by the keyboard. The corresponding path condition for `p_keyb` influencing `u_kg` is

$$\mathrm{PC_L}(3.1,29) \vee \mathrm{PC_L}(10,29) \vee \mathrm{PC_L}(26,29) \vee \mathrm{PC_L}(32,29) \vee \mathrm{PC_L}(35,29)$$

where 3.1 refers to the first statement in l. 3. Fig. 10 shows the chop between ll. 3, 10, 26, 32, 35 and 29, which has been closed under control dependence. Note that 26 and 32 are not part of the chop, i.e. slicing is already able to deduce that these two statements cannot influence 29. Hence, there is no need to compute $\mathrm{PC_L}(26,29)$ and $\mathrm{PC_L}(32,29)$ because both are false.

The remaining path condition (without SSA indices and Φ constraints, but with the prefixed $\Diamond$) simplifies to

$$\Diamond((mode = 5) \wedge ((p\_keyb = 43) \vee (p\_keyb = 45)) \wedge \Diamond(p\_weigh > 0)). \qquad (12)$$

Thus, we know that one of the calibration keys 43 and 45 must be hit while we are in service mode ($mode = 5$) 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 along data dependences, we eliminate most of the edges in the upper half of Fig. 10 before taking the disjunction over

```
 1 mode = 0;
 2 sk0 = 65; sk1 = 43; sk2 = 66; sk3 = 45; sk4 = 13;
 3 p_keyb = input1; p_weigh = input2;
 4 while (true) {
 5   if (p_keyb = 27)
 6     mode = 0;
 7   else {
 8     if ((mode = 4) && (p_keyb = sk4)) {
 9       mode = 5;
10       p_keyb = input1;
11     }
12     if ((mode = 3) && (p_keyb = sk3))
13       mode = 4
14     if ((mode = 2) && (p_keyb = sk2))
15       mode = 3;
16     if ((mode = 1) && (p_keyb = sk1))
17       mode = 2;
18     if ((mode = 0) && (p_keyb = sk0))
19       mode = 1;
20   }
21   if (mode = 5) {
22     if (p_keyb = 43)
23       kal_kg = kal_kg+1;
24     if (p_keyb = 45)
25       kal_kg = kal_kg-1;
26     p_keyb = input1;
27   }
28   if (p_weigh > 0)
29     u_kg = p_weigh * kal_kg;
30   if (p_keyb = 13)
31     for (i = 0; i < 8; i++) {
32       p_keyb = input1;
33       output = p_keyb;
34     }
35   p_keyb = input1;
36   p_weigh = input2;
37 }
```

Fig. 9 A cheese scale measurement software with service mode.

all cycle-free influence paths. This way, less simplification with Lem. 3 is done and we obtain

$$\begin{aligned}&\Diamond((mode = 0) \wedge (p\_keyb = sk0) \wedge \Diamond((mode = 1) \wedge (p\_keyb = sk1) \wedge \\ &\Diamond((mode = 2) \wedge (p\_keyb = sk2) \wedge \Diamond((mode = 3) \wedge (p\_keyb = sk3) \wedge \\ &\Diamond((mode = 4) \wedge (p\_keyb = sk4) \wedge \Diamond((mode = 5) \wedge \\ &\qquad ((p\_keyb = 43) \vee (p\_keyb = 45) \wedge \Diamond(p\_weigh > 0)))))))\end{aligned} \tag{13}$$

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 and that neither (12) nor (13) are sufficient conditions for the influence to occur.

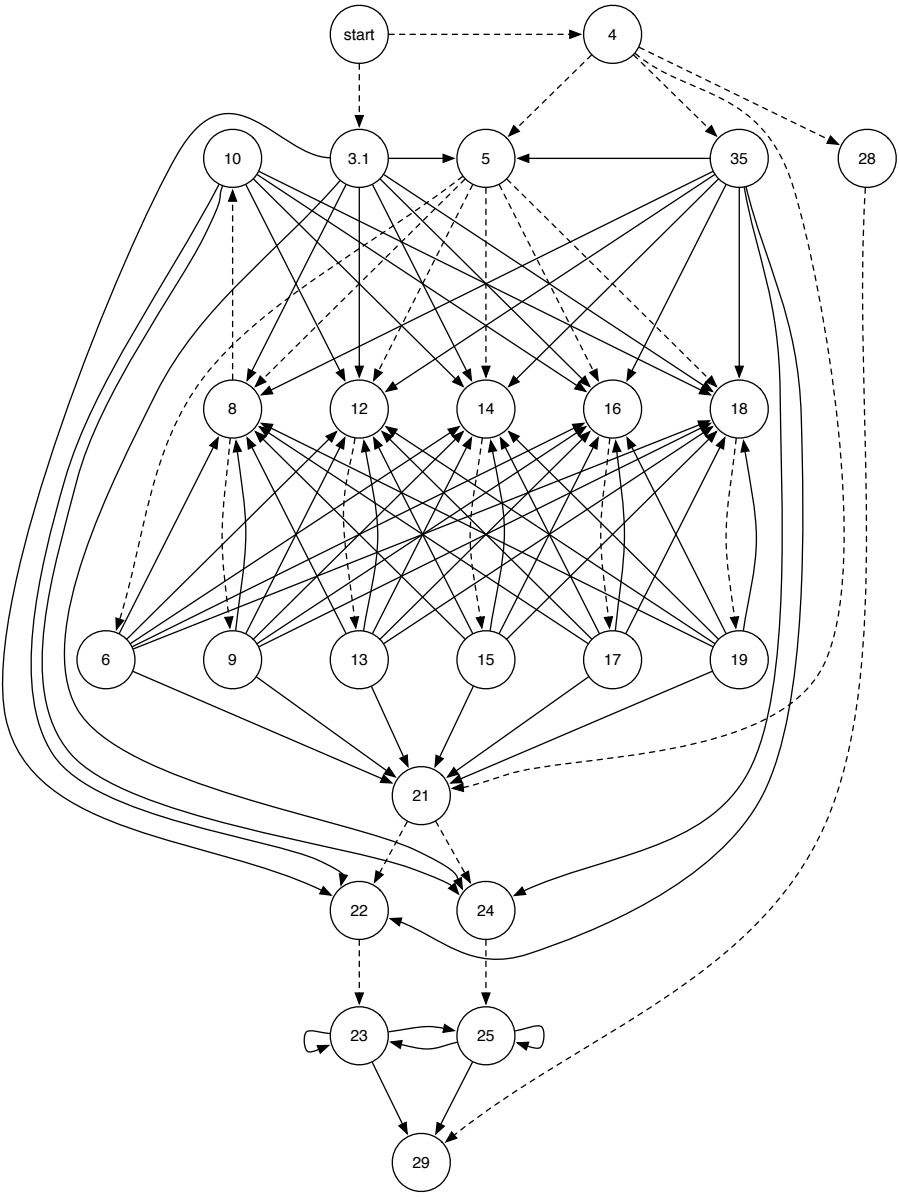

Fig. 10 Chop between `p_keyb` in ll. 3, 10, 26, 32, 35 and `u_kg` in l. 29 for the program in Fig. 9, which has been closed under control dependence.

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 whether a path condition is satisfiable without tool support. Boolean path conditions are therefore fed to a constraint solver which simplifies them as far as possible and maybe 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: 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.

However, this transformation is by no means trivial. We must combine slicing with program abstraction (Corbett et al. 2000; Dwyer et al. 2001) to compute an abstract interpretation of the program as a model for the model checker. Moreover, the model must keep track of SSA variables that occur in the path condition to check. 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" (Holzmann 2003)), 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 (Cimatti et al. 1999)), we have to input the negated LTL path condition. Note that the theory of symbolic model checking (McMillan 1992) 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.

Let us now return to the example of Sec. 4. 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. In order not to run into difficulties due to the state explosion problem, we have reduced the range of `int` variables in the following way:

- Six symbolic values for `p_keyb` and `input1` that represent the keys with key codes 13, 27, 43, 45, 65, 66 and one for subsuming all other key codes
- Three values for the weight sensor `p_weigh` and `input2`: 0, 1, 2
- Five values for `mode`: 0 to 5
- `kal_kg` ranges from -5 to 5 plus an additional "unknown" which corresponds to the top element in the abstract interpretation lattice.

These restrictions are conservative approximations because static analysis, in particular interval-based analysis, yields that either a variable always remains in their restricted range or, as in the case of `p_keyb`, an additional value is introduced to model all values; similarly for `kal_kg`. Reading from the input ports returns an arbitrary value in the range of the input port.

NuSMV generates a trace that directly enters service mode and changes the calibration factor, the keycode sequence starting with 65, 43, 66, 45, 13, 66, other, 45, other. Then, it puts some weight on the scale. The overall output path including a

loop at the end contains 34 states. 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. The model found by SPIN consists of 2053 steps. 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 (Sabelfeld and Myers 2003) for an overview. While type systems are a fast approach, they usually lack precision. For example, most type systems incorrectly 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 (Darvas et al. 2005) 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. (Hong et al. 2003) also use temporal formulae for static program analysis: They use the CFG to construct CTL formulae that express a condition for data flow 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 or 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. While this is perfectly acceptable for their application, it is unable to find all potential security leaks.

In (Ammons et al. 2002), 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 being 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 cannot guarantee that no security leaks creep in or are missed.

Xie and Chou (Xie and Chou 2002) 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 (Ball and Rajamani 2001; Fischer et al. 2005; Dhurjati et al. 2006) have become popular. However, directly formulating precise IFC conditions in these terms is not easy: The SLAM project (Ball and Rajamani 2002) provides a general path-sensitive data flow analysis (Ball and Raja-

mani 2001) which operates on boolean programs abstracted from C programs. As a consequence, exploiting arithmetic to prove a path unfeasible is not possible.

Fischer et al. (Fischer et al. 2005) from the BLAST project propose data flow analysis with path predicates: Their merge operation does not join data flow 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 (Dhurjati et al. 2006; Das et al. 2002) 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 (Hampapuram et al. 2005). 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 and future work

We have seen that temporal path conditions provide "precise" time-dependent information about the specific circumstances of an information flow in a program. By transforming the program into a compact model that preserves the state sequence semantics, using e.g. slicing and program abstraction (Corbett et al. 2000; Dwyer et al. 2001), we can use model checkers such as SPIN (Holzmann 2003) or NuSMV (Cimatti et al. 1999) to compute a specific input or state sequence for 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 (Lochbihler 2006), which have been left out in the present article because arrays introduce much additional technical complexity.

The current paper describes only the theoretical foundations for temporal path conditions. While a temporal path condition for a single path in the PDG can be efficiently computed, dependence between two arbitrary statements s and t reduces to single PDG paths only for very small and simple programs. The presented approach – up to optimizations for prefixes and suffixed – enumerates all cycle-free paths in the chop for s and t, of which exponentially many can exists. In theory, boolean path conditions suffer from the same problem, however, by analysing and decomposing the chop, the exponential blow-up can be avoided (Snelting et al. 2006). Similar techniques must be developped in an efficient implementation for temporal path conditions, which we are still working on. Equally, other program constructs like references or procedures and methods must be added to temporal path conditions.

We have focused on IFC as one particular application of temporal path conditions, but we are confident that they can equally be beneficial to other areas in program analysis that deal with dependences in programs: There are already similar approaches to automated test data generation (Hong et al. 2003), that generate sufficient (but not necessary) conditions for dependence paths. Further, almost every slicing algorithm might benefit from path conditions to eliminate spurious transitive dependences, in particular if non-scalar variables are involved. For example, if a (transitive) dependence requires a minimum number of iterations of some loop for being manifest, then temporal path conditions are able to express this. If the loop is guaranteed to be executed less times, the temporal path condition is never satisfied and the transitive dependence no longer needs to be considered. This may even lead to some kind of bootstrapping: In precise points-to analyses, temporal path conditions, for which points-to information must already have been computed, may be used to remove some unnecessary elements from points-to sets such that the PDG and consequently path conditions become more precise, giving rise to an iterative refinement algorithm. Note that these potential applications only use temporal path conditions on single dependence paths, i.e. the overheads for computing a path condition for a chop do not apply here.

Temporal path conditions can similarily be used to shrink slices: If the path condition from entry to some PDG node in an imprecise slice is unsatisfiable, then this node (and all others that are part of the slice due to this node being in the slice) can be safely removed from the slice. For example, conditioned slicing (Canfora et al. 1998) computes a slice for a given set of program executions given by constraints on the input values. Such constraints, given as a boolean formula, only have to be conjunctively added to temporal path condition to determine whether some statement can be removed from the imprecise slice. Note that loop-carried dependences make this much harder for boolean path conditions.

Acknowledgements The authors would like to thank Christian Hammer and the anonymous reviewers for their helpful comments.

References

Ammons, G., Bodik, R., Larus, J. R.: Mining Specifications. *Symposium on Principles of Programming Languages*, 4–16 (2002)

Ball, T., Rajamani, S. K.: Bebop: A Path-sensitive Interprocedural Dataflow Engine. *Workshop on Program Analysis for Software Tools and Engineering*, 97–103. (2001)

Ball, T., Rajamani, S. K.: The SLAM Project: Debugging System Software via Static Analysis. *Symposium on Principles of Programming Languages*, 1–3 (2002)

Canfora, G., Cimitile A., De Lucia, A.: Conditioned program slicing. *Inf. and Softw. Technol.* 30, 595–607 (1998)

Cimatti, A., Clarke, E. M., Giunchiglia, F., Roveri, M.: NuSMV: A New Symbolic Model Verifier. In: *International Conference on Computer Aided Verification*, *Lect. Notes Comp. Sci.* 1633, 495–499 (1999)

Clarke, Jr, E. M., Grumberg, O., Peled, D. A.: *Model Checking*. The MIT Press (2000)

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: *International Conference on Software Engineering*, 439–448 (2000)

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13(4), 451–490 (1991)

Darvas, A., Hähnle, R., Sands, D.: A Theorem Proving Approach to Analysis of Secure Information Flow. In: *International Conference on Security in Pervasive Computing*, *Lect. Notes Comp. Sci.* 3450, 193–209 (2005)

Das, M., Lerner, S., Seigle, M.: ESP: Path-Sensitive Program Verification in Polynomial Time. *Prog. Lang. Des. Implement.*, pp. 57–68. (2002)

Dhurjati, D., Das, M., Yang, Y.: Path-Sensitive Dataflow Analysis with Iterative Refinement. In: *Static Analysis Symposium*, *Lect. Notes Comp. Sci.* 4134, 425–442 (2006)

Dwyer, M. B., Hatcliff, J., Joehanes, R., Laubach, S., Păsăreanu, C. S., Robby, Visser, W., Zheng, H.: Tool-Supported Program Abstraction for Finite-State Verification. In: *International Conference on Software Engineering*, 177–187 (2001)

Fischer, J., Jhala, R., Majumdar, R.: Joining Dataflow with Predicates *Found. Softw. Eng.*, 227–236 (2005)

Hammer, C., Krinke, J., Snelting, G.: Information Flow Control for Java Based on Path Conditions in Dependence Graphs. In: *International Symposium on Secure Software Engineering*, 87–96 (2006)

Hampapuram, H., Yang, Y., Das, M.: Symbolic Path Simulation in Path-Sensitive Dataflow Analysis. In: *Workshop on Program Analysis for Software Tools and Engineering*, 52-58 (2005)

Holzmann, G. J.: *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley (2003)

Hong, H. S., Cha, S. D., Lee, I., Sokolsky, O., Ural, H.: Data Flow Testing as Model Checking. In: *International Conference on Software Engineering*, 232–242 (2003)

Krinke, J.: *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau (2003)

Lochbihler, A.: Temporal Path Conditions in Dependence Graphs. Master's thesis, Universität Passau (2006)

Lochbihler, A., Snelting, G.: On Temporal Path Conditions in Dependence Graphs. In: *International Working Conference on Source Code Analysis and Manipulation*, 49–58 (2007)

McMillan, K. L.: *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University (1992)

Ranganath, V. P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M. B.: A New Foundation for Control Dependence and Slicing for Modern Program Structures. *ACM Trans. Program. Lang. Syst.* 29(5), Article 27 (2007)

Robschink, T.: *Pfadbedingungen in Abhängigkeitsgraphen und ihre Anwendung in der Softwaresicherheitstechnik*. PhD thesis, Universität Passau (2005)

Robschink, T., Snelting, G.: Efficient Path Conditions in Dependence Graphs. In: *International Conference on Software Engineering*, 478–488 (2002)

Sabelfeld, A. Myers, A. C.: Language-Based Information-Flow Security. *IEEE J. Sel. Areas Commun.* 21(1), 5–19 (2003)

Snelting, G.: Combining Slicing and Constraint Solving for Validation of Measurement Software. In: *Static Analysis Symposium Lect. Notes Comp. Sci.* 1145, 332–348 (1996)

Snelting, G., Robschink, T., Krinke, J.: Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. *ACM Trans. Softw. Eng. Methodol.* 15(4), 410–457 (2006)

Tip, F.: A Survey of Program Slicing Techniques. *J. Program. Lang.* 3(3), 121–189 (1995)

Xie, Y., Chou, A.: Path Sensitive Program Analysis Using Boolean Satisfiability. Technical report, Stanford University (2002)