Fakultät für Mathematik und Informatik

Diplomarbeit

Temporal Path Conditions in Dependence Graphs

Andreas Lochbihler
lochbihl@fmi.uni-passau.de

Betreuer: Professor Dr.-Ing. G. Snelting

Zweitgutachter: Professor Dr. V. Weispfenning

8. Mai 2006

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As software is becoming more and more widespread and complex, ensuring correct software behaviour is more difficult than ever. Testing a software, i. e. running it on a collection of inputs and checking whether the observed output meets the requirements, can never prove its correctness and incurs nonnegligible costs in the software development process. In safety-critical applications, however, proofs for the software meeting its specification are often required. One aspect in this context is information flow control [Com99]. Software developers must make sure that agents, neither internal nor external, may not illicitly influence critical computations and that there are no leaks for confidential information to the environment. For example, the control software for fuel rods in a nuclear power plant had better not been influenced by external manipulations. Equally, in a professional measurement system, the data path from the sensor to the value display must not be manipulated. Regarding information leaks, in an information system for instance, some data must not be retrievable unless a correct password has been correctly entered.

Many formal security models and methods have been developed to control information flow in software systems, e. g. introducing security levels [BLP73] or partitioning a software system into security domains that may not influence each other [GM84]. Although there are sophisticated guidelines for developing safety-critical software systems (e. g. the common criteria [Com99]) that define applicable rules to ensure information flow control, we sometimes, in particular for existing software, have to analyze programs to see whether they meet the criteria such as security levels or noninterference partitioning: First, we need to determine which program parts can influence others. Second, we must decide whether any of these influences is a safety violation.

Obviously, the first part is undecidable in general, so we have to settle with approximations. Slicing [Wei84] determines which program parts can influence a given statement or be influenced by it, and which definitely do not so. Current implementations of slicing (e. g. ValSoft [KSR99, KS98, Kri03]) reduce the computation of slices to reachability analysis in the program or system dependence graph. [SRK] explains the relation between slicing and noninterference according to [GM84]. Today, slicing can be applied to medium-sized programs in reasonable time, but slices are pretty imprecise in practice and they do not give

any information on the conditions necessary for the influence happening between two statements. In [Sne96], Snelting proposed to compute a necessary Boolean condition for every path $\pi : s \rightarrow^* t$ between two given statements s and t in the system/program dependence graph that must hold whenever information can possibly flow along π. After having been simplified the constraint is given to a constraint solver which either proves the influence being impossible in the case the condition is unsatisfiable or returns a necessary condition over input variables for the influence to happen. When input values that satisfy the condition are given to the program, the influence from s to t will hopefully become visible, e. g. as a witness for the illegal influence in the safety-critical setting. Meanwhile, this idea has been developed further [Rob04, SRK] and is now applicable to real programming languages. However, solving for input variables is still a challenge [Sne05].

Unfortunately, Boolean path conditions are not able to capture temporal aspects of programs. Hence, loops and recursion are handled only rudimentarily which may ruin their precision. Equally, it is not possible to express that some (sub)condition must be satisfied only after some other condition. In this thesis, we explore how these issues can be addressed by using linear temporal logic (LTL) for a simple imperative programming language.

Model checking has become a very popular static program analysis tool over the last decade. Today, professional model checkers such as SPIN [Hol03] or NuSMV [CCGR99] are powerful enough to be applied to industrial-sized programs [HRV+03]. But they serve not only as simple verifiers of handcrafted specifications, they are applied in a large number of program analysis and verification approaches, e. g. for solving iterative data flow problems [SS98] and test data generation [HCL+03, GH99]. Similarly to constraint solvers being used for Boolean path conditions, we show how to use model checkers to find program traces that show precisely how information and what piece of information flows along the influence path. Moreover, we can also use model checking to compute a constraint over input variables which are necessary conditions for the influence happening.

Hence, this thesis' goals are:

- Rules for generating necessary LTL path conditions,
- Some ideas on how to simplify them,
- The link between model checking and temporal path conditions, and
- A number of small application examples.

This thesis is organized as follows: Chapter 2 gives an informal overview about path conditions and the issues that have to be resolved in their context. Both syntax and semantics of linear temporal logic (LTL) are presented in chapter 3. We introduce our imperative programming language IMP and a number of dependence graphs in chapter 4. In chapter 5, we revise Boolean path conditions. Chapter 6 contains generation and simplification rules for temporal path conditions for IMP programs and compares them to Boolean path conditions. The connection to model checking is established in chapter 7. We conclude with some examples in chapter 8 and a conclusion (chapter 9).

Chapter 2

Overview

This chapter gives an informal introduction to path conditions and outlines what issues arise in their context. We present the details more formally in the subsequent chapters. Here, we only explain what types of constraints temporal path conditions are composed of before we mention what can be done with LTL path conditions.

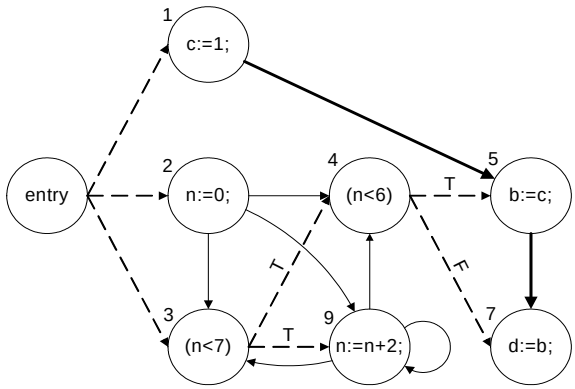
In general, path conditions are conditions for a single path or a set of paths in the program or system dependence graph, which is a combination of the control and the data dependence graph, such that the path condition is satisfiable if the path or one of them can be executed. In this thesis, we concentrate on imperative programs with structured control flow.

Example 1
Figure 2.1 shows an example program and its program dependence graph (PDG). If we want to know whether line 1 can influence line 7, we see that there is a path 1,5,7 in the PDG, i. e. slicing says: Yes, possibly there is such an influence.

A necessary condition for this path in the PDG is that lines 1, 5, and 7 must be executed in this order. Obviously, for line 5 being executed, $n < 6$ must hold. The same argument gives that $6 \leq n < 7$ must hold when line 7 is executed. However, we cannot satisfy both constraints simultaneously, even though – if we run the program – the lines are executed in the correct order.

Figure 2.1 An imperative example program and its PDG.

```
 1  c := 1;
 2  n := 0;
 3  while (n < 7) {
 4      if (n < 6) {
 5          b := c;
 6      } else {
 7          d := b;
 8      }
 9      n := n + 2;
10  }
```



This is because a single program variable can take multiple values in the course of execution. By transforming the program into static single assignment form (SSA) [CFR+91], in which every variable occurs at most once on the left-hand side of an assignment statement, we introduce new program variables such that we can always safely refer to a variable's value that has been assigned to it during the last execution of a specific assignment statement (cf. section 5.1). This way, we know that the execution condition, which is made up of control predicates surrounding the statement, always holds at the statement itself.[1] Unfortunately, this does not save us in the case when we want to combine execution conditions from different statements, i. e. from different execution points, as we do with the execution conditions for lines 5 and 7. The flow dependence edge between nodes 5 and 7 in the PDG is loop-carried, thus even the SSA variables may change their value while execution proceeds from line 5 to line 7.

Boolean path conditions (cf. chapter 5) can handle loop-carried data dependences only unsatisfactorily. In LTL path conditions, eventually $\Diamond$ and until $\mathcal{U}$ operators model that two conditions must hold at different points in execution. Additionally, these operators model the nodes' ordering on the path in the path condition.

As we allow non-scalar variables, i. e. variables that are composed of multiple subcomponents, we also want to make sure that if we write to a variable's component and later read from the variable, then it must be the same component both times for a flow of information taking place, e. g. for arrays, it must be the same array cell that is accessed (cf. section 6.2.2.3). Thus, we include additional constraints for data dependences to ensure this. Moreover, we also require that the array cell must not be overwritten by some other access to the array between the write and read accesses of interest.

Example 2
For the following program, we are interested whether line 1 influences line 3, i. e. whether the value of b stored in cell $2*i+1$ of array a reaches line 3 and is actually read there.

```
1  a[2 * i + 1] := b;
2  a[3 * j] := c;
3  if (a[k] == d) {
4      ...
5  }
```

Hence, we essentially add the data dependence constraint

$$(2 * i + 1 = k) \wedge (2 * i + 1 \neq 3 * j)$$

to the path condition, appropriately distributed over the until operators in it.

Similarly, for scalar variables, we have necessary constraints, so-called Φ constraints from data dependence (cf. section 5.3), to ensure that the SSA variant's

[1] Note that – without SSA form – if we moved line 9 in figure 2.1 before line 4, the constraint $n < 7$ would not necessarily hold in line 7.

value of the data dependence's source variable does reach the SSA variant read at its target, i. e. that the scalar variable has not been overwritten on the way.

Example 3
Consider the following program.

```
1  if (b) { x := 1; }
2  if (b) { x := 0; }
3  if (x < 1) { ... }
```

Here, we obviously have that line 1 (`x := 1`) cannot influence line 3, but from a control flow graph point of view without optimizations, there does exist a direct data dependence from line 1 to line 3. Example 20 (p. 78) deals with this program in more detail.

We also use the until operator to account for cycles in the PDG. Since we do want to generate path conditions for all paths between two statements, if there is a cycle on the way, the number of different paths is infinite, so we cannot simply take the disjunction over the path conditions for every single path (cf. section 6.3). Our aim is to restrict ourselves to cycle-free paths. For constraints with execution conditions only, this is pretty simple, but as we have data dependence conditions, too, things get more complex. While Boolean path conditions opted for ignoring cycles [Rob04], thereby reducing precision (cf. example 25 (p. 86)), we show that we can address this problem at little extra cost. For most types of cycles, we simply introduce an additional until operator, which – in many cases – disappears again during simplification, but there is one specific type of cycles that we must handle separately. Nevertheless, the number of paths still remains finite.

Apart from execution conditions from control dependence and conditions for data dependence edges, we also include constraints to model that a loop that precedes a statement must terminate before the statement can be executed, i. e. the negated loop predicate must hold after the loop (cf. section 6.2.1.1). (This does not mean that we consider a non-termination sensitive control dependence or influence definition.) Equally, when a data dependence exits a loop, we know that the affected loop must terminate in between (cf. section 6.2.2.1). Temporal logic is ideal to incorporate both types of constraints in path conditions, Boolean logic lacks the temporal dimension for that.

At last, we also introduce a new constraint on the statement level to ensure that an incoming data dependence can effect the evaluation at all (cf. section 6.2.3). For instance, if the predicate $(x > 5)||(c)$ is the target of a data dependence with respect to variable x, then c must be false so that x can influence the evaluation.

Once we have generated the LTL condition for a path or set of paths, we almost always must simplify the LTL formula to be understandable. Simplification can happen in two ways:

- There are congruences and implications that are valid for all LTL formulae and all models, i. e. we can simplify the formula without using additional knowledge from the analyzed program (cf. sections 3.1.3 and 3.2.4). For example, we have the law that $\Diamond$ is idempotent.

- Substantial simplification can also be achieved by statically analyzing at the program in the sense that some subformulae of the condition are not satisfiable over any model for the program and can therefore be eliminated (cf. section 6.4).

We will give some examples and ideas on that, but a comprehensive exposé on how to simplify a particular constraint best is beyond the scope of this thesis.

However, when looking carefully at the constraint does not give sufficient insight, model checking is another option to learn more. Since classic model checking can handle only finite state systems, we have designed our programming language to account for this, although all of the constraint types work, of course, for Turing-complete imperative programming languages, too.

In chapter 7, we show how to extract the correct finite state model from the program to be analyzed to which we can then apply either explicit state or symbolic model checking. If we are interested in finding any state sequence that satisfies the LTL formula and thus is a witness for the (illegal) influence, either approach works fine. In explicit state model checking, the LTL formula is also translated into a finite state machine which is then run in parallel with the program's machine. Any accepting infinite run contains a satisfying state sequence. Symbolic model checking converts the LTL formula into a μ-calculus formula with two nested fixpoint operators whose solution is the set of satisfying states. Explicit state model checking tends to produce longer witnesses than symbolic model checking does, but this can sometimes be very useful, too (cf. section 8.3). If, however, we are keen on solving the LTL formula for input variables of the program, symbolic model checking is clearly superior to explicit state model checking. Unfortunately, we have not been able to find a professional LTL model checker that is able to perform this type of model checking.

Thus, model checkers can be regarded as the equivalent to constraint solvers as is done with Boolean path condition. In contrast to quantifier elimination [Sne05], arbitrary program expressions in the LTL formula are no problem for model checkers as they work "inside" the program. They only have to be adapted to fit the model checker's modelling language. Moreover, due to the finite state assumption, every LTL path condition is decidable and – if space and time permit – solvable.

The snag with model checking is the state explosion problem: The number of states in the finite state representation of a program is exponential in the number of its variables. In the worst case, all of them have to be searched when proving that an LTL formula is not satisfiable over the model. Thus, in order to keep model checking feasible, we must use design abstractions when extracting the program model from the program. Although section 7.5 refers to some ideas on this, we are unable to address this issue in this thesis.

Chapter 3

Linear temporal logic (LTL)

Formulae in linear temporal logic (LTL) define predicates over infinite sequences of states. Propositional LTL formulae use the Boolean operators $\neg$, $\wedge$, $\vee$, and $\rightarrow$, and the four temporal operators *always* $\Box$, *eventually* $\Diamond$, *until* $\mathcal{U}$, and *next* $\bigcirc$ to connect propositional variables and the Boolean constants true and false. States are assignments to these variables. In LTL formulae over languages, atomic formulae over a language take the place of propositional variables and states assign values to the variables in the atomic formulae.

Pnueli was the first to propose temporal logic for analyzing distributed systems [Pnu77] and temporal semantics for reactive programs [Pnu81]. Since then, linear temporal logic has become very popular in specifying and verifying concurrent systems, in particular in connection with model checking.

In this chapter, we first introduce notation for well-known concepts before we define linear temporal logic over propositional variables, also known as propositional linear time logic. In the last part, we introduce the notions of typed variables, expressions, and atomic formulae, and present how they can be incorporated into linear time logic.

3.0 Preliminary definitions and notation

In this section, we give some well-known definitions and notations. Even though they are standard in literature, we provide them here to make this thesis self-contained. The notation defined here can also be found in the glossary at the end.

Definition 1 (Unions)
Let M be a set of sets. The **union** of M, denoted by $\bigcup M$, is

$$\bigcup M := \{\, a \mid \exists A \in M : a \in A \,\}.$$

If $(M_\lambda)_{\lambda\in\Lambda}$ is a family of sets, their union $\bigcup_{\lambda\in\Lambda} M_\lambda$ is defined as $\bigcup_{\lambda\in\Lambda} M_\lambda := \bigcup \{\, M_\lambda \mid \lambda \in \Lambda \,\}$. If Λ is finite, say $\Lambda = \{\, \lambda_1, \ldots, \lambda_n \,\}$, we write $M_{\lambda_1} \cup \ldots \cup M_{\lambda_n}$ for $\bigcup_{\lambda\in\Lambda} M_\lambda$.

The **disjoint union** $\bigcup\!\!\!\cdot\, M$ of M is the set

$$\bigcup\!\!\!\cdot\, M := \left\{ (a, A) \in \left(\bigcup M\right) \times M \;\middle|\; a \in A \right\}.$$

For $A \in M$ we identify $a \in A$ with $(a, A) \in \bigcup\!\!\!\cdot\, M$ and write $B \subseteq \bigcup\!\!\!\cdot\, M$ for $B \subseteq \bigcup M$. Equally, the disjoint union for $(M_\lambda)_{\lambda\in\Lambda}$ is the set $\bigcup\!\!\!\cdot_{\lambda\in\Lambda} M_\lambda := \bigcup_{\lambda\in\Lambda} M_\lambda \times \{ \lambda \}$. If $\Lambda = \{ \lambda_1, \ldots, \lambda_n \}$ is finite, we write $M_{\lambda_1} \cup\!\!\!\cdot \ldots \cup\!\!\!\cdot\, M_{\lambda_n}$ for $\bigcup\!\!\!\cdot_{\lambda\in\Lambda} M_\lambda$. For $\lambda \in \Lambda$ we identify $a \in M_\lambda$ with $(a, \lambda) \in \bigcup\!\!\!\cdot_{\mu\in\Lambda} M_\mu$ and write $B \subseteq \bigcup\!\!\!\cdot_{\mu\in\Lambda} M_\lambda$ for $B \subseteq \bigcup_{\lambda\in\Lambda} M_\lambda$.c For $a \in \bigcup\!\!\!\cdot_{\lambda\in\Lambda} M_\lambda$, we can always find exactly one $\lambda \in \Lambda$ such that a comes from M_λ, i. e. $a \in M \times \lambda$.

Definition 2 (Relations and functions)
Let X and Y be sets. Let $R \subseteq X \times Y$ be a binary relation and let $A \subseteq X$. The **image** $R(A)$ of A under relation R is $R(A) := \bigcup_{a\in A} \{ y \in Y \mid (a, y) \in R \}$. The **inverse relation** R^{-1} to R is defined as $R^{-1} := \{ (b, a) \in Y \times X \mid (a, b) \in R \}$.

Y^X denotes the set of all mappings $f : X \mapsto Y$. Let $f : X \mapsto Y$ be such a mapping and let $y \in Y$. The image $f(A)$ of A under f is $f(A) := \{ f(a) \mid a \in A \}$. The **inverse image** $f^{-1}(y)$ of y under f is $f^{-1}(y) := \{ x \in X \mid f(x) = a \}$. If f bijective, we also write $f^{-1}(y)$ for the image of y under the inverse function of f. The **partition on** X **induced by** f is $X/f = \left\{ f^{-1}(y) \;\middle|\; y \in f(X) \right\}$. The **restriction** $f|_A$ of f on A is $f|_A : A \mapsto Y$, $a \mapsto f(a)$ for $a \in A$.

Definition 3
The set of **natural numbers** $\mathbb{N}$ is $\mathbb{N} := \{ 0, 1, 2, \ldots \}$. The set of **integers** is denoted by $\mathbb{Z}$. The set of **32 bit integers** $\overline{\mathbb{Z}}$ is $\overline{\mathbb{Z}} := \left\{ -2^{31}, \ldots, 2^{31} - 1 \right\}$. The set of **Boolean truth values** $\mathbb{B}$ is $\mathbb{B} := \{ \mathtt{T}, \mathtt{F} \}$.

Let X be a set. The **power set** $\mathfrak{P}(X)$ of X is $\mathfrak{P}(X) := \{ Y \subseteq X \}$. X^+ denotes the set $\bigcup_{i=1}^{\infty} X^i$ of all **nonempty finite words or nonempty tuples** over the alphabet X. X^* denotes the set $X^+ \cup \{ \epsilon \}$ of all finite words (inclusively the **empty word** ϵ) or the set $X^+ \cup \{ () \}$ set of all tuples (including the **empty tuple** ()) over the alphabet X. X^ω denotes the set $X^{\mathbb{N}}$ of all **infinite words or infinite tuples** over alphabet X.

Definition 4 (Graphs and multigraphs)
A **graph** G is represented as a tuple $G = (V, E)$ where V is the set of **nodes** or **vertices** and the set of **edges** $E \subseteq V \times V$ is a binary relation on V. If G is a graph, we write $\mathrm{V}(G)$ for the set of nodes in G and $\mathrm{E}(G)$ for the set of edges in G. The set $\mathrm{pred}(v)$ of **predecessor nodes** for $v \in V$ is $\mathrm{pred}(v) := E^{-1}(\{ v \})$. The set $\mathrm{succ}(v)$ of **successor nodes** to $v \in V$ is $\mathrm{succ}(v) = E(\{ v \})$.

A **multigraph** G is represented by a quadruple $G = (V, E, \odot\!\!\succ, \prec\!\!\odot)$ where V is the set of nodes, E the set of edges, and $\odot\!\!\succ, \prec\!\!\odot : E \mapsto V$ are the source and target node mappings that assign each edge its source and target node. G is a graph iff $(\odot\!\!\succ, \prec\!\!\odot) : E \mapsto V \times V$, $e \mapsto (\odot\!\!\succ(e), \prec\!\!\odot(e))$, is injective. In this case, we identify $e \in E$ with $(\odot\!\!\succ(e), \prec\!\!\odot(e))$, i. e. $E \subseteq V \times V$ and write $G = (V, E)$ as shorthand. Conversely, every graph $G = (V, E)$ can be considered as a multigraph with $\odot\!\!\succ((v, w)) := v$ and $\prec\!\!\odot((v, w)) := w$ being implicitly defined for $(v, w) \in E$.

Let now $G = (V, E, \odot\!\!\succ, \prec\!\!\odot)$ denote a multigraph and let $W \subseteq V$. The **multigraph** $G|_W$ **generated by** W is $G|_W := (W, E', \odot\!\!\succ|_{E'}, \prec\!\!\odot|_{E'})$ where $E' := \odot\!\!\succ^{-1}(W) \cap \prec\!\!\odot^{-1}(W)$.

Let $G_1 = (V_1, E_1, \odot\!\!\succ_1, \prec\!\!\odot_1)$ and $G_2 = (V_2, E_2, \odot\!\!\succ_2, \prec\!\!\odot_2)$ be two multigraphs. G_1 is a **subgraph** of G_2, denoted by $G_1 \subseteq G_2$, iff $V_1 \subseteq V_2$, $E_1 \subseteq E_2$, $\odot\!\!\succ_1 = \odot\!\!\succ_2|_{E_1}$, and $\prec\!\!\odot_1 = \prec\!\!\odot_2|_{E_1}$.

Definition 5 (Paths in graphs)
Let $G = (V, E, \odot\!\!\succ, \prec\!\!\odot)$ be a multigraph. A **path** π in G is a finite sequence of edges $\pi := e_1, \ldots, e_n$ such that $\odot\!\!\succ(e_{i+1}) = \prec\!\!\odot(e_i)$. If $n = 0$, identify π with some $v \in V$. If G is a graph (i. e. $E \subseteq V \times V$), we identify π with the sequence of nodes $v_0, \ldots, v_n$ via the bijection $v_0, \ldots, v_n \leftrightarrow (v_0, v_1), \ldots, (v_{n-1}, v_n)$. The **start node** of π is $\odot\!\!\succ(\pi) := \odot\!\!\succ(e_1)$, the **target node** of π is $\prec\!\!\odot(\pi) := \prec\!\!\odot(e_n)$. We write $\rho : v \xrightarrow[G]{}\!\!\!\!\blacktriangleright^{*} w$ for a path ρ in G with $\odot\!\!\succ(\rho) = v$ and $\prec\!\!\odot(\rho) = w$.

Let now $\pi = e_1, \ldots, e_n$ be a path in G. We define the set of **path nodes** $\mathrm{V}(\pi)$ of π as $\mathrm{V}(\pi) := \{ \prec\!\!\odot(\pi) \} \cup \dot{\bigcup}_{i=1}^{n} \{ \odot\!\!\succ(e_i) \}$ and the set of **path edges** $\mathrm{E}(\pi)$ of π as $\mathrm{E}(\pi) := \dot{\bigcup}_{i=1}^{n} \{ e_i \}$.

A path $\rho = f_1, \ldots, f_m$ in G is a **subpath** of π iff there is a $i \in \mathbb{N}$, $i \leq n-m$ such that $f_{i+j} = e_j$ for $1 \leq j \leq m$. ρ is **contained** in π iff there exists a strictly isotone mapping $\mu : \{ 1, \ldots, m \} \mapsto \{ 1, \ldots, n \}$, i. e. $\mu(i) < \mu(j)$ for all $1 \leq i < j \leq m$, such that $f_j = e_{\mu(j)}$ for $1 \leq j \leq m$.

The **length** $|\pi|$ of π is the number of edges in π, i. e. $|\pi| = n$.

For $1 \leq i < n$, the **successor edge** $\mathrm{succ}_\pi(e_i)$ to e_i in π is e_{i+1}. For $1 < i \leq n$, the **predecessor edge** $\mathrm{pred}_\pi(e_i)$ to e_i in π is e_{i-1}. If it is clear from the context, we omit writing the subscript π.

We say π is

- **node disjoint** or **cycle free** iff $\odot\!\!\succ(e_i) \neq \odot\!\!\succ(e_j)$ for $i \neq j$ and $\odot\!\!\succ(e_i) \neq \prec\!\!\odot(e_n)$ for $1 \leq i, j \leq n$,
- **edge disjoint** iff $e_i \neq e_j$ for all $1 \leq i, j \leq n$,
- **closed** iff $\odot\!\!\succ(\pi) = \prec\!\!\odot(\pi)$,
- **cycle disjoint** iff $\rho \neq \rho'$ for all closed subpaths $\rho := e_i, \ldots, e_{i+j}$ and $\rho' := e_{i'}, \ldots, e_{i'+j}$ of π with $i \neq i'$ $(1 \leq i, i' < n,\ 1 \leq j \leq |\pi|)$.

Let $v, w \in \mathrm{V}(\pi)$ be two nodes in π. We write $v \prec_\pi w$ iff v comes before w in π, i. e. let e_i be the edge responsible for $v \in \mathrm{V}(\pi)$ and e_j the one for w, then we demand that $i < j$, or $i = j = n$ and $v = \odot\!\!\succ(e_n)$ and $w = \prec\!\!\odot(e_n)$. We write $v \preceq_\pi w$ iff v comes no later than w in π, i. e. $v = w$ or $v \prec_\pi w$.

If there exists a path $\rho : v \xrightarrow[G]{}\!\!\!\!\blacktriangleright^{*} w$ in G from v to w, we say v **reaches** w and w is **reachable** from v, denoted by $v \overset{G}{\rightsquigarrow} w$. Let $W \subseteq V$ be a set of nodes. $W^+_{E'} := \bigcup_{u \in W} \{ u' \in V \mid \exists \pi : u \xrightarrow[G]{}\!\!\!\!\blacktriangleright^{*} u' : \mathrm{E}(\pi) \subseteq E' \}$ denotes the set of nodes reachable from W in G via edges E'.

Definition 6 (Strongly connected components)
Let $G = (V, E, \dot{\succ}, \dot{\rightarrow})$ be a multigraph. A set of nodes $W \subseteq V$ is **strongly connected** iff $u \overset{G|_W}{\rightsquigarrow} v$ for all $u, v \in W$. If every $W' \subseteq V$ with $W' \supsetneqq W$ is not strongly connected, W is a **strongly connected component** of G.

3.1 LTL formulae over propositional variables

Usually, LTL formulae contain only propositional variables that can only take the Boolean truth values `T` and `F`. Syntactically, these variables and the Boolean constants true and false are combined with the four temporal operators $\Box$, $\Diamond$, $\mathcal{U}$, and $\bigcirc$, and with the usual Boolean connectives to form LTL formulae. Some dialects of LTL also provide past temporal operators (see e. g. [MP91]), but here, we consider only the future segment.

After having defined the syntax of LTL formulae over propositional variables, we give a semantics for them over infinite state sequences, and introduce the concepts of equivalence and implication along with some ideas on how to simplify such formulae.

3.1.1 Syntax

Let $\mathcal{W}$ be the countable set of propositional variables. Later, we will use these variables to represent atomic formulae (cf. section 3.2.1). We usually write Hebrew letters (א, ב, ג, ד, ...) to denote propositional variables.

We now define the syntax of LTL formulae over propositional variables. The alphabet for them is given by $\mathcal{Z}_{\mathsf{LTL}} := \mathcal{O}_{\mathsf{LTL}} \cup \mathcal{W} \cup \{\,(,),\equiv\,\} \cup \{\,,\,\}$ where $\mathcal{O}_{\mathsf{LTL}} := \{\,\neg, \wedge, \vee, \rightarrow, \bigcirc, \Box, \Diamond, \mathcal{U}\,\}$ is the set of LTL operators.

Definition 7 (LTL formulae)
The set LTL of **LTL formulae (over propositional variables)** is inductively defined as follows:

1. Every propositional variable is an LTL formula: $\mathcal{W} \subseteq \mathsf{LTL}$.
2. The Boolean constants true and false are LTL formulae: $\mathrm{true}, \mathrm{false} \in \mathsf{LTL}$.
3. If $\theta, \theta' \in \mathsf{LTL}$ are LTL formulae, so are $\neg(\theta), (\theta) \wedge (\theta'), (\theta) \vee (\theta'), (\theta) \rightarrow (\theta') \in \mathsf{LTL}$.
4. If $\theta, \theta' \in \mathsf{LTL}$ are LTL formulae, so are $\bigcirc(\theta), \Box(\theta), \Diamond(\theta), (\theta)\,\mathcal{U}\,(\theta') \in \mathsf{LTL}$.

If an LTL formula θ is constructed with rules 1, 2, and 3 only, θ is called a **state formula**. We write SF for the set of all state formulae.

In order to save brackets in LTL formulae, we define precedence rules for the operators: We order them as:

$$\neg \quad \bigcirc \quad \Box \quad \Diamond \quad \mathcal{U} \quad \wedge \quad \vee \quad \rightarrow$$

where $\neg$ binds most strongly and $\rightarrow$ least strongly. For instance, we write $\Box(\text{true} \rightarrow \beth) \wedge \Diamond \beth$ for $(\Box((\text{true}) \rightarrow (\beth))) \wedge (\Diamond(\beth))$. Moreover, for the binary operators $\wedge$, $\vee$, and $\rightarrow$, we allow sequences of them without brackets and consider it to be bracketed right-associatively. For example, $\aleph \rightarrow \beth \rightarrow \daleth$ stands for $\aleph \rightarrow (\beth \rightarrow \daleth)$.

Next, we inductively define the set of propositional variables $\mathcal{W}(\theta)$ of an LTL formula $\theta \in \mathsf{LTL}$ over propositional variables:

- If $\theta \in \mathcal{W}$ is a propositional variable, then $\mathcal{W}(\theta) := \{ \theta \}$.
- If $\theta \in \{ \text{true}, \text{false} \}$ is a Boolean variable, then $\mathcal{W}(\theta) := \emptyset$.
- If θ is of the form $\circ(\eta)$ where $\eta \in \mathsf{LTL}$ and $\circ \in \{ \neg, \Box, \Diamond, \bigcirc \}$, then $\mathcal{W}(\theta) := \mathcal{W}(\eta)$.
- If θ is of the form $(\eta) \circ (\eta')$ where $\eta, \eta' \in \mathsf{LTL}$ and $\circ \in \{ \wedge, \vee, \rightarrow, \mathcal{U} \}$, then $\mathcal{W}(\theta) := \mathcal{W}(\eta) \cup \mathcal{W}(\eta')$.

In this thesis, we make heavy use of substitution. We define two types thereof: We want to

- substitute an LTL formula for every occurrence of a propositional variable, or
- identify a subformula of an LTL formula and replace it by another LTL formula.

Definition 8 (LTL formula substitution for propositional variables)
Let $\theta \in \mathsf{LTL}$ be an LTL formula over propositional variables, and $W \subseteq \mathcal{W}$ be a nonempty, finite set of propositional variables, say $W = \{ \beth_1, \ldots, \beth_n \}$, and let $\theta_1, \ldots, \theta_n \in \mathsf{LTL}$. Then, $\theta[\theta_1/\beth_1, \ldots, \theta_n/\beth_n]$ is the LTL formula that we define recursively by:

- If $\theta \in \{ \text{true}, \text{false} \}$ is a Boolean constant, then
$$\theta[\theta_1/\beth_1, \ldots, \theta_n/\beth_n] := \theta.$$
- If $\theta \in W$ is a propositional variable in W, say $\theta = \beth_i$, then
$$\theta[\theta_1/\beth_1, \ldots, \theta_n/\beth_n] := \theta_i.$$
- If $\theta \in \mathcal{W} - W$ is a propositional variable not in W, then
$$\theta[\theta_1/\beth_1, \ldots, \theta_n/\beth_n] := \theta.$$
- If θ is of the form $\circ(\eta)$ where $\circ \in \{ \neg, \bigcirc, \Box, \Diamond \}$ and $\eta \in \mathsf{LTL}$, then
$$\theta[\theta_1/\beth_1, \ldots, \theta_n/\beth_n] := \circ(\eta[\theta_1/\beth_1, \ldots, \theta_n/\beth_n]).$$
- If θ is of the form $(\eta) \circ (\eta')$ where $\circ \in \{ \wedge, \vee, \rightarrow, \mathcal{U} \}$ and $\eta, \eta' \in \mathsf{LTL}$, then
$$\theta[\theta_1/\beth_1, \ldots, \theta_n/\beth_n] := (\eta[\theta_1/\beth_1, \ldots, \theta_n/\beth_n]) \circ (\eta'[\theta_1/\beth_1, \ldots, \theta_n/\beth_n]).$$

If we want to simultaneously substitute the same LTL formula $\vartheta \in \mathsf{LTL}$ for all propositional variables of a finite set $W \subseteq \mathcal{W}$, say $W = \{ \beth_1, \ldots, \beth_n \}$, in $\theta \in \mathsf{LTL}$, we write as shorthand:

$$\theta[\vartheta/W] := \theta[\vartheta/\beth_1, \ldots, \vartheta/\beth_n]$$

Definition 9 (Substituting formulae for formulae)
Let $\theta \in \mathsf{LTL}$ be an LTL formula and let $\beth \in \mathcal{W}(\theta)$ be a propositional variable which occurs exactly once in θ. Let $\circ \in \{ \wedge, \vee, \rightarrow, \mathcal{U} \}$ be a binary LTL operator. Then, $\theta \langle\!\langle \circ, \beth \rangle\!\rangle$ denotes the pair of LTL formulae which are the operands for the innermost $\circ$ operator either of which contains $\beth$:

- If θ does not contain a subformula of the form $(\eta) \circ (\eta')$ such that $\beth \in \mathcal{W}(\eta) \cup \mathcal{W}(\eta')$, then, there is no such pair and we set $\theta \langle\!\langle \circ, \beth \rangle\!\rangle := (\epsilon, \epsilon)$
- If θ is of the form $(\eta) \circ (\eta')$ for some $\eta, \eta' \in \mathsf{LTL}$ and there do not exist $\zeta, \zeta' \in \mathsf{LTL}$ with $\beth \in \mathcal{W}(\zeta) \cup \mathcal{W}(\zeta')$ such that $(\zeta) \circ (\zeta')$ is a subformula of either η or η', then $\theta \langle\!\langle \circ, \beth \rangle\!\rangle := (\eta, \eta')$.
- If θ is not of the form $(\eta) \circ (\eta')$ for some $\eta, \eta' \in \mathsf{LTL}$, but θ contains a subformula ϑ of the form $(\zeta) \circ (\zeta')$ for some $\zeta, \zeta' \in \mathsf{LTL}$ where $\beth \in \mathcal{W}(\zeta) \cup \mathcal{W}(\zeta')$, then $\theta \langle\!\langle \circ, \beth \rangle\!\rangle := \vartheta \langle\!\langle \circ, \beth \rangle\!\rangle$.[2]

Let $\kappa \in \mathsf{LTL}$ be an LTL formula and let $\daleth \in \mathcal{W} - \mathcal{W}(\theta)$ be a propositional variable. Suppose $(\epsilon, \epsilon) \neq \theta \langle\!\langle \circ, \beth \rangle\!\rangle =: (\eta, \eta')$. There exists exactly one LTL formula $\vartheta \in \mathsf{LTL}$ such that $\vartheta[(\eta) \circ (\eta')/\daleth] = \theta$. If we replace $(\eta) \circ (\eta')$ by κ in θ, we obtain the LTL formula $\vartheta[\kappa/\daleth]$, which we denote by $\theta \langle\!\langle \circ, \beth \,|\, \kappa \rangle\!\rangle$. If $\theta \langle\!\langle \circ, \beth \rangle\!\rangle = (\epsilon, \epsilon)$, we set $\theta \langle\!\langle \circ, \beth \,|\, \kappa \rangle\!\rangle := \kappa$.

Example 4
Let us consider the formula

$$\theta := \aleph \wedge \beth \wedge \text{false}\, \mathcal{U}\, (\daleth \vee (\gimel \wedge \aleph)).$$

Then, $\theta \langle\!\langle \mathcal{U}, \gimel \rangle\!\rangle = (\text{false}\,, \daleth \vee (\gimel \wedge \aleph))$ and

$$\theta \langle\!\langle \mathcal{U}, \daleth \,|\, \daleth\, \mathcal{U}\, (\gimel \wedge \beth) \rangle\!\rangle = \aleph \wedge \beth \wedge \daleth\, \mathcal{U}\, (\gimel \wedge \beth).$$

If we take the operator $\wedge$ and the propositional variable $\beth$, we obtain $\theta \langle\!\langle \wedge, \beth \rangle\!\rangle = (\beth, \text{false}\, \mathcal{U}\, (\daleth \vee (\gimel \wedge \aleph)))$ because we agreed upon $\wedge$ being right-associative. Hence,

$$\theta \langle\!\langle \wedge, \beth \,|\, \text{true} \rangle\!\rangle = \aleph \wedge \text{true}$$

[2] Note that, although ϑ may not be uniquely determined, $\theta \langle\!\langle \circ, \beth \rangle\!\rangle$ is nevertheless well-defined: We show this by induction on the length of θ. If θ has length 1, then $\theta \langle\!\langle \circ, \beth \rangle\!\rangle = (\epsilon, \epsilon)$. For the other two cases, $\theta \langle\!\langle \circ, \beth \rangle\!\rangle$ is always well-defined. So suppose $\vartheta = (\zeta) \circ (\zeta')$ and $\vartheta' = (\beta) \circ (\beta')$ are subformulae of θ such that $\beth \in \mathcal{W}(\vartheta)$ and $\beth \in \mathcal{W}(\vartheta')$. Since $\beth$ occurs exactly once in θ, ϑ is a subformula of ϑ' or vice versa. Without loss of generality, let ϑ be a subformula of ϑ'. Then, either $\vartheta = \vartheta'$ or ϑ is shorter than ϑ'. In the latter case, $\vartheta \langle\!\langle \circ, \beth \rangle\!\rangle$ is well-defined by induction hypothesis and by definition of $\vartheta' \langle\!\langle \circ, \beth \rangle\!\rangle$: $\vartheta' \langle\!\langle \circ, \beth \rangle\!\rangle = \vartheta \langle\!\langle \circ, \beth \rangle\!\rangle$.

This way, we can replace subformulae without spelling them out exactly. Since the propositional variable to identify the subformula of interest occurs exactly once, the subformula to be replaced is uniquely determined. However, not every subformula can be replaced in a single step in this way. Consider, for instance

$$\vartheta := \text{false}\,\mathcal{U}\,(\text{true}\,\mathcal{U}\,(\aleph\,\mathcal{U}\,\beth))$$

Then, there is no direct way to replace $\text{true}\,\mathcal{U}\,(\aleph\,\mathcal{U}\,\beth)$ in this setting. Nevertheless, we could replace $\aleph\,\mathcal{U}\,\beth$ by a new propositional variable $\gimel$ first and then replace $\text{true}\,\mathcal{U}\,\gimel$: $(\vartheta\,\langle\!\langle \mathcal{U}, \aleph \mid \gimel \rangle\!\rangle)\,\langle\!\langle \mathcal{U}, \gimel \mid \aleph \rangle\!\rangle = \text{false}\,\mathcal{U}\,\aleph$.

3.1.2 Semantics

LTL formulae are used to express temporal properties. While in Boolean propositional logic, a formula's truth value is defined with respect to an assignment to the propositional variables in it, this is not sufficient for the temporal case. The central notion here is an infinite sequence of states, i. e. of assignments to propositional variables. State sequences that satisfy an LTL formula are its models.

Definition 10 (State and state sequence)
Let $W \subseteq \mathcal{W}$ be a finite set of propositional variables. A **state** ξ over W is a mapping $\xi : W \mapsto \mathbb{B}$ that assigns a Boolean truth value to each propositional variable in W. We write $\mathcal{S}_W$ for the set $\mathbb{B}^W$ of all states over W.

A **state sequence** over W is an infinite sequence $\Xi = (\xi_i)_{i\in\mathbb{N}}$ of states over W. We consider the index $i \in \mathbb{N}$ as time scale. We write $\mathcal{M}_W$ for the set $(\mathcal{S}_W)^{\mathbb{N}} = \left(\mathbb{B}^W\right)^{\mathbb{N}}$ of all state sequences over W.

Definition 11 (Model)
Let $W \in \mathcal{W}$ be a finite set of propositional variables and let $\theta \in \mathsf{LTL}$ be an LTL formula with $\mathcal{W}(\theta) \subseteq W$. Let $\Xi = (\xi_i)_{i\in\mathbb{N}} \in \mathcal{M}_W$ be a state sequence over W. We inductively define the notion of θ **holding in** Ξ at time i, denoted by $(\Xi, i) \vDash \theta$:

- If $\theta \in W$ is a propositional variable, then $(\Xi, i) \vDash \theta$ iff $\xi_i(\theta) = \mathtt{T}$.
- If $\theta \in \{\,\text{true}\,, \text{false}\,\}$ is a Boolean constant, then $(\Xi, i) \vDash \theta$ iff $\theta = \text{true}$.
- If $\theta = \neg(\eta)$ for some $\eta \in \mathsf{LTL}$: $(\Xi, i) \vDash \theta$ iff not $(\Xi, i) \vDash \eta$.
- If $\theta = (\eta) \wedge (\eta')$ for some $\eta, \eta' \in \mathsf{LTL}$, then $(\Xi, i) \vDash \theta$ iff $(\Xi, i) \vDash \eta$ and $(\Xi, i) \vDash \eta'$.
- If $\theta = (\eta) \vee (\eta')$ for some $\eta, \eta' \in \mathsf{LTL}$, then $(\Xi, i) \vDash \theta$ iff $(\Xi, i) \vDash \eta$ or $(\Xi, i) \vDash \eta'$.
- If $\theta = (\eta) \rightarrow (\eta')$ for some $\eta, \eta' \in \mathsf{LTL}$, then $(\Xi, i) \vDash \theta$ iff not $(\Xi, i) \vDash \eta$ or $(\Xi, i) \vDash \eta'$.
- If $\theta = \bigcirc(\eta)$ for some $\eta \in \mathsf{LTL}$, then $(\Xi, i) \vDash \theta$ iff $(\Xi, i+1) \vDash \eta$.
- If $\theta = \Box(\eta)$ for some $\eta \in \mathsf{LTL}$, then $(\Xi, i) \vDash \theta$ iff $(\Xi, i+k) \vDash \eta$ for all $k \in \mathbb{N}$.

- If $\theta = \Diamond(\eta)$ for some $\eta \in \mathsf{LTL}$, then $(\Xi, i) \vDash \theta$ iff $(\Xi, i+k) \vDash \eta$ for some $k \in \mathbb{N}$.
- If $\theta = (\eta)\ \mathcal{U}\ (\eta')$ for some $\eta, \eta' \in \mathsf{LTL}$, then $(\Xi, i) \vDash \theta$ iff $(\Xi, i+k) \vDash \eta'$ for some $k \in \mathbb{N}$ and $(\Xi, i+j) \vDash \eta$ for all $j \in \mathbb{N}$ with $0 \leq j < k$.

We say Ξ **satisfies** (is a **model** for) θ, denoted by $\Xi \vDash \theta$, iff $(\Xi, 0) \vDash \theta$.

Lemma 1 (LTL formula substitution)
Let $\theta, \eta \in \mathsf{LTL}$ be LTL formulae with $\mathcal{W}(\theta) \cup \mathcal{W}(\eta) \subseteq W$, let $\beth \in W$ be a propositional variable and let $\Xi = (\xi_i)_{i\in\mathbb{N}}, \Psi = (\psi_i)_{i\in\mathbb{N}} \in \mathcal{M}_W$ be state sequences such that $\xi_i(\beth) = \mathtt{T}$ iff $(\Psi, i) \vDash \eta$ for all $i \in \mathbb{N}$, and such that $\xi_i(\gimel) = \psi_i(\gimel)$ for all $\gimel \in W - \{\beth\}$ and all $i \in \mathbb{N}$. Then $\Xi \vDash \theta$ iff $\Psi \vDash \theta[\eta/\beth]$.

Proof. We show $(\Xi, i) \vDash \theta$ iff $(\Psi, i) \vDash \theta[\eta/\beth]$ (for $i \in \mathbb{N}$) by induction on the structure of θ:

- If $\theta = \beth$ is the propositional variable $\beth$ itself, then $\theta[\eta/\beth] = \eta$ and $(\Xi, i) \vDash \beth$ iff $\xi_i(\beth) = \mathtt{T}$ iff $(\Psi, i) \vDash \eta$.
- If $\theta \in \mathcal{W} - \{\beth\}$, say $\theta = \aleph$, then $\Xi(\theta[\eta/\beth]) = \Xi(\aleph) = \Psi(\aleph)$ and $(\Xi, i) \vDash \aleph$ iff $(\Psi, i) \vDash \aleph$.
- If θ is of the form $\neg(\theta')$ for some $\theta' \in \mathsf{LTL}$, then $\theta[\eta/\beth] = \neg(\theta'[\eta/\beth])$ and $(\Xi, i) \vDash \theta$ iff not $(\Xi, i) \vDash \theta'$ iff (by induction hypothesis) not $(\Psi, i) \vDash \theta'[\eta/\beth]$ iff $(\Psi, i) \vDash \theta[\eta/\beth]$.
- If θ is of the form $(\theta_1) \circ (\theta_2)$ where $\theta_1, \theta_2 \in \mathsf{LTL}$ and $\circ \in \{\wedge, \vee, \rightarrow\}$, then $\theta[\eta/\beth] = (\theta_1[\eta/\beth]) \circ (\theta_2[\eta/\beth])$ and $(\Xi, i) \vDash \theta$ iff $(\Xi, i) \vDash \theta_1$ and/or/implies $(\Xi, i) \vDash \theta_2$ iff (by induction hypothesis) $(\Psi, i) \vDash \theta_1[\eta/\beth]$ and/or/implies $(\Psi, i) \vDash \theta_2[\eta/\beth]$ iff $(\Psi, j) \vDash \theta[\eta/\beth]$.
- If θ is of the form $\bigcirc(\theta')$ $(\Diamond(\theta'))$, then $\theta[\eta/\beth] = \bigcirc(\theta'[\eta/\beth])$ $(\theta[\eta/\beth] = \Diamond(\theta'[\eta/\beth]))$ and $(\Xi, i) \vDash \theta$ iff $(\Xi, i+1) \vDash \theta'$ $((\Xi, i+k) \vDash \theta'$ for some $k \in \mathbb{N})$ iff (by induction hypothesis) $(\Psi, i+1) \vDash \theta'[\eta/\beth]$ $((\Psi, i+k) \vDash \theta'[\eta/\beth])$ iff $(\Psi, i) \vDash \theta[\eta/\beth]$.
- If θ is of the form $\Box(\theta')$, then $\theta[\eta/\beth] = \Box(\theta'[\eta/\beth])$ and $(\Xi, i) \vDash \theta$ iff $(\Xi, i+k) \vDash \theta'$ for all $k \in \mathbb{N}$ iff (by induction hypothesis) $(\Psi, i+k) \vDash \theta'[\eta/\beth]$ for all $k \in \mathbb{N}$ iff $(\Psi, i) \vDash \theta[\eta/\beth]$.
- If θ is of the form $(\theta_1)\ \mathcal{U}\ (\theta_2)$, then $\theta[\eta/\beth] = (\theta_1[\eta/\beth])\ \mathcal{U}\ (\theta_2[\eta/\beth])$ and $(\Xi, i) \vDash \theta$ iff $(\Xi, i+k) \vDash \theta_2$ for some $k \in \mathbb{N}$ and $(\Xi, i+j) \vDash \theta_1$ for all $0 \leq j < k$ iff (by induction hypothesis) $(\Psi, i+k) \vDash \theta_2[\eta/\beth]$ and $(\Psi, i+j) \vDash \theta_1[\eta/\beth]$ for $0 \leq j < k$ iff $(\Psi, i) \vDash \theta[\eta/\beth]$.

□

3.1.3 Implication and equivalence

Our interest is focused on whether an LTL formula holds for a state sequence or whether there exists a model at all rather than on the specific syntax of the formula. Hence, we say two formulae are equivalent if they have the same models:

Definition 12 (Equivalence)
Let $W \subseteq \mathcal{W}$ be a finite set of propositional variables. We say $\theta, \eta \in \mathsf{LTL}$ with $\mathcal{W}(\theta) \cup \mathcal{W}(\eta) \subseteq W$ are **equivalent** iff for every $\Xi \in \mathcal{M}_W$: $\Xi \vDash \theta$ iff $\Xi \vDash \eta$. We denote this by $\theta \nleftrightarrow \eta$.

Note that $\theta \nleftrightarrow \eta$ iff $\Xi \vDash ((\theta \rightarrow \eta) \wedge (\eta \rightarrow \theta))$ for every $\Xi \in \mathcal{M}_W$. Later, we will use equivalences to simplify LTL formulae. Sometimes, however, equivalences can not deliver us the desired extent of simplification. In this case, we may want to weaken the LTL formula using implications:

Definition 13 (Implication)
Let $W \subseteq \mathcal{W}$ be a finite set of propositional variables. For $\theta, \eta \in \mathsf{LTL}$ with $\mathcal{W}(\theta) \cup \mathcal{W}(\eta) \subseteq W$, we say θ **implies** η iff $\Xi \vDash (\theta \rightarrow \eta)$ for every $\Xi \in \mathcal{M}_W$. We denote this by $\theta \nrightarrow \eta$.

Lemma 2 (Implication and equivalence patterns)
Let $W \subseteq \mathcal{W}$ be a finite set of propositional variables. Let $\theta, \eta \in \mathsf{LTL}$ such that $\mathcal{W}(\theta) \cup \mathcal{W}(\eta) \subseteq W$ and $\theta \nrightarrow \eta$. Then $\theta[\vartheta/\beth] \nrightarrow \eta[\vartheta/\beth]$ for every $\beth \in W$ and every $\vartheta \in \mathsf{LTL}$ with $\mathcal{W}(\vartheta) \subseteq W$. If $\theta \nleftrightarrow \eta$, then also $\theta[\vartheta/\beth] \nleftrightarrow \eta[\vartheta/\beth]$.

Proof. Without loss of generality, we may assume $\beth \in W$. For sake of contradiction, assume that $\theta[\vartheta/\beth] \nrightarrow \eta[\vartheta/\beth]$ does not hold, i. e. there exists a state sequence $\Xi = (\xi_i)_{i \in \mathbb{N}} \in \mathcal{M}_W$ such that $\Xi \vDash \theta[\vartheta/\beth]$, but not $\Xi \vDash \eta[\vartheta/\beth]$. There exists a sequence $\Psi = (\psi_i)_{i \in \mathbb{N}} \in \mathcal{M}_W$ such that $\psi_i(\beth) = \mathtt{T}$ iff $(\Xi, i) \vDash \vartheta$ and $\psi_i|_{W - \{\beth\}} = \xi_i|_{W - \{\beth\}}$ for all $i \in \mathbb{N}$. Then, by lemma 1, $\Psi \vDash \theta$, but not $\Psi \vDash \eta$, a contradiction to $\theta \nrightarrow \eta$. $\theta[\vartheta/\beth] \nleftrightarrow \eta[\vartheta/\beth]$ if $\theta \nleftrightarrow \eta$ follows from $\theta \nleftrightarrow \eta$ iff $\theta \nrightarrow \eta$ and $\eta \nrightarrow \theta$. □

If we know that two formulae are equivalent, we can always substitute one with the other. For example, consider the two formulae $\theta := \Box \neg \beth$ and $\eta := \neg \Diamond \beth$. It follows directly from the definition of $\Box$ and $\Diamond$ that $\theta \nleftrightarrow \eta$. Hence, whenever we encounter an LTL formula of type $\Box \neg \vartheta$ where $\vartheta \in \mathsf{LTL}$ we can instead evaluate $\neg \Diamond \vartheta$. By default, we can do this kind of simplification for whole formulae only. The next lemma shows that we may also perform this inside other LTL formulae:

Lemma 3 (Congruence)
Let $W \subseteq \mathcal{W}$ be a finite set of propositional variables. Let $\theta, \eta, \vartheta \in \mathsf{LTL}$ such that $\mathcal{W}(\theta) \cup \mathcal{W}(\eta) \cup \mathcal{W}(\vartheta) \subseteq W$ and $\theta \nleftrightarrow \eta$. Then, $\vartheta[\theta/\beth] \nleftrightarrow \vartheta[\eta/\beth]$ for every $\beth \in W$.

Proof. This follows directly from the inductive definition of $\vDash$ and the observation that for every $\Xi = (\xi_i)_{i \in \mathbb{N}} \in \mathcal{M}_W$ and every $j \in \mathbb{N}$, $\Psi = (\xi_{i+j})_{i \in \mathbb{N}} \in \mathcal{M}_W$. □

	θ	η
Associativity	$(א \wedge ב) \wedge ד$	$א \wedge (ב \wedge ד)$
	$(א \vee ב) \vee ד$	$א \vee (ב \vee ד)$
Commutativity	$א \wedge ב$	$ב \wedge א$
	$א \vee ב$	$ב \vee א$
Absorption	$א \wedge (א \vee ב)$	$א$
	$א \vee (א \wedge ב)$	$א$
	$\Box \Diamond \Box א$	$\Diamond \Box א$
	$\Diamond \Box \Diamond א$	$\Box \Diamond א$
Distributivity	$(א \wedge ב) \vee ד$	$(א \vee ד) \wedge (ב \vee ד)$
	$(א \vee ב) \wedge ד$	$(א \wedge ד) \vee (ב \wedge ד)$
	$\bigcirc \neg א$	$\neg \bigcirc א$
	$\bigcirc (ב \vee א)$	$\bigcirc ב \vee \bigcirc א$
	$\Box (א \wedge ב)$	$\Box ב \wedge \Box א$
	$\Diamond (א \vee ב)$	$\Diamond ב \vee \Diamond א$
	$א \,\mathcal{U}\, (ב \vee ד)$	$(א \,\mathcal{U}\, ב) \vee (א \,\mathcal{U}\, ד)$
	$(א \wedge ב) \,\mathcal{U}\, ד$	$(א \,\mathcal{U}\, ד) \wedge (ב \,\mathcal{U}\, ד)$
	$\Box \Diamond (א \vee ב)$	$\Box \Diamond א \vee \Box \Diamond ב$
	$\Diamond \Box (א \wedge ב)$	$\Diamond \Box א \wedge \Diamond \Box ב$
Idempotence	$א \wedge א$	$א$
	$א \vee א$	$א$
	$\Box \Box א$	$\Box א$
	$\Diamond \Diamond א$	$\Diamond א$
	$א \,\mathcal{U}\, (א \,\mathcal{U}\, ב)$	$א \,\mathcal{U}\, ב$
	$(א \,\mathcal{U}\, ב) \,\mathcal{U}\, ב$	$א \,\mathcal{U}\, ב$
true, false	$א \wedge \text{true}$	$א$
	$א \vee \text{true}$	true
	$א \wedge \text{false}$	false
	$א \vee \text{false}$	$א$
Complement	$\neg(\neg א)$	$א$
	$א \wedge \neg א$	false
	$א \vee \neg א$	true
De Morgan	$\neg(א \wedge ב)$	$(\neg א) \vee (\neg ב)$
	$\neg(א \vee ב)$	$(\neg א) \wedge (\neg ב)$
Implication	$א \rightarrow ב$	$\neg א \vee ב$
	$א \rightarrow א$	true
	$א \rightarrow \text{true}$	true
	$א \rightarrow \text{false}$	$\neg א$
Expansion	$\Box א$	$א \wedge \bigcirc \Box א$
	$\Diamond א$	$א \vee \bigcirc \Diamond א$
	$א \,\mathcal{U}\, ב$	$ב \vee (א \wedge \bigcirc (א \,\mathcal{U}\, ב))$
Temporal operators	$\Box א$	$\neg(\Diamond \neg א)$
	$\Diamond א$	$\text{true} \,\mathcal{U}\, א$
Until	$א \wedge א \,\mathcal{U}\, (א \wedge ב)$	$א \,\mathcal{U}\, (א \wedge ב)$
	$א \,\mathcal{U}\, (א \wedge א \,\mathcal{U}\, ב)$	$ב \wedge א \,\mathcal{U}\, ב$
	$\Diamond (א \,\mathcal{U}\, ב)$	$\Diamond ב$
	$א \,\mathcal{U}\, \Diamond ב$	$\Diamond ב$

Table 3.1: List of LTL formulae θ and η such that for every row $\theta \nrightarrow \eta$. Some equivalences are taken from [MP91].

	θ	η
Boolean implications	$א$	true
	$א \wedge ב$	$ב$
	$א$	$א \vee ב$
Temporal implications	$\Box א$	$א$
	$א$	$ב \mathcal{U} א$
	$א \mathcal{U} ב$	$א \vee ב$
	$ב$	$א \mathcal{U} ב$
	$א \mathcal{U} ב$	$\Diamond ב$
	$א \mathcal{U} ב \vee ג \mathcal{U} ב$	$(א \vee ג) \mathcal{U} ב$
	$א \mathcal{U} (ב \mathcal{U} ג)$	$(א \vee ב) \mathcal{U} ג$

Table 3.2: List of LTL formulae θ and η such that for every row $\theta \rightarrowtail \eta$.

Table 3.1 lists pairs (θ, η) of LTL formulae each of which represents an equivalence $\theta \nleftrightarrow \eta$. Apart from the usual equivalences for the Boolean operators $\wedge$, $\vee$, $\rightarrow$, and $\neg$ which also hold in LTL, we also provide a number of equivalences for temporal operators. Under the heading "Temporal operators", we note that we can express the operators $\Box$ and $\Diamond$ in terms of the $\mathcal{U}$ operator, so for every LTL formula, there exists an equivalent one that contains neither $\Box$ nor $\Diamond$ operators. Table 3.2 contains some LTL formulae θ, η such that $\theta \rightarrowtail \eta$. Obviously, these lists can not be exhaustive. We only give a selection of those we think to apply in later chapters.

3.2 LTL formulae over structures

Usually, not all variables in a program are Boolean, we also have integers, arrays and the like. As we use LTL formulae to characterize program executions, we want to express constraints over program variables of all types. Hence, we define the concepts of atomic formulae, which take the position of propositional variables in LTL formulae, and structures over which we interpret atomic formulae. Regarding simplification of LTL formulae, we, of course, want to exploit constraints between atomic formulae to simplify the LTL formulae further than this would be possible with equivalences as defined in definition 12 (p. 15). Thus, we refine the concepts of equivalence and implication, in order to be able to simplify formulae like $(x < y) \wedge (y < x)$ to false.

We start by informally defining atomic formulae and related notions. Then, we incorporate these syntactically into LTL formulae and give a semantics for them. At last, we show how to define equivalence and congruence, implication and entailment for LTL formulae over structures and how they fit in with the former notions thereof.

3.2.1 Typed variables, expressions and atomic formulae

We now briefly sketch what we name typed variables, expressions and atomic formulae. For those who are interested in a formal definition, [SW05] contains a formal definition of these concepts. The extension to multi-sorted languages and structures, which we use here, is straightforward.

On the syntactic level, we assume that we have a set of types $\mathcal{T}$, a countable set of typed variable identifiers $\mathcal{V}$ and a typing function $(\!|\cdot|\!) : \mathcal{V} \mapsto \mathcal{T}$ such that $\mathcal{V} \cap \mathcal{W} = \emptyset$, some function operators (denoted by the set $\mathcal{F}$) and some predicate operators (denoted by the set $\mathcal{R}$) each of which has fixed arity with fixed parameter (and return) types where $\mathcal{F} \cap \mathcal{R} = \emptyset$. We write $\mathcal{F}_i$ ($\mathcal{R}_i$) for function (predicate) operators with arity i, $i \in \mathbb{N}$. We group function and predicate operators and their type and arity specification under the concept of a **language** $\mathcal{L}$.

Then, we call some words over the alphabet $\mathcal{Z} = \mathcal{F} \cup \mathcal{R} \cup \mathcal{V} \cup \{\,(,),\equiv\,\} \cup \{\,,\,\}$ expression or atomic formula. More precisely, the set of **expressions** $\mathfrak{E} \subseteq \mathcal{Z}^*$ is the smallest set of words over $\mathcal{Z}$ such that $\mathcal{V} \cup \mathcal{F}_0 \subseteq \mathfrak{E}$ and that is closed under the application of function operators where type constraints are respected. The set of variables $\mathcal{V}(e)$ of an expression $e \in \mathfrak{E}$ is the set of all variable identifier letters in e. We extend the typing function $(\!|\cdot|\!)$ to expressions in the obvious way. The set of **atomic formulae** $\mathfrak{A} \subseteq \mathcal{Z}^*$ is the smallest set of words over $\mathcal{Z}$ such that if $e_1, e_2 \in \mathfrak{E}$ with $(\!|e_1|\!) = (\!|e_2|\!)$ then $e_1 \equiv e_2 \in \mathfrak{A}$ and such that if $r \in \mathcal{R}_n$ is a predicate operator of arity $n \in \mathbb{N}$, then $r(e_1, \ldots, e_n) \in \mathfrak{A}$ for all $e_1, \ldots, e_n \in \mathfrak{E}$ of appropriate type.

If we want to substitute an expression $e' \in \mathfrak{E}$ for a typed variable identifier $x \in \mathcal{V}$ in another expression $e \in \mathfrak{E}$ where $(\!|e'|\!) = (\!|x|\!)$, we write $e[e'/x]$ for the expression we obtain by substituting e' for all occurrences of x in e. We write $e[e_1/x_1, \ldots, e_n/x_n]$ for the expression we obtain by substituting simultaneously e_i for x_i, $1 \leq i \leq n$ if $(\!|e_i|\!) = (\!|x_i|\!)$ for all $1 \leq i \leq n$. We extend this notation to atomic formulae, too.

Regarding the semantics of expressions and atomic formulae over a language $\mathcal{L}$, we assume that we have structure $\mathbf{A} = (\underline{A}, \tau_{\mathcal{T}}, \iota_{\mathcal{F}}, \iota_{\mathcal{R}})$ with a universe $\underline{A}$ that contains all possible values for expressions. A typing function $\tau_{\mathcal{T}}$ assigns to each element of the universe its type, the interpretation functions $\iota_{\mathcal{F}}$ and $\iota_{\mathcal{R}}$ assign each function and predicate operator a function on an appropriate subset of the universe. Interpreted function operators map to elements of the universe whose type matches its return type, predicate operators to $\mathbb{B}$. For an expression $e \in \mathfrak{E}$, we write $(\!|e|\!)^{\mathbf{A}}$ as shorthand for $\tau_{\mathcal{T}}^{-1}((\!|e|\!))$.

When we interpret expressions $e \in \mathfrak{E}$ or atomic formulae $a \in \mathfrak{A}$ as $(\!|e|\!)^{\mathbf{A}}$-valued and $\mathbb{B}$-valued functions over $\mathbf{A}$, we always assume that we have an extension, i. e. a total order $\sqsubseteq$ on some finite set $V \subseteq \mathcal{V}$ that contains $\mathcal{V}(e)$ and $\mathcal{V}(a)$ respectively, to fix the variable ordering. The interpretations $e^{\mathbf{A}}$ and $a^{\mathbf{A}}$ are defined inductively: If $e \in \mathcal{V}$, then $e \in V$ and $e^{\mathbf{A}}$ projects $(a_1, \ldots, a_n)$ to the value at the position at which x occurs in the extension. If $e = f(e_1, \ldots, e_m)$ for some function operator $f \in \mathcal{F}_m$, we set

$$e^{\mathbf{A}} := (a_1, \ldots, a_n) \mapsto \iota_{\mathcal{F}}(f)(e_1^{\mathbf{A}}(a_1, \ldots, a_n), \ldots, e_m^{\mathbf{A}}(a_1, \ldots, a_n))$$

for $a_j \in (\!|x_j|\!)^{\mathbf{A}}$, $(1 \leq j \leq n)$ where we assume that $e_1, \ldots, e_m$ have the same extension as e. For atomic formulae, we have that $(e_1 \equiv e_2)^{\mathbf{A}}(a_1, \ldots, a_n)$ yields $\mathtt{T}$ iff $e_1^{\mathbf{A}}(a_1, \ldots, a_n) = e_2^{\mathbf{A}}(a_1, \ldots, a_n)$. For atomic formulae with predicate operators, we set

$$(r(e_1, \ldots, e_m))^{\mathbf{A}} := (a_1, \ldots, a_n) \mapsto \iota_{\mathcal{R}}(r)(e_1^{\mathbf{A}}(a_1, \ldots, a_n), \ldots, e_m^{\mathbf{A}}(a_1, \ldots, a_n))$$

for $a_j \in (\!|x_j|\!)^{\mathbf{A}}$, $(1 \leq j \leq n)$. Note that interpretation and substitution are compatible, i. e.

$$e[e_1/x_1, \ldots, e_n/x_n]^{\mathbf{A}} = e^{\mathbf{A}} \circ \left(e_1^{\mathbf{A}}, \ldots, e_n^{\mathbf{A}}\right)$$

For the rest of this section, let $\mathcal{L}$ be a fixed language and $\mathbf{A}$ a structure over $\mathcal{L}$.

3.2.2 Syntax

As before, we first define the syntax:

Definition 14 (LTL formulae over $\mathcal{L}$)
The set of LTL formulae over $\mathcal{L}$, denoted by $\mathsf{LTL}^{\mathcal{L}}$, or – if it is clear from the context – simply by LTL, is inductively defined as follows:

1. Every atomic formula is an LTL formula over $\mathcal{L}$: $\mathfrak{A} \subseteq \mathsf{LTL}^{\mathcal{L}}$.
2. Every propositional variable is an LTL formula over $\mathcal{L}$: $\mathcal{W} \subseteq \mathsf{LTL}^{\mathcal{L}}$.
3. The Boolean constants true and false are LTL formulae over $\mathcal{L}$: $\mathrm{true}, \mathrm{false} \in \mathsf{LTL}^{\mathcal{L}}$.
4. If θ is an LTL formula over $\mathcal{L}$, so are $\neg(\theta)$, $\Box(\theta)$, $\bigcirc(\theta)$, and $\Diamond(\theta)$.
5. If θ and θ' are LTL formulae over $\mathcal{L}$, so are $(\theta) \wedge (\theta')$, $(\theta) \vee (\theta')$, $(\theta) \rightarrow (\theta')$, and $(\theta)\, \mathcal{U}\, (\theta')$.

As before, if $\theta \in \mathsf{LTL}^{\mathcal{L}}$ does not contain temporal operators, we say θ is a **state formula** over $\mathcal{L}$. We write $\mathsf{SF}^{\mathcal{L}}$ for the set of all state formulae over $\mathcal{L}$.

Again, we introduce the same precedence rules to save brackets: We order the operators as:

$$\neg \quad \bigcirc \quad \Box \quad \Diamond \quad \mathcal{U} \quad \wedge \quad \vee \quad \rightarrow$$

where $\neg$ binds most strongly and $\rightarrow$ least strongly. As for the variables we naturally extend the definition of typed and propositional variables in a formula θ to LTL formulae over $\mathcal{L}$:

- If $\theta \in \mathfrak{A}$ is an atomic formula, then we take the old definition for $\mathcal{V}(\theta)$ and set $\mathcal{W}(\theta) := \emptyset$.[3]

[3] Although we treat atomic formulae like propositional variables, we do not include them in the set $\mathcal{W}(\theta)$.

- If $\theta \in \{ \text{true}, \text{false} \} \cup \mathcal{W}$, then we take the old definition for $\mathcal{W}(\theta)$ and set $\mathcal{V}(\theta) := \emptyset$.
- If θ is of the form $\circ(\eta)$ where $\circ \in \{ \neg, \bigcirc, \Box, \Diamond \}$ and $\eta \in \mathsf{LTL}^{\mathcal{L}}$, then $\mathcal{V}(\theta) := \mathcal{V}(\eta)$ and $\mathcal{W}(\theta) := \mathcal{W}(\eta)$.
- If θ is of the form $(\eta) \circ (\eta')$ where $\circ \in \{ \wedge, \vee, \rightarrow, \mathcal{U} \}$ and $\eta, \eta' \in \mathsf{LTL}^{\mathcal{L}}$, then $\mathcal{V}(\theta) := \mathcal{V}(\eta) \cup \mathcal{V}(\eta')$ and $\mathcal{W}(\theta) := \mathcal{W}(\eta) \cup \mathcal{W}(\eta')$.

Next, we extend the concept of term substitution to arbitrary LTL formulae over $\mathcal{L}$. As with expressions and atomic formulae, we want to be able to substitute expressions of appropriate type for typed variables inside an LTL formula over $\mathcal{L}$. Again, if types match, we write $\theta[e_1/x_1, \ldots, e_n/x_n]$ for the LTL formula over $\mathcal{L}$ we obtain by simultaneously substituting every atomic formula a in θ by $a[e_1/x_1, \ldots, e_n/x_n]$.

Equally, we extend LTL formula substitution for propositional variables (cf. definition 8 (p. 11)) to LTL formulae over $\mathcal{L}$: We add one more item to definition 8 (p. 11): If $\theta \in \mathfrak{A}$ is an atomic formula, we set $\theta[\theta_1/\beth_1, \ldots, \theta_n/\beth_n] := \theta$. We extend definition 9 (p. 12) to LTL formulae over $\mathcal{L}$ in the same way. If $\theta \in \mathsf{LTL}^{\mathcal{L}}$, we write $\overline{\theta}$ as an abbreviation for $\theta[\text{true}/\mathcal{W}(\theta)]$.

3.2.3 Semantics

Next, we define the semantics of an LTL formula over $\mathcal{L}$. First, we need to adapt the notion of a state. Let $X \subseteq \mathcal{V}$ be a finite set of typed variables. Let $\mathfrak{E}_X$ denote the set of all expressions with variables from X, i. e. $\mathfrak{E}_X := \{ e \in \mathfrak{E} \mid \mathcal{V}(e) \subseteq X \}$, and let $\mathfrak{A}_X$ denote the set of all atomic formulae with variables from X, i. e. $\mathfrak{A}_X := \{ a \in \mathfrak{A} \mid \mathcal{V}(a) \subseteq X \}$. For $W \subseteq \mathcal{W}$ we write $\mathsf{LTL}^{\mathcal{L}}_{X,W}$ for the set $\left\{ \theta \in \mathsf{LTL}^{\mathcal{L}} \;\middle|\; \mathcal{V}(\theta) \subseteq X \wedge \mathcal{W}(\theta) \subseteq W \right\}$ of LTL formulae with typed variables from X and propositional variables from W. The set $\mathsf{LTL}^{\mathcal{L}}_X := \mathsf{LTL}^{\mathcal{L}}_{X,\emptyset}$ of LTL formulae over X denotes the set of all LTL formulae over $\mathcal{L}$ with typed variables from X and no propositional variables. We use the analogous notation for sets of state formulae, $\mathsf{SF}^{\mathcal{L}}_{X,W}$ and $\mathsf{SF}^{\mathcal{L}}_X$. Without loss of generality we assume in this section that $X = \{ x_1, \ldots, x_n \}$ with the order $x_i \sqsubseteq x_j$ iff $1 \leq i \leq j \leq n$.

Definition 15 (States and state sequences)
Let X be a finite set of typed variables and W a finite set of propositional variables. An **extended state** ξ over X and W is a mapping $\xi : X \cup W \mapsto \underline{\mathrm{A}} \cup \mathbb{B}$ that assigns each typed variable $x \in X$ a value of its appropriate domain, i. e. $\xi(x) \in (\!|x|\!)^{\mathbf{A}}$, and to each propositional variable $\beth \in W$ a Boolean value, i. e. $\xi(\beth) \in \mathbb{B}$. Let $\mathcal{S}^{\mathbf{A}}_{X,W}$ denote the set of all extended states over X and W for structure $\mathbf{A}$.

For $e \in \mathfrak{E}_X$ we write $\xi(e)$ for $e^{\mathbf{A}}(\xi(x_1), \ldots, \xi(x_n))$, and for $a \in \mathfrak{A}_X$ we write $\xi(a)$ for $a^{\mathbf{A}}(\xi(x_1), \ldots, \xi(x_n))$.

A **sequence of extended states** Ξ over X and W is an infinite sequence $\Xi = (\xi_i)_{i \in \mathbb{N}}$ of extended states ξ_i over X and W. We denote the set of all sequences of extended states over X and W by $\mathcal{M}^{\mathbf{A}}_{X,W}$.

If $W = \emptyset$, we say ξ is a **state** over X and Ξ is a **state sequence** over X. We write $\mathcal{S}^{\mathbf{A}}_X$ for $\mathcal{S}^{\mathbf{A}}_{X,\emptyset}$ and $\mathcal{M}^{\mathbf{A}}_X$ for $\mathcal{M}^{\mathbf{A}}_{X,\emptyset}$.

Definition 16 (Model)
Let $\Xi = (\xi_i)_{i\in\mathbb{N}} \in \mathcal{M}^{\mathbf{A}}_{X,W}$ be a sequence of extended states over X and W and let $\theta \in \mathsf{LTL}^{\mathcal{L}}_{X,W}$ be an LTL formula over $\mathcal{L}$ with propositional variables from W and typed variables from X. Let $\mathfrak{A}(\theta) \subseteq \mathfrak{A}$ be the set of all atomic formulae that are contained in θ. Let $\Xi' = (\xi'_i)_{i\in\mathbb{N}}$ be the sequence defined by $\xi'_i : \mathfrak{A}(\theta) \cup W \mapsto \mathbb{B}, a \mapsto \xi_i(a)$. We treat atomic formulae in $\mathfrak{A}(\theta)$ like propositional variables. Then, Ξ' is a state sequence in the sense of LTL formulae over propositional variables and we set $(\Xi, i) \vDash \theta$ iff $(\Xi', i) \vDash \theta$ and $\Xi \vDash \theta$ iff $\Xi' \vDash \theta$. If $\Xi \vDash \theta$, we say Ξ is an **extended model** for θ. If $W = \emptyset$, we omit "extended."

3.2.4 Equivalence, congruence, and entailment

We want to extend the notions of equivalence and implication to LTL formulae over $\mathcal{L}$. Therefore, we generalize the definitions as follows:

Definition 17 (Equivalence, congruence, implication, and entailment)
Let $X \subseteq \mathcal{V}$ be a finite set of typed variables and W a finite set of propositional variables. Let $\theta, \eta \in \mathsf{LTL}^{\mathcal{L}}_{X,W}$ be two LTL formulae over $\mathcal{L}$ with typed variables from X and propositional variables from W. Let $M \subseteq \mathcal{M}^{\mathbf{A}}_X$ be a nonempty set of state sequences and set

$$M^W := \left\{ (\xi_i)_{i\in\mathbb{N}} \in \mathcal{M}^{\mathbf{A}}_{X,W} \;\middle|\; (\xi_i|_X)_{i\in\mathbb{N}} \in M \right\}.$$

We say

- θ **implies** η with respect to M, denoted by $\theta \overset{M}{\rightarrowtail} \eta$ iff for all $\Xi \in M^W$: $\Xi \vDash (\theta \rightarrow \eta)$,
- θ **entails** η with respect to M, denoted by $\theta \overset{M}{\Longrightarrow} \eta$ iff for all $\Xi \in M^W$: $\Xi \vDash \Box(\theta \rightarrow \eta)$,
- θ is **equivalent** to η with respect to M, denoted by $\theta \overset{M}{\nrightarrow} \eta$ iff $\theta \overset{M}{\rightarrowtail} \eta$ and $\eta \overset{M}{\rightarrowtail} \theta$,
- θ is **congruent** to η with respect to M, denoted by $\theta \overset{M}{\Longleftrightarrow} \eta$ iff $\theta \overset{M}{\Longrightarrow} \eta$ and $\eta \overset{M}{\Longrightarrow} \theta$.

If $M = \mathcal{M}^{\mathbf{A}}_X$, we omit writing the M on the relation symbol: $\rightarrowtail$, $\Longrightarrow$, $\nrightarrow$, and $\Longleftrightarrow$.

Lemma 4 (LTL formulae over propositional variables and over languages)
Let $W \subseteq \mathcal{W}$ be finite, say $W = \{ \beth_1, \ldots, \beth_n \}$ and $\theta, \eta \in \mathsf{LTL}$ be two LTL formulae over propositional variables such that $\mathcal{W}(\theta) \cup \mathcal{W}(\eta) \subseteq W$ and $\theta \rightarrowtail \eta$. Let $X \subseteq \mathcal{V}$ be a finite set of typed variables and for $1 \leq i \leq m \leq n$ let $\vartheta_i \in \mathsf{LTL}^{\mathcal{L}}_{X,W}$. Then $\theta[\vartheta_1/\beth_1, \ldots, \vartheta_m/\beth_m], \eta[\vartheta_1/\beth_1, \ldots, \vartheta_m/\beth_m] \in \mathsf{LTL}^{\mathcal{L}}_{X,\mathcal{W}}$ and

$$\theta[\vartheta_1/\beth_1, \ldots, \vartheta_m/\beth_m] \Longrightarrow \eta[\vartheta_1/\beth_1, \ldots, \vartheta_m/\beth_m].$$

Proof. Let $\Xi \in \mathcal{M}^{\mathbf{A}}_{X,W}$ and let $i \in \mathbb{N}$. Since $\theta \rightarrowtail \eta$, we have $(\Xi, i) \vDash \theta \rightarrow \eta$. From the inductive definition of LTL formulae and their semantics, it follows immediately that $(\Xi, i) \vDash \theta[\vartheta_1/\beth_1, \ldots, \vartheta_m/\beth_m] \rightarrowtail \eta[\vartheta_1/\beth_1, \ldots, \vartheta_m/\beth_m]$. □

Definition 18 (Positive and negative occurrence)
Let $\theta \in \mathsf{LTL}^{\mathcal{L}}$ be an LTL formula over $\mathcal{L}$ and let $\beth \in \mathcal{W}(\theta)$. If $\beth$ occurs in an even number of (implicit or explicit) negations, $\beth$ is said to occur **positively**. If $\beth$ occurs in an odd number of (implicit or explicit) negations, $\beth$ is said to occur **negatively**. $\beth$ occurring inside the antecedent of an implication (i. e. inside η in $\eta \rightarrow \eta'$) counts as an implicit negation. The $\neg$ operator generates explicit negations.

Lemma 5 (Entailment and Congruence)
Let $W \subseteq \mathcal{W}$ be a finite set of propositional variables, say $W = \{\beth_1, \ldots, \beth_n\}$. Let $X \subseteq \mathcal{V}$ be a finite set of typed variables, $M \subseteq \mathcal{M}^{\mathbf{A}}_X$ nonempty, and for $1 \leq i \leq m \leq n$ let $\theta_i, \eta_i \in \mathsf{LTL}^{\mathcal{L}}_{X,\mathcal{W}}$ such that $\theta_i \overset{M}{\rightarrowtail} \eta_i$. Let $\vartheta \in \mathsf{LTL}^{\mathcal{L}}_{X,W}$.

- *If $\beth_i$ does not occur negatively in ϑ for all $1 \leq i \leq m$, then*

$$\vartheta[\theta_1/\beth_1, \ldots, \theta_m/\beth_m] \overset{M}{\rightarrowtail} \vartheta[\eta_1/\beth_1, \ldots, \eta_m/\beth_m].$$

- *If $\beth_i$ does not occur positively in ϑ for every $1 \leq i \leq m$, then*

$$\vartheta[\eta_1/\beth_1, \ldots, \eta_m/\beth_m] \overset{M}{\rightarrowtail} \vartheta[\theta_1/\beth_1, \ldots, \theta_m/\beth_m].$$

- *If additionally $\eta_i \overset{M}{\rightarrowtail} \theta_i$ for $1 \leq i \leq m$, then also*

$$\vartheta[\theta_1/\beth_1, \ldots, \theta_m/\beth_m] \overset{M}{\leftrightarrowtail} \vartheta[\eta_1/\beth_1, \ldots, \eta_m/\beth_m].$$

Proof. This can be easily shown by induction on the structure of ϑ. □

Lemma 6
Let $M = \mathcal{M}^{\mathbf{A}}_X$. Then the notions of implication and entailment coincide. The same is true for equivalence and congruence.

Proof. If $\theta \rightarrowtail \eta$, we also have $\theta \rightarrowtail \eta$ by applying the rule $\Box\aleph \rightarrowtail \aleph$ from table 3.2 (p. 17). So, suppose $\theta \rightarrowtail \eta$. Let $W = \mathcal{W}(\theta) \cup \mathcal{W}(\eta)$. Let $\Xi = (\xi_i)_{i\in\mathbb{N}} \in \mathcal{M}^{\mathbf{A}}_{X,W}$. Let $i \in \mathbb{N}$ and $\Psi^i = (\xi_{i+j})_{j\in\mathbb{N}}$ be the subsequence of Ξ starting at i. Then $\Psi^i \in \mathcal{M}^{\mathbf{A}}_{X,W}$ and thus $\Psi^j \vDash (\theta \rightarrow \eta)$. So $(\Xi, i) \vDash (\theta \rightarrow \eta)$ for every $i \in \mathbb{N}$. Hence $\Xi \vDash \Box(\theta \rightarrow \eta)$. □

Lemma 7
Let θ entail η with respect to M, $\theta \overset{M}{\rightarrowtail} \eta$. Then $\theta \vee \eta \overset{M}{\leftrightarrowtail} \eta$ and $\theta \wedge \eta \overset{M}{\leftrightarrowtail} \theta$.

Chapter 4

IMP – An imperative programming language

In this chapter, we present a simple imperative programming language called IMP and a number of graph types that are designed to model different aspects of such an imperative program: The control flow, control dependence, and data dependence graph, and, at last, the program dependence graph, which is the base data structure for our path conditions' construction and reasoning.

4.1 Syntax, the language $\mathcal{L}_{\mathsf{IMP}}$, and structures $\mathrm{A}_{\mathsf{IMP}}$ and $\mathrm{A}'_{\mathsf{IMP}}$

We start by introducing IMP. Apart from assignments, it features two control statements: the conditional statement `if` and the loop statement `while` for arbitrary loops. Formally, a **program p in IMP** is a word produced by the following context-free grammar with start symbol S:

$$
\begin{aligned}
A ::= \;& V_i \mid C \mid I \mid \texttt{(}A\texttt{)+(}A\texttt{)} \mid \texttt{-(}A\texttt{)} \mid \texttt{(}A\texttt{)*(}A\texttt{)} \mid \texttt{(}A\texttt{)/(}A\texttt{)} \\
B ::= \;& \texttt{true} \mid \texttt{false} \mid V_b \mid \\
& \texttt{(}A\texttt{)==(}A\texttt{)} \mid \texttt{(}A\texttt{)!=(}A\texttt{)} \mid \texttt{(}A\texttt{)<=(}A\texttt{)} \mid \texttt{(}A\texttt{)<(}A\texttt{)} \mid \\
& \texttt{(}A\texttt{)>=(}A\texttt{)} \mid \texttt{(}A\texttt{)>(}A\texttt{)} \mid \\
& \texttt{!(}B\texttt{)} \mid \texttt{(}B\texttt{)\&\&(}B\texttt{)} \mid \texttt{(}B\texttt{)||(}B\texttt{)} \\
C ::= \;& V_a\texttt{[}A\texttt{]} \\
I ::= \;& \ldots \mid \texttt{-3} \mid \texttt{-2} \mid \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \ldots \\
S ::= \;& C\texttt{:=}A\texttt{;} \mid V_b\texttt{:=}B\texttt{;} \mid V_i\texttt{:=}A\texttt{;} \mid \\
& \texttt{if (}B\texttt{) \{}S\texttt{\}} \mid \texttt{if (}B\texttt{) \{}S\texttt{\} else \{}S\texttt{\}} \mid \\
& \texttt{while (}B\texttt{) \{}S\texttt{\}} \mid SS \\
V_a ::= \;& \text{variables of type } \texttt{array} \\
V_b ::= \;& \text{variables of type } \texttt{bool} \\
V_i ::= \;& \text{variables of type } \texttt{int}
\end{aligned}
$$

Expressions in IMP are all words that can be derived by the grammar when we start with the nonterminal symbols A or B. We will omit brackets in programs whenever the syntactic structure is obvious even without brackets. As usual we consider `*` and `/` to have higher precedence than + and -. For example, we write `x:=a[50+3*i]+(-7);` for `x:=(a[(50)+((3)*(i))])+(-(7));`.

We have not yet specified how variables should look like in IMP programs. We assume that they are a nonempty, but finite sequence of alphanumeric characters and that for every program variable, we have a typed variable in $\mathcal{V}$ which we identify with it. This way, we have an unlimited number of variables at our disposal whose type (`bool`, `int`, or `array`) either is obvious from the context or we state it in the text. Let $\mathbb{V}$ denote the set of all variables in IMP. If $p \in \mathsf{IMP}$ is a program, we write $\mathbb{V}_p$ for the set of variables occurring in p.

We do not give an operational semantics for IMP programs here, we assume that the reader is familiar with the standard semantics of the constructs in IMP. Since we want to generate path formulae IMP programs, we now present how to incorporate IMP expressions in the formalism presented in the previous chapter.

First, we define the language $\mathcal{L}_{\mathsf{IMP}}$ and present two structures $\mathbf{A}_{\mathsf{IMP}}$ and $\mathbf{A}'_{\mathsf{IMP}}$ over which we interpret temporal and Boolean path conditions respectively. For IMP, we choose the language $\mathcal{L}_{\mathsf{IMP}}$ with function operators $\mathcal{F}_{\mathsf{IMP}}$ and predicate operators $\mathcal{R}_{\mathsf{IMP}}$ where

$$\mathcal{F}_{\mathsf{IMP}} := \mathbb{Z} \cup \{\, \mathrm{true}, \mathrm{false}, +, -, *, /, \cdot[\cdot], ==, !=, <, \leq, \geq, >, !, \&\&, \parallel \,\}^{4}$$

and $\mathcal{R}_{\mathsf{IMP}} := \{\, \cdot \,\}^{5}$ with types $\mathcal{T}_{\mathsf{IMP}} := \{\, \texttt{array}, \texttt{bool}, \texttt{int} \,\}$. The arity and type definitions for function and predicate operators are given in table 4.1.

We do not specify the set of variables $\mathcal{V}$ yet. Variables of type `int` or `bool` are called **scalar variables**. In order to save space and keep LTL formulae over $\mathcal{L}_{\mathsf{IMP}}$ readable, if it is obvious from the context, we write a for $\cdot(a)$ for all atomic formulae $\cdot(a) \in \mathfrak{A}$ and use infix notation for all binary function operators.

Next, we define an appropriate structure $\mathbf{A}_{\mathsf{IMP}} = (\underline{\mathrm{A}_{\mathsf{IMP}}}, \tau, \iota_{\mathcal{F}}, \iota_{\mathcal{R}})$ over $\mathcal{L}_{\mathsf{IMP}}$ which we will use for temporal path conditions. Since in most imperative programming languages, the native type `int` can only represent a finite subset of $\mathbb{Z}$, we model this by allowing only 32 bit integers with the usual value range $\overline{\mathbb{Z}}$. However, we need to lift this set to account for variables to which no value has yet been assigned or whose value is undefined. Hence, let $\overline{\mathbb{Z}}^{\perp} := \overline{\mathbb{Z}} \cup \{\, \perp_i \,\}$ denote the set of values for type `int`. Similarly, $\mathbb{B}^{\perp} := \mathbb{B} \cup \{\, \perp_b \,\}$ denotes the set of `bool`

[4] Note that we do distinguish between the operators !, &&, || on `bool` and the Boolean operators $\neg$, $\wedge$, $\vee$ that are part of the logic over $\mathcal{L}_{\mathsf{IMP}}$. Equally, we consider the arithmetic comparators ==, !=, <, $\leq$, $\geq$, > as function operators which return a value of type `bool`. This way we are able to strictly separate between conjunctions and disjunctions in IMP (with their different interpretation options in different structures) that form a single expression and Boolean conjunctions and disjunctions on the logic level. Nevertheless, definition 17 (p. 21) allows us to exploit the properties of !, &&, and || in the specific structure when we want to simplify an LTL formula over $\mathcal{L}_{\mathsf{IMP}}$.

[5] The predicate operator $\cdot$ is designed to map values of `bool` to $\mathbb{B}$.

Operator s	**Arity**	**Parameter types**	**Return type**	**Comment**
$i \in \mathbb{Z}$	0		`int`	Integer constants
true, false	0		`bool`	Boolean constants
$+, *, /$	2	`int`,`int`	`int`	
$-$	1	`int`	`int`	
$\cdot[\cdot]$	2	`array`,`int`	`int`	Array cell access operator
$==$	2	`int`,`int`	`bool`	Equality on `int`
$!=$	2	`int`,`int`	`bool`	Inequality on `int`
$<, \leq, \geq, >$	2	`int`,`int`	`bool`	Comparators on `int`
!	1	`bool`	`bool`	Negation on `bool`
&&, ||	2	`bool`,`bool`	`bool`	Binary operators on `bool`
$\cdot$	1	`bool`		Mapping `bool` to $\mathbb{B}^\perp$

Table 4.1: Arity and type definitions for function and predicate operators in $\mathcal{L}_{\mathsf{IMP}}$.

values and for `array` types we choose the set

$$\mathbb{A} := \left\{ a : \overline{\mathbb{Z}}^\perp \mapsto \overline{\mathbb{Z}}^\perp \;\middle|\; a(\perp_i) = \perp_i \right\}$$

We write $\perp_a$ for the constant map $\overline{\mathbb{Z}}^\perp \mapsto \overline{\mathbb{Z}}^\perp$, $x \mapsto \perp_i$.

Then, the universe $\underline{\mathrm{A}_{\mathsf{IMP}}}$ of $\mathbf{A}_{\mathsf{IMP}}$ is

$$\underline{\mathrm{A}_{\mathsf{IMP}}} := \overline{\mathbb{Z}}^\perp \cup \mathbb{B}^\perp \cup \mathbb{A}$$

We write $\perp$ for the set $\{ \perp_a, \perp_b, \perp_i \}$. The type function τ is uniquely determined by $\tau\left(\overline{\mathbb{Z}}^\perp\right) = \{ \texttt{int} \}$, $\tau\left(\mathbb{B}^\perp\right) = \{ \texttt{bool} \}$, and $\tau(\mathbb{A}) = \{ \texttt{array} \}$.

We omit the formal definitions for the interpretation functions. $+$, $-$, $*$ and $/$ are interpreted as usual with $+$ meaning addition, $-$ computing the additive inverse, $*$ meaning multiplication and $/$ division, always within $\overline{\mathbb{Z}}$. The undefined value $\perp_i$ is propagated from every parameter, division by 0 yields 0. Similarly, $==$ ($!=$) yields `T` iff both parameter values are in $\overline{\mathbb{Z}}$ and (not) equal. They yield $\perp_b$ iff at least one parameter is $\perp_i$. Analogous definitions hold for the other comparators $<$, $\leq$, $\geq$, and $>$. For the connectives over `bool`, they have the usual semantic, but they, too, propagate $\perp_b$ from parameters to results. The array cell access operator $\cdot[\cdot]$ is defined by $a[x] := a(x)$ for $a \in \mathbb{A}$ and $x \in \overline{\mathbb{Z}}^\perp$. The only relational symbol $\cdot$ is interpreted as the map $\cdot^{\mathbf{A}_{\mathsf{IMP}}} : \mathbb{B}^\perp \mapsto \mathbb{B}$, $\cdot^{\mathbf{A}_{\mathsf{IMP}}}(x) = \texttt{T}$ iff $x = \texttt{T}$.

Boolean path conditions do not model temporal aspects of program execution. Hence, there is no need for undefined variable values. Thus, we define $\mathbf{A}'_{\mathsf{IMP}}$ to be the structure for $\mathcal{L}_{\mathsf{IMP}}$ whose universe $\underline{\mathrm{A}'_{\mathsf{IMP}}} = \overline{\mathbb{Z}} \cup \mathbb{B} \cup \overline{\mathbb{Z}}^{\overline{\mathbb{Z}}}$ is $\underline{\mathrm{A}_{\mathsf{IMP}}}$ without lifted values. The typing function and interpretation functions for function and predicate symbols are the functions from $\mathbf{A}_{\mathsf{IMP}}$ which are appropriately restricted to the smaller universe $\underline{\mathrm{A}'_{\mathsf{IMP}}}$.

4.2 The control flow graph

As a first level of abstraction from the program source code, control flow graphs, which were first proposed by Allen [All70], are widely used in compilers and optimization as an intermediate representation of a program. Unlike in these applications do we neither merge several statements into so-called basic blocks nor do we have multiple nodes for a single statement. In our setting, each node corresponds to a single statement and edges represent the control flow between nodes.

Definition 19 (Control flow graph)
Formally, a **control flow graph (CFG)**, represented as a 5-tuple (V, E, v_e, v_x, α), is a directed graph (V, E) with two distinguished nodes (the entry node) $v_e \in V$ and (the exit node) $v_x \in V$ and such that $\text{pred}(v_e) = \emptyset = \text{succ}(v_x)$ and that there exists a path from v_e to every node $v \in V$, and a labelling function $\alpha : E \mapsto \{ \text{true}, \text{false}, \epsilon \}$.

If CFG_p is the control flow graph for a program $p \in \mathsf{IMP}$, each node $v \in V - \{ v_e, v_x \}$ represents a statement or branching / looping condition of p, each edge $e \in E$ represents a possible flow of control from the source node to the target node, i. e. no other statement is executed between e's source node $\odot\!\!\rightarrow(e)$ and its target node $\rightarrow\!\!\odot(e)$ when the control flow passes along e. If e's source node represents a branching / looping condition, $\alpha(e)$ is this condition's evaluation result that makes the control flow take e. Otherwise, we set $\alpha(e) = \epsilon$. In addition, we require $(v_e, v_x) \in E$ for every control flow graph.

Note that for a program $p \in \mathsf{IMP}$, we have that all strongly connected components in $\text{CFG}_p = (V, E, v_e, v_x, \alpha)$, which correspond to `while` loops, have a single entry and exit node, namely the node corresponding to the `while` predicate because IMP allows only programs with structured control flow, i. e. no `goto`s, exceptions and the like are part of IMP. A **back edge** $e \in E$ is an edge that returns to a while loop node. More formally, a back edge is a retreating edge in the spanning tree computed by a depth-first search starting at v_e. Due to the structure of CFG_p, this spanning tree is uniquely determined, so back edges are well-defined. We say, a back edge $e \in E$ **belongs to** $\rightarrow\!\!\odot(e)$. See figure 6.1 (p. 48) for an example CFG. There, the edge $(5, 4)$ is a back edge that belongs to node 4.

4.2.1 Dominance

Dominators and postdominators [Pro59, LM69] are used to characterize the nodes in a CFG at which control flow branches or at which different control flow paths are joined again.

Definition 20 (Dominance and postdominance)
We say that for a CFG $G = (V, E, v_e, v_x, \alpha)$, node $v \in V$ **dominates** $w \in V$ iff every path $\pi : v_e \xrightarrow[G]{}{}^{*} w$ from entry to w contains v, denoted by $v \,\mathsf{DOM}\, w$. Conversely, v **postdominates** w iff every path $\pi : w \xrightarrow[G]{}{}^{*} v_x$ from w to exit contains v, denoted by $v \,\mathsf{PDOM}\, w$. We write $\mathsf{DOM}\,(w) = \{ w' \in V \mid w' \,\mathsf{DOM}\, w \}$

for the set of nodes that dominate w and $\mathsf{PDOM}\,(w) = \{\, w' \in V \mid w'\,\mathsf{PDOM}\,w \,\}$ for the set of nodes that postdominate w.

Note that every node dominates and postdominates itself. Moreover, dominators in $G = (V, E, v_e, v_x)$ are exactly the postdominators in the reverse graph $G^{-1} = (V, E^{-1}, v_x, v_e)$ and vice versa. A stronger notion is the concept of immediate dominators:

Definition 21 (Immediate dominators)
A node $v \in V$ **immediately dominates** $w \in V$, denoted by $v\,\mathsf{IDOM}\,w$, iff $v \neq w$, $v\,\mathsf{DOM}\,w$ and $w'\,\mathsf{DOM}\,v$ holds for every $w' \in \mathsf{DOM}\,(w) - \{\, w \,\}$.

Intuitively, v immediately dominates w if v is the closest proper dominator of w. Note that every node has exactly one immediate dominator.

4.2.2 Control dependence

Usually, one says a node $v \in V$ is control dependent on $w \in V$ if w corresponds to a control statement in p and v being executed depends on the evaluation of w [LM69]. If we abstract from loops not necessarily terminating, then only the nodes that are in a loop's body ought to be control dependent on the loop's condition node. Otherwise, each statement outside a while loop that can be reached from this loop becomes control dependent on this loop, too, because its execution may depend on that loop terminating. [RAB+05] discusses this issue in detail and presents a number of different notions of control dependence. Although non-termination is an issue in the context of influence analysis, we stick to the old-fashioned definition of control dependence (see below) because non-termination sensitive control dependence would make path conditions much more complicated as the control dependence graph is no longer a tree. Instead, we include extra generation rules for path conditions to require preceding loops having terminated. Note however that this does not capture influences that arise from a loop not terminating.

Definition 22 (Control dependence)
For an IMP program p, we say $v \in W$ is **control dependent** on $w \in W$ (denoted by $w \xrightarrow[p]{\mathsf{cd}\triangleright} v$) iff

- there exists a path $w \xrightarrow[\mathrm{CFG}_p]{}{}^{*} v$ from w to v in CFG_p,
- v is a postdominator for some $w' \in \mathrm{succ}(w)$, and
- v is not a postdominator for some $w'' \in \mathrm{succ}(w)$.

In terms of the CFG, v is control dependent on w if w has multiple outgoing edges. Following one of them we ultimately reach v, following another we can avoid reaching v. We will write $\xrightarrow[p]{\mathsf{cd}\triangleright}{}^{*}$ to denote the reflexive and transitive closure of the relation $\xrightarrow[p]{\mathsf{cd}\triangleright}$. The control dependence relation is represented by the control dependence graph:

Definition 23 (Control dependence graph)
The **control dependence graph** CDG_p for the control flow graph $\mathrm{CFG}_p = (V, E, v_e, v_x, \alpha)$ is the graph $\mathrm{CDG}_p = (V - \{ v_x \}, C, v_e, \overline{\alpha})$ where

- $(v, w) \in C$ iff $v \xrightarrow[p]{\mathsf{cd}} w$,
- $\overline{\alpha} : C \to \mathfrak{P}(\alpha(E))$ is a labelling function such that

$$\overline{\alpha}((v, w)) = \left\{ \alpha((v, w')) \;\middle|\; w' \in \mathrm{succ}(v) \wedge w \,\mathsf{PDOM}\, w' \right\},$$

  i. e. $\overline{\alpha}$ collects all labels of outgoing edges from v which lead to successors of v that are postdominated by w.

As IMP allows only structured control flow, the control dependence graph CDG_p is a tree for all programs $p \in \mathsf{IMP}$.

4.3 The data dependence graph

Apart from control dependences there are also data dependences in a program. In the broadest sense, a data dependence occurs if two statements s and t access the same variable or memory location. Four different types of data dependences are distinguished [Wol96]: Let statement s precede t in the execution order.

- If both s and t are read accesses with respect to the memory location considered, the dependence is called an **input dependence**.
- If both s and t write to it, it is called an **output dependence**.
- If s reads from and t writes to it, **anti dependence**.
- If s writes to and t reads from the memory location, we call it a **true dependence**.

A data dependence is called **direct** if there is no operation r between s and t which modifies this memory location. A direct true dependence is called **flow dependence**.

Here, we are only interested in direct output and flow dependences. More precisely, we ought to consider operation instances instead of statements in a program $p \in \mathsf{IMP}$, because if a statement accessing an array cell in a loop is executed multiple times, the cell accessed at the first time (and thus the memory location we are looking at) may be different from the one affected in later iterations. Hence, we would have to define flow dependences over operation instances which may be infinitely many in the case of a nonterminating loop. Instead of representing operation instances separately, we subsume all of a statement in a single node and conservatively approximate the data dependence relation: If there can possibly be a data dependence between two operation instances of two statements, then there must be a data dependence edge between both statements' nodes. Since we will create far too many data dependences this way, we will formulate necessary conditions over the program's variables for every such dependence edge being real.

To define these data dependences more formally, we consider again a program $p \in \mathsf{IMP}$ and its control flow graph $\mathrm{CFG}_p = (V, E, v_e, v_x, \alpha)$. For every node $v \in V$, we define the set $\mathbf{def}(v)$ of variables which are defined (written) at v and the set $\mathbf{use}(v)$ of variables which used (read) at v. Moreover, we also define a set $\mathbf{reuse}(v) \subseteq \mathbf{def}(v)$ that contains all array variables which are written to at v.

First, we inductively define **use** sets for expressions in IMP, i. e. for subprograms that are produced by the nonterminal symbols A, and B.

- If e is a variable, i. e. e is produced by one of the nonterminal symbols V_a, V_b, and V_i, we set $\mathbf{use}(e) := \{ e \}$.
- If e is a constant, i. e. e is produced by the nonterminal symbol I or $e \in \{ \texttt{true}, \texttt{false} \}$, we set $\mathbf{use}(e) := \emptyset$.
- If $e = \circ(f)$ where $\circ \in \{ -, ! \}$ and f is another IMP expression, we set $\mathbf{use}(e) := \mathbf{use}(f)$.
- If $e = (f_1) \circ (f_2)$ where $\circ \in \{ +, *, /, ==, !=, <=, <, >, >=, \&\&, \| \}$ and f_1, f_2 are IMP expressions, we set $\mathbf{use}(e) := \mathbf{use}(f_1) \cup \mathbf{use}(f_2)$.
- If $e = a[f]$ where a is a variable of type `array` and f an IMP expression, we set $\mathbf{use}(e) := \{ a \} \cup \mathbf{use}(f)$.

Next, we define the **def**, **reuse** and **use** sets for all nodes in the CFG. For the entry and exit nodes v_e and v_x, set $\mathbf{def}(v_e) := \mathbf{reuse}(v_e) := \mathbf{use}(v_e) := \emptyset =: \mathbf{def}(v_x) =: \mathbf{reuse}(v_x) =: \mathbf{use}(v_x)$. For every other node $v \in V$, we have:

- If v corresponds to an assignment `x:=`e`;` where `x` is a Boolean or integer variable, we set $\mathbf{def}(v) := \{ x \}$, $\mathbf{reuse}(v) := \emptyset$, and $\mathbf{use}(v) := \mathbf{use}(e)$.
- If v corresponds to an assignment `a[`i`]:=`e`;` where `a` is an array variable, $i \in \mathfrak{E}$ is the index expression and $e \in \mathfrak{E}$ is the expression whose value is assigned to the array cell `a[`i`]`, we set $\mathbf{def}(v) := \{ a \}$, $\mathbf{reuse}(v) := \{ a \}$, and $\mathbf{use}(v) := \mathbf{use}(i) \cup \mathbf{use}(e)$.
- If v is a branching node with condition b, we set $\mathbf{def}(v) := \emptyset$, $\mathbf{reuse}(v) := \emptyset$, and $\mathbf{use}(v) := \mathbf{use}(b)$.

Note that we consider $\mathbf{def}(v)$ and $\mathbf{use}(v)$ to be always disjoint.

We now are able to define the notions of data dependences we are interested in. First, we consider data dependences along a given path in the control flow graph.

Definition 24 (Flow and def-def dependence for CFG paths)
Let π be a path in CFG_p and $v, w \in \pi$ such that $v \prec_\pi w$. If $x \in \mathbf{def}(v) \cap \mathbf{use}(w) \neq \emptyset$, then w is **flow dependent** on v with respect to x if $x \notin \mathbf{def}(w')$ for every $w' \in \mathrm{V}(\pi)$ with $v \prec_\pi w'' \prec_\pi w$. We denote this dependence by $v \xrightarrow[x,\pi]{\mathsf{fd}\triangleright} w$.[6]

Similarly, if $x \in \mathbf{def}(v) \cap \mathbf{reuse}(w)$, we say w is **def-def dependent**[7] on v with respect to x if $x \notin \mathbf{def}(w')$ for every $w' \in \mathrm{V}(\pi)$ with $v \prec_\pi w' \prec \pi w$. We denote this kind of dependence by $v \xrightarrow[x,\pi]{\mathsf{dd}\triangleright} w$.

We can distinguish further between loop-carried and loop-independent flow and def-def dependences [HRB88].

Definition 25 (Loop-carried and loop-independent data dependence)
Let $v \xrightarrow[x,\pi]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x,\pi]{\mathsf{dd}\triangleright} w$) be a flow (def-def) dependence between v and w with respect to x. We say $v \xrightarrow[x,\pi]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x,\pi]{\mathsf{dd}\triangleright} w$) is **loop carried by loop** u if there is a back edge $e \in \mathrm{E}(\pi)$ with $u = \multimap\!\!\odot(e)$ such that

- e is between v and w in π, i. e. $v \preceq_\pi \odot\!\!\succ(e)$ and $\multimap\!\!\odot(e) \preceq_\pi w$,
- both $u \xrightarrow[p]{\mathsf{cd}\triangleright}{}^{*} v$ and $u \xrightarrow[p]{\mathsf{cd}\triangleright}{}^{*} w$, and
- for every further back edge $e' \in \mathrm{E}(\pi) - \{ e \}$ we have not $\multimap\!\!\odot(e') \xrightarrow[p]{\mathsf{cd}\triangleright}{}^{*} u$.

We denote this by $v \xrightarrow[x,\pi,u]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x,\pi,u]{\mathsf{dd}\triangleright} w$). If $v \xrightarrow[x,\pi]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x,\pi]{\mathsf{dd}\triangleright} w$) is not loop carried by some $u \in V$, we say $v \xrightarrow[x,\pi]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x,\pi]{\mathsf{dd}\triangleright} w$) is **loop independent**.

In other words, if u is the loop node for the loop-carried dependence edge $v \xrightarrow[x,\pi]{\mathsf{fd}\triangleright} w$, u is the node that corresponds to the loop predicate of the outermost loop among all loops to which a back edge belongs that is contained in $\mathrm{E}(\pi)$ between v and w.

By joining all flow dependences over all paths between two statements v and w we abstract data dependence from concrete CFG paths:

Definition 26 (Flow and def-def dependence)
We say w is **flow dependent** on v with respect to variable x, denoted by $v \xrightarrow[x]{\mathsf{fd}\triangleright} w$, iff there exists a path $\pi : v \xrightarrow[\mathrm{CFG}_p]{}{}^{*} w$ such that $v \xrightarrow[x,\pi]{\mathsf{fd}\triangleright} w$. Similarly, we say w is **def-def dependent** on v with respect to variable x, denoted by $v \xrightarrow[x]{\mathsf{dd}\triangleright} w$, iff there exists a path $\pi : v \xrightarrow[\mathrm{CFG}_p]{}{}^{*} w$ such that $v \xrightarrow[x,\pi]{\mathsf{dd}\triangleright} w$.

Let

$$\Pi := \left\{ \pi : v \xrightarrow[\mathrm{CFG}_p]{}{}^{*} w \;\middle|\; \forall w' \in \mathrm{V}(\pi) - \{ v, w \} : x \notin \mathbf{def}(w') \right\}$$

[6] Note that $x \in \mathbf{def}(v) \cap \mathbf{use}(w)$ not only represents a simple program variable but also encodes both the exact occurrence of `x` in v responsible for defining `x` and the occurrence of `x` in w that has generated $x \in \mathbf{use}(w)$. In this sense, we may even have $\mathbf{def}(v) \cap \mathbf{use}(v) \neq \emptyset$.

[7] Notice that def-def dependence is a special case of output dependence: If we consider a whole array as a single memory location, every direct output dependence with respect to an array variable is also a def-def dependence.

be the set of all paths from v to w in CFG_p where no redefinition of x occurs. For a flow (def-def) dependence $v \xrightarrow[x]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x]{\mathsf{dd}\triangleright} w$) we write

$$\mathrm{L}^c\left(v \xrightarrow[x]{\mathsf{fd}\triangleright} w\right) := \left\{ u \in V \;\middle|\; \exists \pi \in \Pi : v \xrightarrow[x,\pi,u]{\mathsf{fd}\triangleright} w \right\}$$

$$\left(\mathrm{L}^c\left(v \xrightarrow[x]{\mathsf{dd}\triangleright} w\right) := \left\{ u \in V \;\middle|\; \exists \pi \in \Pi : v \xrightarrow[x,\pi,u]{\mathsf{dd}\triangleright} w \right\}\right)$$

for the set of loop predicate nodes in CFG_p which may carry a flow (def-def) dependence from v to w with respect to x.

Note that if $v \xrightarrow[x,\pi]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x,\pi]{\mathsf{dd}\triangleright} w$) for every path $\pi \in \Pi$ is loop independent, then $\mathrm{L}^c\left(v \xrightarrow[x]{\mathsf{fd}\triangleright} w\right) = \emptyset$ ($\mathrm{L}^c\left(v \xrightarrow[x]{\mathsf{dd}\triangleright} w\right) = \emptyset$). Besides the notions of loop-carried and loop-independent dependences, we also want to know whether a flow (def-def) dependence $v \xrightarrow[x]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x]{\mathsf{dd}\triangleright} w$) leaves a loop u or not.

Definition 27 (Data dependences leaving loops)
We say $v \xrightarrow[x]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x]{\mathsf{dd}\triangleright} w$) **leaves loop** u iff

- u is a loop predicate node,
- v is transitively control dependent on u: $u \xrightarrow[p]{\mathsf{cd}\triangleright}{}^{*} v$, and
- w is not transitively control dependent on u: not $u \xrightarrow[p]{\mathsf{cd}\triangleright}{}^{*} w$.

Let $\mathrm{L}^x\left(v \xrightarrow[x]{\mathsf{fd}\triangleright} w\right)$ ($\mathrm{L}^x\left(v \xrightarrow[x]{\mathsf{dd}\triangleright} w\right)$) denote the set of all loop nodes u that are left by $v \xrightarrow[x]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x]{\mathsf{dd}\triangleright} w$).

As with control dependence, the flow and def-def dependence relation can be represented as a graph, the so-called data dependence graph (DDG). Note that neither entry nor exit node participates in either flow or def-def dependence because their **def**, **reuse**, and **use** sets are all empty.

Definition 28 (Data dependence graph)
The **data dependence graph (DDG)** for a program $p \in \mathsf{IMP}$ is a multi-graph $\mathrm{DDG}_p = (V, D, \mathrm{L})$ where

- $V := V' - \{ v_e, v_x \}$ is the set of nodes as in the control flow graph $\mathrm{CFG}_p = (V', E, v_e, v_x, \alpha)$ of p except for the entry and exit nodes,
- D is the set of all flow and def-def dependences, i. e. $\ominus\!\!\succ$ and $\prec\!\!\ominus$ are implicitly defined as $v \xrightarrow[x]{\mathsf{fd}\triangleright} w \mapsto v$, $v \xrightarrow[x]{\mathsf{dd}\triangleright} w \mapsto v$ and $v \xrightarrow[x]{\mathsf{fd}\triangleright} w \mapsto w$, $v \xrightarrow[x]{\mathsf{dd}\triangleright} w \mapsto w$, respectively, and
- $\mathrm{L} : D \mapsto \mathfrak{P}(V) \times \mathfrak{P}(V)$ is the labelling function that assigns each flow dependence $v \xrightarrow[x]{\mathsf{fd}\triangleright} w \in D$ the tuple of sets

  $$\mathrm{L}(v \xrightarrow[x]{\mathsf{fd}\triangleright} w) := \left(\mathrm{L}^c\left(v \xrightarrow[x]{\mathsf{fd}\triangleright} w\right), \mathrm{L}^x\left(v \xrightarrow[x]{\mathsf{fd}\triangleright} w\right)\right)$$

  and each def-def dependence $v \xrightarrow[x]{\mathsf{dd}\triangleright} w \in D$ the tuple of sets

  $$\mathrm{L}(v \xrightarrow[x]{\mathsf{dd}\triangleright} w) := \left(\mathrm{L}^c\left(v \xrightarrow[x]{\mathsf{dd}\triangleright} w\right), \mathrm{L}^x\left(v \xrightarrow[x]{\mathsf{dd}\triangleright} w\right)\right).$$

4.4 The program dependence graph

The program dependence graph [FOW87] combines both control and data dependence in an imperative program.

Definition 29 (Program dependence graph)
The **program dependence graph (PDG)** for a program $p \in \mathsf{IMP}$ is a multigraph $\mathrm{PDG}_p = (V, C, D, v_e, \overline{\alpha}, \mathrm{L})$ where

- $V := V' - \{ v_x \}$ is the set of nodes from the control flow graph $\mathrm{CFG}_p = (V', E, v_e, v_x, \alpha)$ except for the exit node v_x,
- C is the set of control dependence edges from the control dependence graph $\mathrm{CDG}_p = (V, C, v_e, \overline{\alpha})$,
- D is the set of flow and def-def dependence edges from the data dependence graph $\mathrm{DDG}_p = (V, D, \mathrm{L})$
- v_e is the entry node of the CFG and the root of the CDG
- $\overline{\alpha}$ the labelling function for control dependence edges in C, and
- L is the labelling function for flow and def-def dependence edges in D.

Without loss of generality, we assume $C \cap D = \emptyset$. Then, the mappings $\odot\!\!\rightarrow$ and $\rightarrow\!\!\otimes$ are the combined versions of $\odot\!\!\rightarrow$ and $\rightarrow\!\!\otimes$ from the CDG and DDG.

Chapter 5

Boolean path conditions for IMP

We now present Boolean path conditions as proposed by Snelting, Krinke and Robschink in [Sne96, RS02, Rob04, SRK]. Since there have been proposed a large number of variants, we pick one that is easy to understand and that fits well into our setting and adopt it to our programming language and notation. Note however that there are numerous optimizations in [Rob04, SRK] that we have not included here.

Definition 30 (Boolean path condition)
A **Boolean path condition** for two statements s and t of an IMP program p is a Boolean formula θ over atomic formulae over $\mathcal{L}_{\mathsf{IMP}}$, i. e. $\theta \in \mathsf{SF}^{\mathcal{L}_{\mathsf{IMP}}}$, such that if there is an influence from s to t then there is a state $\xi \in \mathcal{S}^{\mathbf{A}'_{\mathsf{IMP}}}_{\mathcal{V}(\theta)}$ with $\xi(\theta) = \mathtt{T}$, i. e. ξ is an assignment to variables $\mathcal{V}(\theta)$ that satisfies θ.

Obviously, true is always a path condition, but this gives exactly the information slicing also provides. We therefore want that Boolean path conditions are almost always unsatisfiable if there is no influence.

This chapter is organized as follows: First, we present three different types of constraints that are the basic components of Boolean path conditions:

- Φ constraints generated by the SSA transformation of IMP programs,
- Execution conditions from control dependence, and
- Φ constraints from data dependence edges.

Next, we combine these elements to define Boolean path conditions for IMP programs with scalar variables. Before we discuss how to handle cycles in the PDG, we show how arrays and the additional constraints we generate for them can be incorporated into Boolean path conditions.

5.1 Static single assignment form

In an IMP program, variables can occur multiple times on the left-hand side of an assignment, i. e. a variable can have different values at different program points. Therefore, for Boolean path conditions, all programs are transformed to static single assignment form (SSA, [CFR+91]): A program is in SSA form if every program variable appears at most once on the left-hand side of an assignment. At nodes where control flow paths meet, Φ functions are introduced which dynamically select the right source variable.

Figure 5.1 A program and its SSA transformation on the right.

```
 1 a := 1;                          1 a1 := 1;
 2 i := 1;                          2 i1 := 1;
 3 if (b) {                         3 if (b) {
 4     i := 2;                      4     i2 := 2;
 5 }                                5 }
                                      i3 := Φ(i1, i2)
 6                                  6
 7 while (i < 3) {                  7 while (i4 := Φ(i3, i5),
                                             a2 := Φ(a1, a3),
                                             i4 < 3) {
 8     a := a * 2;                  8     a3 := a2 * 2;
 9     i := i + 1;                  9     i5 := i4 + 1;
10 }                               10 }
11 y := a;                         11 y := a2;
```

Example 5 (SSA)
Let us take a look at the program shown in figure 5.1 on the left. There are two lines where control flow paths can meet: line 5 (at the end of the `if` statement) and line 7 (at the `while` loop predicate).

At line 5, we have more than one possible def node only for variable `i`, so we introduce a single Φ function which selects the correct definition at runtime. Inside the while loop variables i and a are redefined. Therefore, we need to include in the loop predicate their Φ functions.

Since we do not need a minimal SSA form, i. e. one that avoids unnecessary indices and Φ functions, we can consider our approach of distinguishing all occurrences of a variable as an SSA form, too. In fact, ValSoft, a slicer for full ANSI C [KSR99, KS98, Kri03], contains an implementation of Boolean path conditions that does distinguish all occurrences of a variable and generates all necessary Φ functions.

For Boolean path conditions, Φ functions are translated into additional constraints. Suppose we have a Φ function of the form $\mathtt{x}_{i_0}$:= $\Phi(\mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_k})$ for SSA

variants $\mathtt{x}_{i_0}, \mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_k}$ of program variable $\mathtt{x}$. This generates the constraint

$$\Phi(\mathtt{x}_{i_0}, \mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_k}) := \bigvee_{j=1}^{k} \left(\mathtt{x}_{i_0} \equiv \mathtt{x}_{i_j}\right). \tag{5.1}$$

5.2 Execution conditions from control dependence

Execution conditions from control dependence are based on the observation that if statement s is to be executed, then there must be a CDG path $v_e \xrightarrow[\text{CDG}_p]{}\!\!\!\!\blacktriangleright^{*}\ s$ where all control conditions are satisfied [Sne96]. More precisely, on at least one CDG path $v_e \xrightarrow[\text{CDG}_p]{}\!\!\!\!\blacktriangleright^{*}\ s$ all control conditions must be satisfied the last time the corresponding statements have been executed. Since we assume that the program is in SSA form, these conditions are also satisfied when s is executed.

Definition 31 (Execution condition from control dependence)
Let $\Pi'_s := \left\{ \pi : v_e \xrightarrow[\text{CDG}_p]{}\!\!\!\!\blacktriangleright^{*}\ s \right\}$ be the set of paths from the entry node v_e to s in $\text{CDG}_p = (V, C, v_e, \overline{\alpha})$.

Let $(v, w) \in C$ be a control dependence edge. The execution condition for edge (v, w) and label $\lambda \in \overline{\alpha}((v, w))$ is given by

$$\mathsf{E}((v,w),\lambda) := \begin{cases} \text{true} & \text{if } \lambda = \epsilon \\ \cdot(v) & \text{if } \lambda = \text{true} \\ \cdot(!(v)) & \text{if } \lambda = \text{false} \end{cases}$$

where we identify v with the control predicate of the statement that v belongs to.

The execution condition $\mathsf{E}(\pi)$ for s along a control dependence path $\pi \in \Pi'_s$ is defined as

$$\mathsf{E}(\pi) := \bigwedge_{e \in \mathrm{E}(\pi)} \left(\bigvee_{\lambda \in \overline{\alpha}(e)} \mathsf{E}(e, \lambda) \right).$$

Then, the **execution condition from control dependence** $\mathsf{E}_{\mathsf{cd}}(s)$ for s is defined as the disjunction over all paths in Π'_s:

$$\mathsf{E}_{\mathsf{cd}}(s) := \bigvee_{\pi \in \Pi'_s} \mathsf{E}(\pi) = \bigvee_{\pi \in \Pi'_s} \bigwedge_{e \in \mathrm{E}(\pi)} \bigvee_{\lambda \in \overline{\alpha}(e)} \mathsf{E}(e, \lambda) \tag{5.2}$$

Note that we generate for every edge $e \in \mathrm{E}(\pi)$ the term $\bigvee_{\lambda \in \overline{\alpha}(e)} \mathsf{E}(e, \lambda)$ only once since, due to idempotence of the Boolean $\wedge$ operator, the formula we obtain is equivalent. Since $\vee$ is idempotent, too, we can restrict ourselves to edge-disjoint paths. This means that in any case $\mathsf{E}_{\mathsf{cd}}(s)$ can be written as a formula of finite length. Furthermore, it is even sufficient to consider only node-disjoint, i. e. cycle-free, paths

$$\pi \in \Pi_s := \left\{ v_1, \ldots, v_n \in \Pi'_s \;\middle|\; v_i \neq v_j \text{ for } 1 \leq i < j \leq n \right\} :$$

Let $\rho \in \Pi'_s - \Pi_s$ be a path containing cycles and let $\rho' \in \Pi$ be the path obtained from ρ by removing all cycles. The execution condition for ρ can then be written as

$$\mathsf{E}(\rho) \Longleftrightarrow \underbrace{\left(\bigwedge_{e \in \mathrm{E}(\rho')} \bigvee_{\lambda \in \overline{\alpha}(e)} \mathsf{E}(e, \lambda) \right)}_{= \mathsf{E}(\rho')} \wedge \left(\bigwedge_{e \in (\mathrm{E}(\rho) - \mathrm{E}(\rho'))} \bigvee_{\lambda \in \overline{\alpha}(e)} \mathsf{E}(e, \lambda) \right)$$

Absorption law $(\aleph \vee (\aleph \wedge \beth)) \Longleftrightarrow \aleph$ gives that $\mathsf{E}_{\mathsf{cd}}(s) \Longleftrightarrow \bigvee_{\pi \in \Pi_s} \mathsf{E}(\pi)$. See example 10 (p. 49) for an example for execution conditions.

5.3 Φ constraints for data dependence edges

Suppose we have a flow (def-def) dependence edge $v \xrightarrow[x]{\mathsf{fd}} w$ ($v \xrightarrow[x]{\mathsf{dd}} w$) with respect to variable x. Let $x_v \in \mathbf{def}(v)$ denote the occurrence of x in v and $x_w \in \mathbf{use}(w)$ ($x_w \in \mathbf{reuse}(w)$) the occurrence of x in w. w being influenced by v with respect to x depends on the value stored in x_v being passed to x_w. Hence a necessary constraint for the influence is that x_v has the same value as x_w [SRK].

Definition 32 (Φ contraints for data dependence edge)
Let the identifiers v, w, x, x_v and x_w be as above. The Φ constraint ${}^{v}\Phi_x^w$ for $v \xrightarrow[x]{\mathsf{fd}} w$ ($v \xrightarrow[x]{\mathsf{dd}} w$) is then defined as

- If x is of type `int`, i. e. $(\!|x|\!) = \texttt{int}$, we set

$$ {}^{v}\Phi_x^w := (x_v == x_w). $$

- If x is of type `bool`, i. e. $(\!|x|\!) = \texttt{bool}$, we set

$$ {}^{v}\Phi_x^w := (x_v \&\& x_w) || (!(x_v) \&\& !(x_w))). $$

- If x is of type `array`, i. e. $(\!|x|\!) = \texttt{array}$, let i_v be the index expression of array x in node v and i_w the one in node w. We set

$$ {}^{v}\Phi_x^w := (x_v[i_v] == x_w[i_w]). \tag{5.3} $$

5.4 Boolean path conditions without arrays

For now, we only consider programs that do not make use of arrays.

The fundamental formula for a Boolean path condition for s and t has been introduced by Snelting [Sne96]: For a path $\pi : s \xrightarrow[\mathrm{PDG}_p]{}{}^{*} t$, we set

$$ \mathsf{BPC}(\pi) := \bigwedge_{v \in \mathrm{V}(\pi)} \mathsf{E}_{\mathsf{cd}}(v) \tag{5.4} $$

where $\mathsf{E}_{\mathsf{cd}}(v)$ is the execution condition for node v as defined in 5.2. Then

$$\mathsf{BPC}(s,t) := \bigvee_{\pi : s \xrightarrow[\mathrm{PDG}_p]{}^{*} t} \mathsf{BPC}(\pi). \tag{5.5}$$

However, different SSA variants of a program variable `x` are still unrelated. Hence, we include some additional constraints:

1. We know that if we have $\mathtt{x}_{i_0}$ `:=` $\Phi(\mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_k})$, then $\mathtt{x}_{i_0}$ does have one of the values of $\mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_k}$. Therefore, we can conjunctively add the constraint $\Phi(\mathtt{x}_{i_0}, \mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_k})$ from equation 5.1 (p. 35) to the path condition:

   Let $\pi : s \xrightarrow[\mathrm{PDG}_p]{}^{*} t$ be a path from s to t and let Φ_π denote the set of all Φ conditions $\Phi(\mathtt{x}_{i_0}, \mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_k})$ for program p that contain variables which occur in $\bigwedge_{v \in \mathrm{V}(\pi)} \mathsf{E}_{\mathsf{cd}}(v)$, i. e.

$$\mathcal{V}\left(\Phi(\mathtt{x}_{i_0}, \mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_k})\right) \cap \bigcup_{v \in \mathrm{V}(\pi)} \mathcal{V}\left(\mathsf{E}_{\mathsf{cd}}(v)\right) \neq \emptyset$$

   Then, an improved version of equation 5.4 is

$$\mathsf{BPC}(\pi) := \bigwedge_{v \in \mathrm{V}(\pi)} \mathsf{E}_{\mathsf{cd}}(v) \wedge \bigwedge \Phi_\pi \tag{5.6}$$

   where for a finite set M of Boolean formulae $\bigwedge M := \bigwedge_{m \in M} m$.

2. We also include Φ constraints for data dependence edges: For every data dependence edge $v \xrightarrow[x]{\mathsf{fd}}\!\triangleright\, w \in \mathrm{E}(\pi)$, let

$$\Phi(v \xrightarrow[x]{\mathsf{fd}}\!\triangleright\, w) := {}^{v}\Phi_{x}^{w}\,.$$

   We then rewrite equation 5.6 to obtain:

$$\mathsf{BPC}(\pi) := \bigwedge_{v \in \mathrm{V}(\pi)} \mathsf{E}_{\mathsf{cd}}(v) \wedge \bigwedge \Phi_\pi \wedge \bigwedge_{e \in \mathrm{E}(\pi) \cap D} \Phi(e) \tag{5.7}$$

Since we interpret Boolean path conditions over $\mathbf{A}'_{\mathsf{IMP}}$ where $\cdot^{\mathbf{A}'_{\mathsf{IMP}}} \circ == ^{\mathbf{A}'_{\mathsf{IMP}}}$ behaves identically to $\equiv$ for expressions of type `int` we can immediately simplify the formula by identifying x_i and x_j whenever we add a Φ constraint of the form $(x_i == x_j)$.

Unfortunately, SSA form is not sufficient to generate correct path conditions.

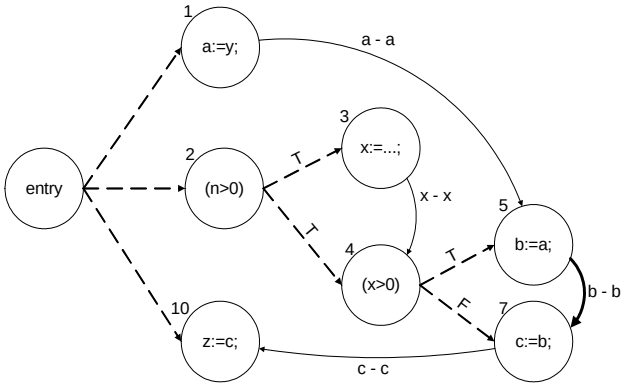
Example 6 (SSA form is not sufficient)
Consider, for example, the program [Kri03] and its PDG in figure 5.2:

This program fragment is already in SSA form, i. e. we do not need to include Φ constraints, but if we compute $\mathsf{BPC}(1, 10)$ we obtain

$$\begin{aligned}\mathsf{BPC}(1,10) &= \mathrm{true} \wedge ((n > 0) \wedge (x > 0)) \wedge ((n > 0) \wedge !(x > 0)) \wedge \mathrm{true} \\ &\leftrightarrow (n > 0) \wedge (x > 0) \wedge !(x > 0) \leftrightarrow \mathrm{false}\end{aligned}$$

Figure 5.2 Program fragment in SSA from with a loop carried flow dependence and its PDG.

```
 1  a := y;
 2  while (n > 0) {
 3      x := ...;
 4      if (x > 0) {
 5          b := a;
 6      } else {
 7          c := b;
 8      }
 9  }
10  z := c;
```



Obviously, $\mathsf{BPC}(1, 10)$ is incorrect. The reason for this is that the flow dependence edge $5 \xrightarrow[b]{\mathsf{fd}} 7$ is loop carried and $x > 0$ and $!(x > 0)$ refer to different runtime instances of variable `x`.

One solution to this [Kri03] is to separate all variables whenever a path passes along a loop carried edge. Let $\pi : s \xrightarrow[\mathrm{PDG}_p]{}\!\!^{*}\ t$ be a PDG path such that $\pi = (\pi_1, e_1, \pi_2, e_2, \ldots, e_{n-1}, \pi_n)$ where, for $1 \leq i \leq n$, π_i is a subpath of π that does not contain loop carried data dependence edges, i. e. $\mathrm{L}^c(\mathrm{E}(\pi_i)) = \{\, \emptyset \,\}$, and, for $1 \leq j < n$, e_j is a loop carried data dependence edge. Then we set

$$\mathsf{BPC}(\pi) := \bigwedge_{i=1}^{n} \mathsf{BPC}(\pi_i, i) \tag{5.8}$$

where $\mathsf{BPC}(\pi_j, j)$ is the path condition for π_j from equation 5.7 where every typed variable $x \in \mathcal{V}(\mathsf{BPC}(\pi_j))$ has been replace by x_j, a new instance of x.

In the example above, $\mathsf{BPC}(1, 10)$ becomes [Kri03]:

$$\mathsf{BPC}(1, 10) = \mathrm{true} \wedge ((n_1 > 0) \wedge (x_1 > 0)) \wedge ((n_2 > 0) \wedge !(x_2 > 0)) \wedge \mathrm{true}$$

Robschink [Rob04] proposes another option: Instead of separating the variables we can also replace all execution constraints that cause a contradiction by true. Suppose we generate a path condition $\mathsf{BPC}(\pi)$ for a cycle-free path $\pi = e_1, e_2, \ldots, e_n : s \xrightarrow[\mathrm{PDG}_p]{}\!\!^{*}\ t$ and we have already generated $\mathsf{BPC}(\pi_i)$ for $\pi_i := e_i, e_{i+1}, \ldots, e_n$ $(i > 1)$.[8] Let $\theta := \mathsf{E}_{\mathsf{cd}}\big(\odot\!\!\succ(e_{i-1})\big) \wedge \mathsf{BPC}(\pi')$. If we have not $\mathsf{E}_{\mathsf{cd}}\big(\odot\!\!\succ(e_{i-1})\big) \nleftrightarrow \mathrm{false}$ and not $\mathsf{BPC}(\pi') \nleftrightarrow \mathrm{false}$, we set

$$\mathsf{BPC}(e_{i-1}, \pi_i) := \begin{cases} \mathsf{BPC}(\pi_i) & \text{if } \theta \nleftrightarrow \mathrm{false} \\ \theta & \text{otherwise} \end{cases} \tag{5.9}$$

[8] Here we assume that we have not yet added Φ constraints to $\mathsf{BPC}(\pi_i)$, i. e. $\mathsf{BPC}(\pi_i)$ has been generated as described in equation 5.4 (p. 36). The Φ constraints are included in a second generation step.

else we have $\mathsf{BPC}(e_{i-1}, \pi_i) := \text{false}$.

In the example above we obtain with this approach

$$\mathsf{BPC}(1, 10) = \text{true} \wedge (n > 0) \wedge !(x > 0)$$

5.5 Arrays

We now allow p to use arrays. Then, not all paths in the PDG from s to t are paths for potential information flow. Suppose, for example, that the last edge in such a path is a def-def dependence edge. Then, there is no influence along this path since the value which is "reused" in the last node does not affect the array cell that is accessed. Therefore, we restrict ourselves to information flow paths, i. e. the disjunction in equation 5.5 (p. 37) now ranges only over all information flow paths between s and t.

Definition 33 (Information flow path)
A path $\pi : s \xrightarrow[\text{PDG}_p]{}\!\!\!\!\blacktriangleright^* t$ is called an **information flow path** iff for every def-def dependence edge $v \xrightarrow[x]{\mathsf{dd}\triangleright} w \in \mathrm{E}(\pi)$ there is a successor edge $\mathrm{succ}(v \xrightarrow[x]{\mathsf{dd}\triangleright} w)$ in π which is a flow or def-def dependence edge with respect to $\mathtt{x}$.[9] Let $\Pi(s, t)$ denote the set of all information flow paths from s to t.

From now on, "path" in the PDG always means information flow path in the PDG unless we state it differently.

For Boolean path conditions, the execution and Φ conditions are still valid, but now, we add extra constraints to make the conditions more precise.

Let $\pi : s \xrightarrow[\text{PDG}_p]{}\!\!\!\!\blacktriangleright^* t$ be a cycle-free path in the PDG from s to t and let $\rho := v_0 \xrightarrow[a]{\mathsf{dd}\triangleright} v_1, \ldots, v_{n-2} \xrightarrow[a]{\mathsf{dd}\triangleright} v_{n-1}, v_{n-1} \xrightarrow[a]{\mathsf{fd}\triangleright} v_n$ be a maximal subpath of this form in π.

The δ constraint for ρ [Rob04] is defined as

$$\delta_\pi(\rho) := \bigvee_{j=0}^{n-1} \left(\bigwedge_{k=j+1}^{n-1} (i_j != i_k) \wedge (i_j == i_n) \right) \tag{5.10}$$

where i_j is the array index expression for array a in node v_j for $0 \leq j \leq n$.

Let $\Pi_A(\pi)$ denote the set of all maximal subpaths of the above form in π. Then, we can extend equation 5.7 to

$$\mathsf{BPC}(\pi) := \bigwedge_{v \in \mathrm{V}(\pi)} \mathsf{E}_{\mathsf{cd}}(v) \wedge \bigwedge \Phi_\pi \wedge \bigwedge_{e \in \mathrm{E}(\pi) \cap D} \Phi(e) \wedge \bigwedge_{\rho \in \Pi_A(\pi)} \delta_\pi(\rho). \tag{5.11}$$

Here, we assume that π does not contain loop-carried data dependence edges. If π does, we have to combine the δ constraints with the options from above to address the problems created by these edges.

Note that we generate array conditions only for flow dependence edges $v \xrightarrow[a]{\mathsf{fd}\triangleright} w$, not for def-def dependences.

[9] Here, we require only that the successor edge is a data dependence edge with respect to the program variable $\mathtt{x}$. In general, when we consider x to encode the two occurrences of $\mathtt{x}$ in p, these $x$es will be different.

Example 7 (δ constraints for arrays)

```
1 x := b;
2 a[i] := x;
3 a[1] := 5;
4 y := a[1];
```

If we look at lines 1 and 4 we see that there is only one path

$$\pi := 1 \xrightarrow[x]{\mathsf{fd}\triangleright} 2, 2 \xrightarrow[a]{\mathsf{dd}\triangleright} 3, 3 \xrightarrow[a]{\mathsf{fd}\triangleright} 4$$

from line 1 to line 4 in the PDG.

If we compute the δ constraint for the maximal subpath $\rho := 2 \xrightarrow[a]{\mathsf{dd}\triangleright} 3, 3 \xrightarrow[a]{\mathsf{fd}\triangleright} 4$, we get

$$\delta_\pi(\rho) = (i! = 1) \wedge (i == 1) \vee (1 == 1) \tag{5.12}$$

Robschink [Rob04] also suggests another type of δ constraints for arrays: global array constraints. They differ from the above in that they subsume all maximal def-def paths preceding a flow dependence edge $v \xrightarrow[a]{\mathsf{fd}\triangleright} w$ for array variable a, not only the one on the path that is currently examined. Let $(\rho_\lambda)_{\lambda\in\Lambda}$ denote the family of all maximal cycle-free paths

$$\rho_\lambda := v_0^\lambda \xrightarrow[a]{\mathsf{dd}\triangleright} v_1^\lambda, v_1^\lambda \xrightarrow[a]{\mathsf{dd}\triangleright} v_2^\lambda, \ldots, v_{k_\lambda-1}^\lambda \xrightarrow[a]{\mathsf{dd}\triangleright} v_{k_\lambda}^\lambda, v_{k_\lambda}^\lambda \xrightarrow[a]{\mathsf{fd}\triangleright} w \tag{5.13}$$

that are subpaths of information flow paths $\pi_\lambda \in \Pi(s,t)$ where $v_{k_\lambda}^\lambda = v$. We define the global array constraint for $v \xrightarrow[a]{\mathsf{fd}\triangleright} w$ to be

$$\delta_G(v \xrightarrow[a]{\mathsf{fd}\triangleright} w) := \bigvee_{\lambda\in\Lambda} \delta_{\pi_\lambda}(\rho_\lambda)\,. \tag{5.14}$$

Let A_a denote the set of all flow dependence edges $v \xrightarrow[a]{\mathsf{fd}\triangleright} w \in \bigcup \mathrm{E}(\Pi(s,t))$ with respect to array variable a.

Equation 5.5 (p. 37) can then be rewritten as

$$\mathsf{BPC}(s,t) := \left(\bigvee_{\pi\in\Pi(s,t)} \mathsf{BPC}(\pi)\right) \wedge \left(\bigvee_{a\in V_p, (\!|a|\!)=\texttt{array}} \bigvee_{v \xrightarrow[a]{\mathsf{fd}\triangleright} w\in A_a} \delta_G(v \xrightarrow[a]{\mathsf{fd}\triangleright} w)\right) \tag{5.15}$$

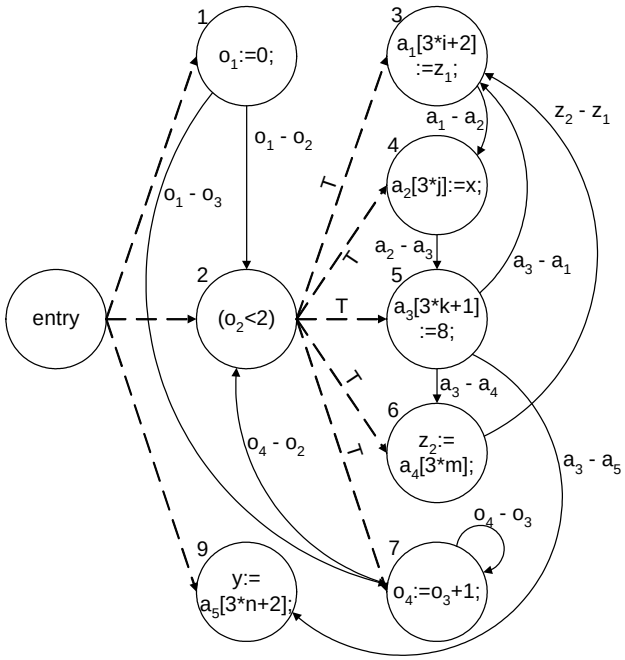
where $\mathsf{BPC}(\pi)$ is as in equation 5.7 (p. 37).

Example 8 (Global array constraints)
Consider the program given in figure 5.3. For $5 \xrightarrow[a_3-a_5]{\mathsf{fd}\triangleright} 9$, we obtain the global array constraint as

$$\begin{aligned}\delta_G(5 \xrightarrow[a]{\mathsf{fd}\triangleright} 9) =& ((3*i+2! = 3*j) \wedge (3*i+2! = 3*k+1) \wedge \\ & (3*i+2 == 3*n+2)) \vee ((3*j! = 3*k+1) \wedge \\ & (3*j == 3*n+2)) \vee (3*k+1 == 3*n+2)\end{aligned}$$

Figure 5.3 Example program for global array constraints and its PDG.

```
1  o := 0;
2  while (o < 2) {
3      a[3 * i + 2] := z;
4      a[3 * j] := x;
5      a[3 * k + 1] := 8;
6      z := a[3 * m];
7      o := o + 1;
8  }
9  y := a[3 * n + 2];
```



5.6 Handling cycles in information flow paths

Unfortunately, the disjunction in equation 5.5 (p. 37) may range over infinitely many paths, e. g. if the PDG contains a cycle which is reachable from s and from which we can reach t. Thus, it is necessary to generate path conditions that capture a whole class of paths in the PDG. Obviously, cycles in the PDG always contain a loop-carried data dependence edge. With Boolean path conditions we can separate the variable instances or omit some constraints that would make the (unevaluated) condition false: Whenever we can insert a cycle ρ in a path $\pi = (\pi^*, \pi') \in \Pi(s,t)$ so that $(\pi^*, \rho, \pi') \in \Pi(s,t)$, we distinguish the variables in the part of the condition that belongs to π^* from those in its part that belongs to π'.

Let $\Pi^{**}(s,t)$ denote the set of all cycle-free paths from s to t in PDG_p. Then, we have to generate path conditions only for influence paths in $\Pi^{**}(s,t)$ and take the disjunction over them:[10]

$$\mathsf{BPC}(s,t) := \bigvee_{\pi \in \Pi^{**}(s,t)} \mathsf{BPC}(\pi) \tag{5.16}$$

The next example shows that we must use global array constraints to obtain correct path conditions.

[10] In [SRK], Snelting, Robschink and Krinke present more sophisticated techniques how to reduce the number of paths for which one has to generate path conditions and how to generate shorter formulae by exploiting the structure of the paths and distributivity of the $\vee$ and $\wedge$ operators.

Example 9
Let us revisit example 8 from above. For Boolean path conditions, we only want to consider cycle-free paths. There is only one cycle-free information flow path in $\Pi^{**}(4,10)$, namely $\pi := 4 \xrightarrow[a_2-a_3]{\mathsf{dd}} 5,5 \xrightarrow[a_2-a_5]{\mathsf{fd}} 9$.

If we used the path condition as described in equation 5.11 (p. 39), we would obtain (without Φ constraints):

$$\begin{aligned}\mathsf{BPC}(4,9) =&(o_2^{(1)} < 2) \wedge (o_2^{(2)} < 2) \wedge \\ &(((3 * j^{(1)}! = 3 * k^{(1)} + 1) \wedge (3 * j^{(1)}! = 3 * k^{(2)} + 1) \wedge \\ &(3 * j^{(1)} == 3 * n^{(2)} + 2)) \vee \\ &(3 * k^{(2)} + 1 == 3 * n^{(2)} + 2)) \\ \nleftrightarrow\ &\text{false}\end{aligned}$$

Clearly, this path condition is wrong because for $j = m$ and $i = n$ the value of x in line 4 is transferred to y in line 9 via

$$pi_2 := 4 \xrightarrow[a_2-a_3]{\mathsf{dd}} 5,5 \xrightarrow[a_3-a_4]{\mathsf{fd}} 6,6 \xrightarrow[z_2-z_1]{\mathsf{fd}} 3,3 \xrightarrow[a_1-a_2]{\mathsf{dd}} 4,4 \xrightarrow[a_2-a_3]{\mathsf{dd}} 5,5 \xrightarrow[a_3-a_5]{\mathsf{fd}} 9.$$

If we use global array conditions instead, we get (without Φ constraints):

$$\begin{aligned}\mathsf{BPC}(4,9) =&(o_2^{(1)} < 2) \wedge (o_2^{(2)} < 2) \wedge ((3 * i^{(1)} + 2! = 3 * j^{(1)}) \wedge \\ &(3 * i^{(1)} + 2! = 3 * k^{(1)} + 1) \wedge (3 * i^{(1)} + 2! = 3 * k^{(2)} + 1) \wedge \\ &(3 * i^{(1)} + 2 == 3 * n^{(2)} + 2) \vee (3 * j^{(1)}! = 3 * k^{(1)} + 1) \wedge \\ &(3 * j^{(1)}! = 3 * k^{(2)} + 1) \wedge (3 * j^{(1)} == 3 * n^{(2)} + 2) \vee \\ &(3 * k^{(2)} + 1 == 3 * n^{(2)} + 2)) \\ \nleftrightarrow&(o_2^{(1)} < 2) \wedge (o_2^{(2)} < 2) \wedge (3 * i^{(1)} + 2! = 3 * j^{(1)}) \wedge \\ &(3 * i^{(1)} + 2! = 3 * k^{(1)} + 1) \wedge (3 * i^{(1)} + 2! = 3 * k^{(2)} + 1) \wedge \\ &(3 * i^{(1)} + 2 == 3 * n^{(2)} + 2)\end{aligned}$$

which is obviously satisfiable.

Chapter 6

Temporal path conditions

The program dependence graph contains both data and control dependences within a program. Slicing [FOW87, Kri03] as reachability analysis in the PDG exploits this property to find all statements of a program which can influence a specified node or be influenced by it. This approach can therefore answer the question whether a statement s can possibly influence some other statement t. However, slices can be only a conservative approximation and, in practice, they become very large, i. e. imprecise: In most cases, this approximation is too conservative. Snelting [Sne96] proposed to consider all possible paths in the PDG from s to t and generate a Boolean formula over the program variables which is a necessary condition for any such path being executed. If this formula is not satisfiable, then no influence is possible, even though the backward slice of t contains s. See chapter 5 for more details on Boolean path conditions. Here, we generalize this idea to generating necessary LTL conditions.

This chapter is organized as follows: First, we define the state sequences that can be generated by an IMP program over the structure $\mathbf{A}_{\mathsf{IMP}}$. Next, we present different types of constraints that are then combined to generate an LTL for an information flow path in the PDG. In section 6.3, we show how to capture a whole class of paths in the PDG with a single LTL influence condition. Next, some lemmata for simplifying influence conditions are presented before we provide some examples to justify why we have used some constructions that may be not intuitively clear in the beginning. We conclude this chapter with a comparison between Boolean and LTL path conditions where we see that LTL path conditions are stronger than Boolean path conditions.

6.1 From an IMP program to state sequences

In chapter 3, we said an LTL formula is satisfiable iff there exists an infinite sequence of states which is a model for it. However, there are infinitely many state sequences, most of which are not related to the program for which we have generated the LTL formula. Since we are interested in whether there is a program execution that satisfies the LTL formula, we now define how to obtain the set of state sequences of interest from an IMP program. We use the language $\mathcal{L}_{\mathsf{IMP}}$ to write atomic formulae in and the structure $\mathbf{A}_{\mathsf{IMP}}$ to interpret them over.

6.1.1 Transition graphs

In a state sequence for an IMP program, we want to have states only for assignment statements because it is only them who change the program variables' values. Control flow statements such as `if` and `while` do not contribute directly to the program state but decide based on the program variables' values which state is the next one. We use the transition graph for a program as an intermediate representation from which we can easily construct state sequences for a program.

Definition 34 (Transition graph)
Let $p \in \mathsf{IMP}$ be an IMP program, let $\mathrm{CFG}_p = (V \cup \{ v_x \}, E, v_e, v_x, \alpha)$ be the CFG of p, and $\mathrm{PDG}_p = (V, C, D, v_e, \overline{\alpha}, \mathrm{L})$ the PDG of p. The **transition graph** for p is the multigraph graph $TG_p = (V', E', \odot\!\!\succ, -\!\!\!\not\odot, \lambda)$ where

- $V' := \{ v \in V \mid \mathbf{def}(v) \neq \emptyset \} \cup \{ v_e \}$ is the set of assignment nodes plus the entry node,
- E' is the set of edges where $(v, \pi, w) \in E'$ iff $v \in V'$, $w \in V' \cup \{ v_x \}$ and $\pi : v \xrightarrow[\mathrm{CFG}_p]{}\!\!\!\!\blacktriangleright^{*} w$ is a cycle-free CFG path such that $u \notin V'$ for all $u \in \mathrm{V}(\pi)$ with $v \prec_\pi u \prec_\pi w$. We call an edge $(v, \pi, w) \in E'$ with $w \neq v_x$ **transition edge** and an edge $(v, \pi, v_x) \in E'$ **idling edge**.
- For $(v, \pi, w) \in E'$ we set

$$\odot\!\!\succ((v, \pi, w)) := v \text{ and } -\!\!\!\not\odot((v, \pi, w)) := \begin{cases} w & \text{if } w \in V' \\ v & \text{if } w = v_x \end{cases}.$$

- $\lambda : E \mapsto \mathsf{SF}^{\mathcal{L}_{\mathsf{IMP}}}$ is the labelling function for edges that assigns every edge $e \in E'$ a state formula over program variables as a guard:

$$\lambda((v, \pi, w)) := \bigwedge_{(u,u') \in \mathrm{E}(\pi)} \begin{cases} \mathrm{true} & \text{if } \alpha((u, u')) = \epsilon \\ \cdot(u) & \text{if } \alpha((u, u')) = \mathrm{true} \\ \cdot(!(u)) & \text{if } \alpha((u, u')) = \mathrm{false} \end{cases}.$$

The guards are meant as follows: If we have a program state, i. e. an assignment ξ to the program variables, in some node v of the transition graph, then for every outgoing edge e of v, we check whether $\xi(\lambda(e)) = \mathtt{T}$ holds. If so, the target node w of e is a possible successor node to v in state ξ. If we do pass along e, then ξ must be updated: Suppose $w \neq v_x$. Let `x := e;` be the statement that corresponds to w. Then, the new state ψ is given by $\psi(\mathtt{x}) := \xi(\mathtt{e})$ and $\psi(\mathtt{y}) := \xi(\mathtt{y})$ for all program variables `y` other than x. If e is an idling edge, then ξ itself is the successor state to ξ.

6.1.2 State sequences over $\mathcal{L}_{\mathsf{IMP}}$

Above, we have already hinted at how to generate state sequences from a transition graph: Start with an arbitrary assignment to program variables and follow an

infinite path through the transition graph whose edges' guards are all satisfied by the correctly updated program states. However, in temporal path conditions, we want to easily refer to a variable's former values in a given state without using past operators. Thus, we enlarge the set of variables we are interested in:

Definition 35 (Program variables)
Let $p \in \mathsf{IMP}$ be an IMP program and let $\mathrm{PDG}_p = (V, C, D, v_e, \overline{\alpha}, \mathrm{L})$ be the program dependence graph for p. The set $\mathbb{V}_p$ of **program variables** for p is

$$\begin{aligned}\mathbb{V}_p &:= \bigcup_{v \in V} (\mathbf{def}(v) \uplus \mathbf{use}(v)) \\ &= \bigcup_{v \in V} ((\mathbf{def}(v) \times \{\, \mathbf{def} \,\}) \cup (\mathbf{use}(v) \times \{\, \mathbf{use} \,\})) \times \{\, v \,\}\,.\end{aligned}$$

The set V_p of program variable names is

$$V_p := \{\, y \mid \exists a \in \{\, \mathbf{def}, \mathbf{use} \,\} : \exists v \in V : (y, a, v) \in \mathbb{V}_p \,\}\,.^{11} \tag{6.1}$$

Concerning the variables' types, we set $(\!|(x, a, v)|\!) :=$ type of variable x.

This definition allows us to distinguish all variables by the PDG node where they are accessed and by whether the access is a read or a write. We may, however, want to have extra variables at our disposal when generating formulae, e. g. to bind a value to a variable for future use. Hence, we do not exactly specify the set of all program variables $\mathbb{V}$, we require just $\mathbb{V} \supseteq \mathbb{V}_p \cup V_p$ for the IMP program p of interest.

Definition 36 (Rigid and flexible variables)
We distinguish between rigid and flexible variables [MP91] in $\mathbb{V}$. A **rigid** variable $y \in \mathbb{V}$ is a variable which can not change its value within a state sequence $\Xi = (\xi_i)_{i \in \mathbb{N}}$, i. e. $\xi_i(y) = \xi_{i+1}(y)$ for all $i \in \mathbb{N}$. If a variable is not rigid, we say it is **flexible**.

By convention, all program variables $\mathbb{V}_p$ are flexible. Unless we state it explicitly, we assume that all variables mentioned are flexible.

The set of states as defined in section 3.2.3 is then $\mathcal{S}_{\mathbb{V}}^{\mathbf{A}_{\mathsf{IMP}}}$. However, not all of these are suitable to being used as initial states. For example, it is not sensible to have a `int` variable initialized to $\perp_i$ because in an implementation of IMP, we would not have this special value. Hence, the set of initial states for a program p is given by

$$\mathcal{IS}_p := \left\{\, \xi \in \mathcal{S}_{\mathbb{V}}^{\mathbf{A}_{\mathsf{IMP}}} \;\middle|\; \xi(\mathbb{V} - \mathbb{V}_p) \cap \perp = \emptyset \wedge \xi(\mathbb{V}_p) \subseteq \perp \,\right\}.$$

For the set of state sequences $\mathcal{M}_{\mathbb{V}}^{\mathbf{A}_{\mathsf{IMP}}}$ we consider only state sequences that respect the rigidity condition for rigid variables, i. e., if $\mathbb{V}^r \subseteq \mathbb{V} - (\mathbb{V}_p \cup V_p)$ denotes

[11] In this definition, we consider the use sets to be generated with ordinary instead of disjoint union.

the set of rigid variables,[12]

$$\mathcal{M}_{\mathbb{V}}^{\mathbf{A}_{\mathsf{IMP}}} := \left\{ \Xi = (\xi_i)_{i\in\mathbb{N}} \in \left(\mathcal{S}_{\mathbb{V}}^{\mathbf{A}_{\mathsf{IMP}}}\right)^{\mathbb{N}} \;\middle|\; \forall y \in \mathbb{V}^r : \forall i \in \mathbb{N} : \xi_i(y) = \xi_{i+1}(y) \right\}.$$

From this set of sequences, we define a subset $\mathcal{M}_p \subseteq \mathcal{M}_{\mathbb{V}}^{\mathbf{A}_{\mathsf{IMP}}}$ which contains all sequences that can be generated by program $p \in \mathsf{IMP}$. We construct state sequences using the transition graph $TG_p = (V, E, \odot\!\!\succ, -\!\!\!\times\!\!\odot, \lambda)$ of p. Let $\xi_0 \in \mathcal{IS}_p$ be an initial state for p. Let $v_0 = v_e$ be the entry node of the CFG $\mathrm{CFG}_p = (V', E', v_e, v_x, \alpha)$ of p. We recursively define the successor states ξ_{i+1} of ξ_i and successor nodes v_{i+1} of v_i. Suppose that ξ_i is a program state in node v_i. Then, there exists at least one outgoing edge $e := (v_i, \pi, w) \in E$ of v_i such that $\xi_i(\lambda(e)) = \mathtt{T}$. Pick any such $e = (v_i, \pi, w)$. Suppose e is a transition edge and suppose $\pi = e_1, \ldots, e_n$. Set $\psi_0 = \xi_i$. For $1 \le j \le n$, define ψ_j recursively by

$$\begin{aligned}
\psi_j(y) &:= \psi_{j-1}(y) \text{ for all } y \in \mathbb{V}^r \cup V_p,\\
\psi_j((y,a,v)) &:= \psi_{j-1}((y,a,v)) \text{ for all } (y,a,v) \in \mathbb{V}_p \text{ such that } v \neq -\!\!\!\times\!\!\odot(e_j),\\
\psi_j((y,a,v)) &:= \psi_{j-1}(y) \text{ for all } (y,a,v) \in \mathbb{V}_p \text{ such that } v = -\!\!\!\times\!\!\odot(e_j).
\end{aligned}$$

Then, ξ_{i+1} is given by

$$\begin{aligned}
\xi_{i+1}(y) &:= \begin{cases} \psi_n(y) & \text{if } y \notin \mathbf{def}(w) \\ y' & \text{otherwise} \end{cases} \quad \text{for } y \in V_p \\
\xi_{i+1}((y,a,v)) &:= \begin{cases} \psi_n((y,a,v)) & \text{if } y \notin \mathbf{def}(w) \\ y' & \text{otherwise} \end{cases} \quad \text{for } (y,a,v) \in \mathbb{V}_p \qquad (6.2)\\
\xi_{i+1}(y) &:= \xi_i(y) \text{ for } y \in \mathbb{V}^r \\
\xi_{i+1}(y) &\in (\!|y|\!)^{\mathbf{A}_{\mathsf{IMP}}} \text{ arbitrary for } y \in \mathbb{V} - (\mathbb{V}_p \cup \mathbb{V}^r \cup V_p)
\end{aligned}$$

where

- for $(\!|y|\!) \in \{\, \mathtt{bool}, \mathtt{int} \,\}$ we set

  $$y' := e_w^{\mathcal{A}_{\mathsf{IMP}}}(\xi_i(y_1), \ldots, \xi_i(y_n))$$

  where e_w is the assignment expression for variable y in node w over program variables $y_1, \ldots, y_n \in V_p$ with variable ordering $y_1 \sqsubseteq \ldots \sqsubseteq y_n$.

- for $(\!|y|\!) = \mathtt{array}$ we set

  $$y' := j \mapsto \begin{cases} \bot_i & \text{if } j = \bot_i \\ q & \text{if } j = e^{\mathbf{A}_{\mathsf{IMP}}}(\xi_i(y_1), \ldots, \xi_i(y_n)) \\ \xi_i(y)(j) & \text{otherwise} \end{cases}$$

  where $q = e'^{\mathbf{A}_{\mathsf{IMP}}}(\xi_i(y_1), \ldots, \xi_i(y_n))$ is the value of the assignment expression e' over program variables $y_1, \ldots, y_n \in V_p$ with variable ordering $y_1 \sqsubseteq \ldots \sqsubseteq y_n$ and e is the array cell expression over program variables $y_1, \ldots, y_n$ with variable ordering $y_1 \sqsubseteq \ldots \sqsubseteq y_n$).

Then, the set $\mathcal{M}_p$ of state sequences for p is the set of all such $(\xi_i)_{i\in\mathbb{N}}$.

[12] Note that this restriction on $\mathcal{M}_{\mathbb{V}}^{\mathbf{A}_{\mathsf{IMP}}}$ does not affect the validity of lemma 6 (p. 22).

6.2 Generating LTL formulae for paths

Slicing uses the influence criterion to determine which statements can possibly be influenced. The forward slice for a statement s contains all PDG nodes that are reachable from s via an information flow path, the backward slice for s all PDG nodes from which s is reachable via an information flow path, i. e. the forward slice for s contains all statements that can be influenced by s, the backward slice all those that can influence s. Given two statements s and t, the chop for s and t is the PDG subgraph generated by the intersection of forward slice for s and backward slice for t. Obviously, if the chop for s and t is empty, s cannot influence t, and any influence from s to t must happen along some information flow path in the chop for s and t.

However, not all paths in a chop can happen during program execution, because reachability search in the PDG can not find contradicting conditions over variables.

Like Snelting, Krinke and Robschink did for Boolean path conditions in [Sne96, RS02, Rob04, SRK], we now want to generate a necessary LTL condition for a given path from s to t. Thereby, we distinguish three different kinds of dependence constraints:

1. **Execution conditions**: These conditions are derived for every statement on the path from the control dependences in the PDG. They express necessary constraints for the statement to be executed.

2. **Data dependence conditions**: These conditions are derived for every data dependence edge in the path. They formulate conditions for information flowing along the edge.

3. **Intrastatement conditions**: These conditions impose constraints on the single-statement level to ensure that the value that is passed along a flow dependence edge can actually influence the target node's evaluation.

We first present every kind on its own. Thereafter, all these conditions are combined in a single LTL formula by Boolean and temporal connectives. Just as we distinguish every occurrence of a program variable in terms of states and state sequences do we use these different variants of a program variable in the LTL condition. To keep the examples simple and readable, we index all occurrences for every program variable with ascending numbers.

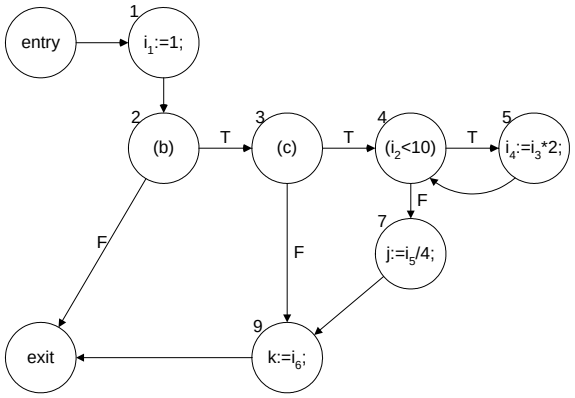
6.2.1 Execution conditions

Clearly, for a statement s to be executed, it is necessary that all control conditions on at least one path from the entry node v_e to s in CDG_p are satisfied the last time they have been executed before s is being executed [Sne96]. This forms one part of the execution conditions we consider.

Secondly, like in Hoare logic [Hoa69], if we know that there is a loop l that directly precedes, but does not contain s, then the loop predicate does not hold

Figure 6.1 Example program for execution conditions and the program's CFG.

```
 1  i := 1;
 2  if (b) {
 3      if (c) {
 4          while (i < 10) {
 5              i := i * 2;
 6          }
 7          j := i / 4;
 8      }
 9      k := i;
10  }
```



when s is executed. We adapt this idea to our setting to complete the execution condition for s.

For the first type of execution conditions, we use the execution conditions from control dependence as defined in 5.2. Note that the Boolean absorption, associativity, and idempotence laws also hold for LTL formulae (cf. table 3.1 (p. 16)). Hence, we can also always restrict ourselves to cycle-free control dependence paths from the entry node to the node of interest when we generate execution conditions.

6.2.1.1 Loop predicates in execution conditions

Suppose we consider a statement s that is preceded by the loop predicate l, but that is not inside the loop body of l. Then, we know that the loop must have terminated when the control flow reach s, i. e. the loop predicate does not hold in the state corresponding to s. Hence, it is safe to include this extra constraint in the execution condition of s.

Definition 37 (Execution condition from loop predicates)
Let $s \in V$ be a node in $\mathrm{CFG}_p = (V, E, v_e, v_x, \alpha)$ and let $l \in V$ be a loop predicate node with $l \neq s$. If l dominates s ($l \,\mathsf{DOM}\, s$) and s is not control dependent on l (not $l \xrightarrow[p]{\mathsf{cd}}\!\!\triangleright^{*}\ s$) we say l **must terminate** before s. We write $\mathrm{L}^t(s)$ for the set of all such l for node s.

The **execution condition from loop predicates** for s is given by

$$\mathsf{E}_{\circlearrowleft}(s) := \bigwedge_{l \in \mathrm{L}^t(s)} \cdot(!(l))$$

If $\mathrm{L}^t(s) = \emptyset$, we set $\mathsf{E}_{\circlearrowleft}(s) := \mathrm{true}$.

Definition 38 (Execution condition)
For every node $s \in V - \{ v_e, v_x \}$ in $\mathrm{CFG}_p = (V, E, v_e, v_x)$, the **execution condition** for s is given by

$$\mathsf{E}(s) := \mathsf{E}_{\mathsf{cd}}(s) \wedge \mathsf{E}_{\circlearrowleft}(s)$$

Line l	$\mathsf{E}_{\mathsf{cd}}(l)$	$\mathsf{E}_{\circlearrowleft}(l)$
1	true	true
2	true	true
3	(b)	true
4	$(b) \wedge (c)$	true
5	$(b) \wedge (c) \wedge (i_2 < 10)$	true
7	$(b) \wedge (c)$	$!(i_2 < 10)$
9	(b)	true

Table 6.1: Execution conditions for the program in figure 6.1.

Example 10
Let us consider the example program p in figure 6.1. Table 6.1 lists the execution conditions for every node of the CFG. Note that for line 7 (`j := i / 4;`) we have $\mathsf{E}_{\circlearrowleft}(7) = !(i_2 < 10)$ since the loop predicate node 4 dominates 7, but 7 is not control dependent on 4, whereas for line 9 (`k := i;`) we do not know whether $i_2 < 10$ holds or not because the loop predicate 4 does not dominate node 9. Hence we have $\mathsf{E}_{\circlearrowleft}(9) = \text{true}$.

In all examples that follow we will always simplify execution conditions before we include them in other formulae. Although, strictly speaking, it is not correct to use the equals sign $=$ (we ought to use $\Longleftrightarrow$ or $\nrightarrow$ instead), we will nevertheless use the former since for the rest it does not matter if we use simplified formulae or not, unless we state it differently.

6.2.2 Data dependence conditions

While execution conditions are computed for every node on a path in the PDG, data dependence conditions are generated to describe the flow along edges in the path. There are two types of edges: control dependence edges and data dependence edges. We do not generate any conditions for control dependence edges as they are already taken care of in the execution conditions of the source and target statements. On the other hand, flow and def-def dependence edges can generate four types of conditions:

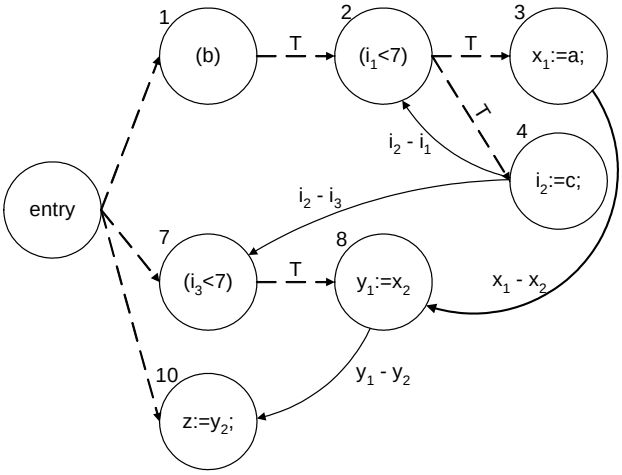
- The Φ constraint for a data dependence edge e ensures that the variable's value that is written in the source node of e arrives at the target node of e. We use the same Φ constraints for data dependence edges as we did for Boolean path conditions (cf. section 5.3 for details) except for equation 5.3 where we use instead

$$^{v}\Phi_{x}^{w} := (x_v[y] == x_w[y])$$

  where y is the rigid variable from equations 6.4 (p. 52), 6.5 (p. 52), and 6.6 (p. 52) (see below). If e is a flow dependence edge with $\circledcirc\!\!\rightarrowtail(e) = \rightarrowtail\!\!\circledcirc(e)$, we use true as the Φ constraint for e. For examples, see examples 20 (p. 78) and 21 (p. 79).

Figure 6.2 Program with flow dependence 3 $\xrightarrow[x_1-x_2]{\text{fd}}$ 8 which leaves the loop with loop predicate node 2.

```
 1  if (b) {
 2      while (i < 7) {
 3          x := a;
 4          i := c;
 5      }
 6  }
 7  if (i < 7) {
 8      y := x;
 9  }
10  z := y;
```



- Loop termination conditions for data dependence edges introduce constraints for loops terminating. They are very similar to execution conditions from loop predicates (cf. section 6.2.1.1).
- Execution conditions can be computed not only for nodes in the path under consideration, but also for all nodes on all CFG paths carrying the flow dependence. In other words, this type defines conditions for all states between the source state and the target state.
- If the flow dependence $v \xrightarrow[a]{\text{fd}} w$ involves an array variable `a`, we want to make sure that we formulate conditions as strong as possible to ensure that the flow of information actually happens, i. e. we want to have that the array cell read is the same as the one written before. Hence, this type will specify conditions which extend their scope both on the source and target state of the edge.

6.2.2.1 Loop termination conditions

In 6.2.1.1 we have seen that we can include negated loop predicates in execution conditions. However, there are cases when we know that in some state we must have left a loop l but the execution condition of that node does not contain the negated loop predicate. Suppose we have a flow (def-def) dependence $v \xrightarrow[x]{\text{fd}} w$ ($v \xrightarrow[x]{\text{dd}} w$) such that $\mathrm{L}^x\left(v \xrightarrow[x]{\text{fd}} w\right) \neq \emptyset$ ($\mathrm{L}^x\left(v \xrightarrow[x]{\text{dd}} w\right) \neq \emptyset$). Then, we know that every loop $u \in \mathrm{L}^x\left(v \xrightarrow[w]{\text{fd}} x\right)$ ($u \in \mathrm{L}^x\left(v \xrightarrow[w]{\text{dd}} x\right)$) must have terminated between v and w.

Example 11
Consider, for example, the program p and its PDG shown in figure 6.2. We see that the flow dependence edge 3 $\xrightarrow[x_1-x_2]{\text{fd}}$ 8 leaves the loop with loop predicate node 2. Hence we know that if we have a state sequence that carries information

along $3 \xrightarrow[x_1-x_2]{\mathsf{fd}}\triangleright 8$, there is a state ξ corresponding to node 8 in it and we have $\xi(!(i_1 < 7)) = \mathtt{T}$, i. e. $\xi(i_1) \geq 7$. From the execution condition of node 8 we know that $(i_3 < 7)$ holds in ξ, so $\xi(i_3) < 7$. If we look at the program at more detail, we easily see that $\xi(i_3) = \xi(i_1)$. Therefore there can be no information flow along $3 \xrightarrow[x_1-x_2]{\mathsf{fd}}\triangleright 8$.

Note that the execution condition of node 8 is just $(i_3 < 7)$. The extra piece of information $!(i_1 < 7)$ is not part of this condition because the while loop is nested in a conditional statement. Hence, it is sensible to have loop termination conditions both in execution conditions and for data dependence edges.

Definition 39 (Loop termination condition)
Let $e := v \xrightarrow[x]{\mathsf{fd}}\triangleright w$ ($e := v \xrightarrow[x]{\mathsf{dd}}\triangleright w$) be a flow (def-def) dependence. The **loop termination condition** along e is defined as:

$$\delta_l(e) := \bigwedge_{u \in \mathrm{L}^x(e)} \cdot(!(u)).$$

If $\mathrm{L}^x(e) = \emptyset$, we set $\delta_l(e) := \text{true}$.

6.2.2.2 Execution conditions along flow dependence edges

Let us look at a flow (def-def) dependence edge $v \xrightarrow[x]{\mathsf{fd}}\triangleright w$ ($v \xrightarrow[x]{\mathsf{dd}}\triangleright w$) from v to w with respect to variable x. Let Π denote the set of all CFG paths $v \xrightarrow[\mathrm{CFG}]{}{}^{*}\blacktriangleright w$ carrying the dependence. Since we do not want to generate execution conditions which connect execution conditions for every single node on every path with temporal operators because the formulae would become very long, or maybe even infinitely long, we do generate a joint execution condition that is satisfied by all nodes on all paths in Π.

Definition 40
Let $V_\Pi := \{\, v \in \bigcup \mathrm{V}(\Pi) \mid \mathbf{def}(v) \neq \emptyset \,\} \cup \{\, w \,\}$ be the set of nodes of interest[13] for the flow (def-def) dependence edge $v \xrightarrow[x]{\mathsf{fd}}\triangleright w$ ($v \xrightarrow[x]{\mathsf{dd}}\triangleright w$). We define

$$\mathsf{E}_\delta\big(v \xrightarrow[x]{\mathsf{fd}}\triangleright w\big) := \bigvee_{v \in V_\Pi} \mathsf{E}(v) \text{ and } \mathsf{E}_\delta\big(v \xrightarrow[x]{\mathsf{dd}}\triangleright w\big) := \bigvee_{v \in V_\Pi} \mathsf{E}(v)\,. \tag{6.3}$$

In programs with well-structured control flow (where the control dependence graph is a tree) we can always find a node $u \in V$ such that $\mathsf{E}(v') \rightarrowtail \mathsf{E}(u)$ for all $v' \in V_\Pi$, if we want to. For example, if $v \xrightarrow[x]{\mathsf{fd}}\triangleright w$ is loop-carried by loops $\mathrm{L}(v \xrightarrow[w]{\mathsf{fd}}\triangleright x)$, then the outermost loop $u \in \mathrm{L}(v \xrightarrow[w]{\mathsf{fd}}\triangleright x)$ satisfies $\mathsf{E}(v') \rightarrowtail \mathsf{E}(u)$ for all $v' \in V_\Pi$.

Thus, we can often reduce the execution condition for a flow dependence to an ordinary execution condition. Since the condition is necessary for all nodes in V_Π, we will use the $\mathcal{U}$ operator to insert this constraint.

[13] State sequences contain only states for assignment statements, so we do not have to care about control statements. However, we do not know whether w is an assignment statement. Hence, we explicitly include w in V_Π.

6.2.2.3 Data dependence edge conditions

Suppose we have a path ρ from $v_0 := v$ to w of the form

$$v_0 \xrightarrow[x]{\mathsf{dd}\triangleright} v_1, \ldots, v_{n-1} \xrightarrow[x]{\mathsf{dd}\triangleright} v_n, v_n \xrightarrow[x]{\mathsf{fd}\triangleright} w$$

for some $n \in \mathbb{N}$. Then, x is an array variable and this path expresses a potential flow of information from v to w via $v_1, \ldots, v_n$ at which this information is not destroyed.

We now want to generate an LTL formula that expresses that $v_0, \ldots, v_n, w$ are executed in this order and that the information in x written in v does reach w. This is accomplished by the following scheme $\delta(\rho, \eta)$

$$\delta(\rho, \eta) := (y == i_0) \wedge \delta'(y, \rho, \eta) \tag{6.4}$$

where $\eta \in \mathsf{LTL}^{\mathcal{L}_{\mathsf{IMP}}}$ is an arbitrary LTL formula over $\mathcal{L}_{\mathsf{IMP}}$, $y \in \mathbb{V}^r - \mathcal{V}(\eta)$[14] arbitrary, but rigid, i_0 is the index expression for array variable x in node v_0, and δ' is the scheme defined by

$$\begin{aligned}\delta'(y, u_1 \xrightarrow[x]{\mathsf{fd}\triangleright} u_2, \eta) := \mathsf{E}_\delta\big(u_1 \xrightarrow[x]{\mathsf{fd}\triangleright} u_2\big)\, \mathcal{U}\, ((y == i_2) \wedge \delta_l\big(u_1 \xrightarrow[x]{\mathsf{fd}\triangleright} u_2\big) \wedge \\ {}^{u_1}\Phi_x^{u_2} \wedge \eta)\end{aligned} \tag{6.5}$$

and

$$\begin{aligned}\delta'(y, (u_1 \xrightarrow[x]{\mathsf{dd}\triangleright} u_2, \rho'), \eta) := \mathsf{E}_\delta\big(u_1 \xrightarrow[x]{\mathsf{dd}\triangleright} u_2\big)\, \mathcal{U}\, ((y! = i_2) \wedge \delta_l\big(u_1 \xrightarrow[x]{\mathsf{dd}\triangleright} u_2\big) \wedge \\ {}^{u_1}\Phi_x^{u_2} \wedge \mathsf{E}(u_2) \wedge \delta'(y, \rho', \eta))\end{aligned} \tag{6.6}$$

where i_2 represents the index expression for array variable x in node u_2, ρ' is a nonempty information flow path, and ${}^{u_1}\Phi_x^{u_2}$ is the Φ condition for $u_1 \xrightarrow[x]{\mathsf{fd}\triangleright} u_2$ or $u_1 \xrightarrow[x]{\mathsf{dd}\triangleright} u_2$.

Intuitively, $\delta(\rho, \eta)$ generates the complete path condition for ρ, which consists of execution and loop termination conditions and Φ constraints, and includes η such that η must hold in the last state of the state sequence that is constrained by the path condition for ρ. For non-array dependence edges, we give the rules for combining the different types of constraints in 6.2.4. We have already included the complete generation rule here, because the rigid variable y relates all states on ρ.

6.2.3 Intrastatement conditions

Execution conditions on the one hand are purely local, i. e. independent of the path we are looking at. Flow dependence conditions, on the other hand, are constraints that deal with what happens between the source and target node of a flow dependence edge. Intrastatement conditions express constraints concerning what happens with a value that is passed along a flow dependence condition once it has arrived at the target node.

Suppose we are generating a path condition for a path that contains a flow dependence edge $v \xrightarrow[x]{\mathsf{fd}\triangleright} w$. However, whether the value of `x` in w indeed influences w, i. e. the evaluation result of w, may depend on other input parameters of w.

[14] Note that we require that y does not occur in $\mathcal{V}(\eta)$ nor in any formula we substitute for any propositional variable in η.

Operator f	$p_{f,1}$	$p_{f,2}$
$+, \cdot[\cdot], ==, !=, <, >$	true	true
$-, !, \cdot$	true	
$*, /$	$(x_2! = 0)$	$(x_1! = 0)$
$\leq$	$(x_2! = 2147483647)$	$(x_1! = -2147483648)$
$\geq$	$(x_2! = -2147483648)$	$(x_1! = 2147483647)$
&&	x_2	x_1
$\|\|$	$!(x_2)$	$!(x_1)$

Table 6.2: Predicates $p_{f,i}$ for function and predicate operators in $\mathcal{L}_{\mathsf{IMP}}$.

Example 12 (Intrastatement conditions)
Consider, for example, the following program fragment:

```
1  x := 5;
2  if (0 * x == 12) {
3      y := 7;
4  }
```

Clearly, line 1 cannot influence line 3, even though the latter is control dependent on the `if` predicate in line 2 which itself is flow dependent on line 1. The expression `(0*x==12)` always evaluates to false, no matter what the value of x is, because the function $i \mapsto (*(a,b))^{\mathbf{A}_{\mathsf{IMP}}}(0,i)$, $i \in \overline{\mathbb{Z}}$ (for the expression $*(a,b) \in \mathfrak{E}$ with variable ordering $a \sqsubseteq b$) is constant.

We use the symbols $\perp_a$, $\perp_b$, and $\perp_i$ only for uninitialized variables. Since by definition of state sequences, every variable is initialized before it is used we may restrict interpreted function operators to non-lifted values. In the example, we can say that the result of function $(a,b) \mapsto *(a,b)^{\mathbf{A}_{\mathsf{IMP}}}(a,b)$, $(a,b) \in \overline{\mathbb{Z}}^2$ only depends on b if $a \neq 0$.

More generally, for every n-ary function or predicate operator $f \in \mathcal{F}_n \cup \mathcal{R}_n$ ($n \in \mathbb{N} - \{0\}$), i. e. we consider the expression or atomic formula $e := f(x_1, \ldots, x_n)$ with variable ordering $x_1 \sqsubseteq \ldots \sqsubseteq x_n$, and for every parameter $i \in \{1, \ldots, n\}$ we define a predicate $p_{f,i} \in \mathsf{SF}^{\mathcal{L}_{\mathsf{IMP}}}$ with variable ordering $x_1 \sqsubseteq \ldots \sqsubseteq x_{i-1} \sqsubseteq x_{i+1} \sqsubseteq \ldots \sqsubseteq x_n$) such that for all $(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n) \in A_1 \times \ldots \times A_{i-1} \times A_{i+1} \times \ldots \times A_n$ (where $A_j := \tau^{-1}(\llbracket f \rrbracket_j) - \perp$ for $1 \leq j \leq n$) if $a \mapsto e^{\mathbf{A}_{\mathsf{IMP}}}(a_1, \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_n)$, $a \in A_i$, is not constant then

$$p_{f,i}^{\mathbf{A}_{\mathsf{IMP}}}(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n) = \mathtt{T}.^{15}$$

For IMP, we define the $p_{f,i}$s in table 6.2. With these predicates in mind, we define intrastatement conditions next:

Definition 41 (Intrastatement condition)
Given a statement w and a variable $x \in \mathbf{use}(w)$ which is contained in the maximal expression $e \in \mathfrak{E}$ with variable ordering $x_1 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq x$, an **intrastate-**

[15] We can do something analogous for atomic formulae of the form $e_1 \equiv e_2$, too. However, as we do not make use of $\equiv$ in LTL path conditions, we do not give a definition for this case here.

ment dependence condition δ_x^w is a state formula $\delta_x^w \in \mathsf{SF}^{\mathcal{L}_{\mathsf{IMP}}}$ with variable ordering $x_1 \sqsubseteq \ldots \sqsubseteq x_n$ such that for all $(a_1, \ldots, a_n) \in \prod_{i=1}^{n} \left((\!| x_i |\!)^{\mathbf{A}_{\mathsf{IMP}}} - \bot \right)$ if there are $a, b \in (\!| x |\!)^{\mathbf{A}_{\mathsf{IMP}}} - \bot$ with $e^{\mathbf{A}_{\mathsf{IMP}}}(a_1, \ldots, a_n, a) \neq e^{\mathbf{A}_{\mathsf{IMP}}}(a_1, \ldots, a_n, b)$ then $(\delta_x^w)^{\mathbf{A}_{\mathsf{IMP}}}(a_1, \ldots, a_n) = \mathtt{T}$.

For IMP, we choose the following intrastatement conditions: We first define recursively intrastatement conditions δ_x^e for expressions $e \in \mathfrak{E}$. Let e be an expression and $x \in \mathbf{use}(e)$.

- If e itself is a variable, we set $\delta_x^e := \text{true}$.
- If $e = f(e_1, \ldots, e_n)$ for some $n \in \mathbb{N}$, $f \in \mathcal{F}_n$, and $e_1, \ldots, e_n \in \mathfrak{E}$, let $j \in \{ 1, \ldots, n \}$ be the unique index of the expression e_j with $x \in \mathbf{use}(e_j)$. We then set

$$\delta_x^e := p_{f,j}[e_1/x_1, \ldots, e_{j-1}/x_{j-1}, e_{j+1}/x_{j+1}, \ldots, e_n/x_n] \wedge \delta_x^{e_j}$$

Let $e \in \mathfrak{E}$ be the expression to evaluate in w that contains the use of variable x. We then set $\delta_x^w := \delta_x^e$. When we include intrastatement conditions in examples, we feel free to simplify them before that just as we do with execution conditions.

6.2.4 Putting everything together

Above, we have defined several types of constraints that we want to include in path conditions. We now combine them to generate an LTL formula $\mathsf{PC}(\pi)$ for the whole path $\pi : s \xrightarrow[\mathrm{PDG}_p]{}\!\!\!\!\blacktriangleright^{*} t$. A **path condition** for path π in PDG_p is an LTL formula $\mathsf{PC}(\pi)$ that is a necessary condition for the data flow between $\odot\!\!\succ(\pi)$ and $\prec\!\!\odot(\pi)$ to happen along π. Necessary in this context means that if this data flow does happen there is a sequence of states Ξ in $\mathcal{M}_p$ such that $\Xi \vDash \Diamond\, \mathsf{PC}(\pi)$.

Let $\mathrm{PDG}_p = (V, C, D, v_e, \overline{\alpha}, \mathrm{L})$ be the PDG for p and let $\pi : s \xrightarrow[\mathrm{PDG}_p]{}\!\!\!\!\blacktriangleright^{*} t$ be an information flow path in PDG_p. We define $\mathsf{PC}(\pi)$ recursively:

- If $|\pi| = 0$, i. e. $\pi = t$ is an empty path, we set

$$\mathsf{PC}(\pi) := \mathsf{E}(t)\,. \tag{6.7}$$

- If $|\pi| > 0$ and $\pi = e, \pi'$ for some control dependence edge $e = (s, v) \in C$, we set

$$\mathsf{PC}(\pi) := \mathsf{E}(s) \wedge \mathsf{PC}(\pi'). \tag{6.8}$$

- If $|\pi| > 0$ and $\pi = s \xrightarrow[x]{\mathsf{fd}}\!\!\triangleright\ v, \pi'$ for some $s \xrightarrow[x]{\mathsf{fd}}\!\!\triangleright\ v \in D$ and x is a scalar variable, we set

$$\mathsf{PC}(\pi) := \mathsf{E}(s) \wedge \mathsf{E}_\delta\big(s \xrightarrow[x]{\mathsf{fd}}\!\!\triangleright\ v\big)\ \mathcal{U}\ \big(\delta_l\left(s \xrightarrow[x]{\mathsf{fd}}\!\!\triangleright\ v\right) \wedge {}^{s}\Phi_x^v \wedge \delta_x^v \wedge \mathsf{PC}(\pi')\big) \tag{6.9}$$

- If $|\pi| > 0$ and $\pi = \rho, \pi'$ for some

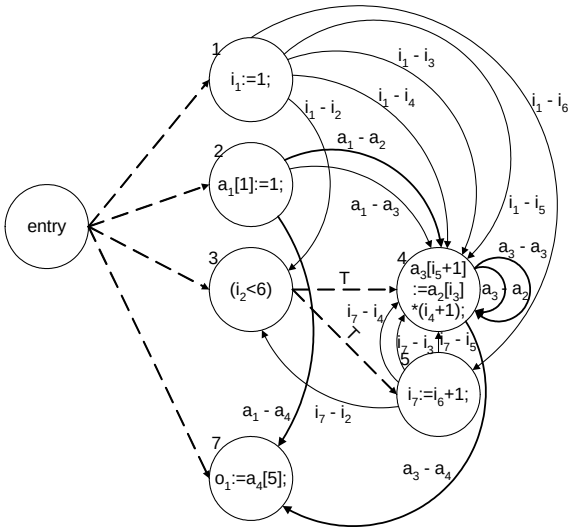
$$\rho = v_0 \xrightarrow[x]{\mathsf{dd}}\!\!\triangleright\ v_1, v_1 \xrightarrow[x]{\mathsf{dd}}\!\!\triangleright\ v_2, \ldots, v_{n-2} \xrightarrow[x]{\mathsf{dd}}\!\!\triangleright\ v_{n-1}, v_{n-1} \xrightarrow[x]{\mathsf{fd}}\!\!\triangleright\ v_n,$$

some $n \in \mathbb{N} - \{ 0 \}$, an array variable x, and $\mathrm{E}(\rho) \subseteq D$, we set

$$\mathsf{PC}(\pi) := \mathsf{E}(v_0) \wedge \delta(\rho, \delta_x^{v_n} \wedge \mathsf{PC}(\pi')) \tag{6.10}$$

Figure 6.3 A simple program for computing the factorial function and its PDG.

```
1  i:=1;
2  a[1]:=1;
3  while (i<6) {
4     a[i+1]:=a[i]*(i+1);
5     i:=i+1;
6  }
7  o:=a[5];
```



Since s need not be the entry node v_e whereas all sequences in $\mathcal{M}_p$ start in v_e, if we want to decide whether this path is feasible in $\mathcal{M}_p$ we need to check whether there is a sequence $\Xi \in \mathcal{M}_p$ such that $\Xi \vDash \Diamond\, \mathsf{PC}(\pi)$.

6.2.5 A simple example

Example 13

In figure 6.3 we now consider the program, which computes the factorial function and stores the values in an array variable. Its PDG is shown in the same figure. We are interested in whether there is an information flow from line 2 (`a[1]:=1;`) to line 7 (`o:=a[5];`). We now look at different paths from node 2 to node 7.

The simplest path π, which consists only of the flow dependence edge $2 \xrightarrow[a_1-a_4]{\mathsf{fd}}\triangleright$ 7 from node 2 to 7 labelled $a_1 - a_4$, generates the path condition

$$\mathsf{PC}(\pi) = \underbrace{\mathsf{true}}_{=\mathsf{E}(2)} \wedge \overbrace{(y == 1) \wedge \underbrace{\mathsf{true}}_{=\mathsf{E}_\delta\left(2 \xrightarrow[a_1-a_4]{\mathsf{fd}}\triangleright 7\right)} \mathcal{U}\, ((y == 5) \wedge \underbrace{\mathsf{true}}_{=\delta_l\left(2 \xrightarrow[a_1-a_4]{\mathsf{fd}}\triangleright 7\right)} \wedge \overbrace{\underbrace{(a_1[y] == a_4[y])}_{=^2\Phi^7_{a_1-a_4}} \wedge \underbrace{\mathsf{true}}_{=\delta^7_{a_4}} \wedge \underbrace{!(i_2 < 6)}_{=\mathsf{E}(7)}}^{=\delta(2 \xrightarrow[a_1-a_4]{\mathsf{fd}}\triangleright 7, \delta^7_{a_4} \wedge \mathsf{E}(7))}}$$

where y is a rigid variable. We can simplify $\mathsf{PC}(\pi)$ to

$$\mathsf{PC}(\pi) \Longleftrightarrow (y == 1) \wedge \Diamond((y == 5) \wedge (a_1[y] == a_4[y]) \wedge !(i_2 < 6))$$

Since y is a rigid variable, we also have

$$\mathsf{PC}(\pi) \Longleftrightarrow \Diamond((y == 1) \wedge (y == 5) \wedge (a_1[y] == a_4[y]) \wedge !(i_2 < 6)) \Longleftrightarrow \text{false}\,,$$

so $\mathsf{PC}(\pi)$ is not satisfiable.

Next, let us look at another path

$$\pi' := 2 \xrightarrow[a_1-a_2]{\mathsf{fd}} 4,4 \xrightarrow[a_3-a_2]{\mathsf{fd}} 4,4 \xrightarrow[a_3-a_3]{\mathsf{dd}} 4,4 \xrightarrow[a_3-a_4]{\mathsf{fd}} 7.$$

We obtain

$$\begin{aligned}
\mathsf{PC}(\pi') = {} & \underbrace{\mathsf{true}}_{=\mathsf{E}(2)} \wedge (y_1 == 1) \wedge \underbrace{\mathsf{true}}_{=\mathsf{E}_\delta\left(2 \xrightarrow[a_1-a_2]{\mathsf{fd}} 4\right)} \mathcal{U}\, ((y_1 == i_3) \wedge \underbrace{\mathsf{true}}_{=\delta_l\left(2 \xrightarrow[a_1-a_2]{\mathsf{fd}} 4\right)} \wedge \\
& \underbrace{(a_1[y_1] == a_2[y_1])}_{{}^2\Phi^4_{a_1-a_2}} \wedge \underbrace{(i_4 + 1! = 0)}_{=\delta^4_{a_2}} \wedge \underbrace{(i_2 < 6)}_{=\mathsf{E}(4)} \wedge (y_2 == i_5 + 1) \wedge \\
& \underbrace{(i_2 < 6)}_{=\mathsf{E}_\delta\left(4 \xrightarrow[a_3-a_2]{\mathsf{fd}} 4\right)} \mathcal{U}\, ((y_2 == i_3) \wedge \underbrace{\mathsf{true}}_{\delta_l\left(4 \xrightarrow[a_3-a_2]{\mathsf{fd}} 4\right)} \wedge \underbrace{\mathsf{true}}_{={}^4\Phi^4_{a_3-a_2}} \wedge \\
& \underbrace{(i_4 + 1! = 0)}_{=\delta^4_{a_2}} \wedge \underbrace{(i_2 < 6)}_{=\mathsf{E}(4)} \wedge (y_3 == i_5 + 1) \wedge \\
& \underbrace{(i_2 < 6)}_{=\mathsf{E}_\delta\left(4 \xrightarrow[a_3-a_3]{\mathsf{dd}} 4\right)} \mathcal{U}\, ((y_3! = i_5 + 1) \wedge \underbrace{\mathsf{true}}_{=\delta_l\left(4 \xrightarrow[a_3-a_3]{\mathsf{dd}} 4\right)} \wedge \underbrace{(a_3[y_3] == a_3[y_3])}_{={}^4\Phi^4_{a_3-a_3}} \wedge \\
& \underbrace{(i_2 < 6)}_{=\mathsf{E}(4)} \wedge \underbrace{((i_2 < 6) \vee !(i_2 < 6))}_{=\mathsf{E}_\delta\left(4 \xrightarrow[a_3-a_4]{\mathsf{fd}} 7\right)}{}^{16}\, \mathcal{U}\, ((y_3 == 5) \wedge \underbrace{!(i_2 < 6)}_{=\delta_l\left(4 \xrightarrow[a_3-a_4]{\mathsf{fd}} 7\right)} \wedge \\
& \underbrace{(a_3[y_3] == a_4[y_3])}_{={}^4\Phi^7_{a_3-a_4}} \wedge \underbrace{\mathsf{true}}_{=\delta^7_{a_4}} \wedge \underbrace{!(i_2 < 6)}_{=\mathsf{E}(7)}))))
\end{aligned}$$

where y_1, y_2, and y_3 are rigid variables. We can simplify $\mathsf{PC}(\pi')$ to

$$\begin{aligned}
\mathsf{PC}(\pi') \Longleftrightarrow {} & (y_1 == 1) \wedge \Diamond((y_1 == i_3) \wedge (a_1[y_1] == a_2[y_1]) \wedge (i_4 + 1! = 0) \wedge \\
& (y_2 == i_5 + 1) \wedge (i_2 < 6)\, \mathcal{U}\, ((y_2 == i_3) \wedge (i_4 + 1! = 0) \wedge \\
& (y_3 == i_5 + 1) \wedge (i_2 < 6)\, \mathcal{U}\, ((y_3! = i_5 + 1) \wedge (a_3[y_3] == a_3[y_3]) \wedge \\
& (i_2 < 6) \wedge ((i_2 < 6) \vee !(i_2 < 6))\, \mathcal{U}\, ((y_3 == 5) \wedge !(i_2 < 6) \wedge \\
& (a_3[y_3] == a_4[y_3])))))
\end{aligned}$$

Note however, that $\mathsf{PC}(\pi')$ is not satisfiable over $\mathcal{M}_p$ because the only information flow from line 2 to line 7 happens along the path

$$\rho = 2 \xrightarrow[a_1-a_2]{\mathsf{fd}} 4,4 \xrightarrow[a_3-a_2]{\mathsf{fd}} 4,4 \xrightarrow[a_3-a_2]{\mathsf{fd}} 4,4 \xrightarrow[a_3-a_2]{\mathsf{fd}} 4,4 \xrightarrow[a_3-a_2]{\mathsf{fd}} 4,4 \xrightarrow[a_3-a_4]{\mathsf{fd}} 7$$

[16] Note that in general not $(i_2 < 6) \vee !(i_2 < 6) \nleftrightarrow \mathsf{true}$. Consider for example a state $\xi \in \mathcal{S}_{\mathbb{V}}^{\mathbf{A}_{\mathsf{IMP}}}$ with $\xi(i_2) = \bot_i$, then $\xi(i_2 < 6) = \mathtt{F} = \xi(!(i_2 < 6))$ and $\xi(\mathsf{true}) = \mathtt{T}$.

for which we obtain

$$\begin{aligned}
&\mathsf{PC}(\rho) \Longleftrightarrow \\
&(y_1 == 1) \wedge \Diamond((y_1 == i_3) \wedge (a_1[y_1] == a_2[y_1]) \wedge (i_4 + 1! = 0) \wedge \\
&(y_2 == i_5 + 1) \wedge (i_2 < 6)\, \mathcal{U}\, ((y_2 == i_3) \wedge (i_4 + 1! = 0) \wedge \\
&(y_3 == i_5 + 1) \wedge (i_2 < 6)\, \mathcal{U}\, ((y_3 == i_3) \wedge (i_4 + 1! = 0) \wedge \\
&(y_4 == i_5 + 1) \wedge (i_2 < 6)\, \mathcal{U}\, ((y_4 == i_3) \wedge (i_4 + 1! = 0) \wedge \\
&(y_5 == i_5 + 1) \wedge (i_2 < 6)\, \mathcal{U}\, ((y_5 == i_3) \wedge (i_4 + 1! = 0) \wedge \\
&(i_2 < 6) \wedge (y_6 == i_5 + 1) \wedge ((i_2 < 6) \vee !(i_2 < 6))\, \mathcal{U}\, ((y_5 == 5) \wedge !(i_2 < 6) \wedge \\
&(a_3[y_6] == a_4[y_6])))))))
\end{aligned} \tag{6.11}$$

where y_1, y_2, y_3, y_4, y_5, and y_6 are rigid. Note that we have generated some subformulae multiple times to capture the repeated flow along edge $4 \xrightarrow[a_3 - a_2]{\mathsf{fd}} 4$. Since, in most cases, we do not know in advance the correct number of flows along such an edge, we would have to generate formulae for all possible paths. Even though this issue is decidable in our finite state setting, we are not willing to generate formulae that large.

6.3 Influence conditions

We want to use path conditions to gain information about whether and how a statement s can possibly influence a statement t in a program p. As we have seen above, we can generate necessary conditions for every information flow path from s to t in the PDG for p. Since every influence happens along an information flow path in the PDG of p between s and t, if we take the disjunction over conditions for all such paths, we obtain a necessary condition for the influence to happen.

Definition 42 (Influence condition between statements)
Let s and t be two nodes in the PDG for program $p \in \mathsf{IMP}$. The influence condition (first version) for s and t is then given by

$$\mathsf{IC}(s,t) := \bigvee_{\pi \in \Pi(s,t)} \mathsf{PC}(\pi) \tag{6.12}$$

Unfortunately, in most cases, the disjunction will be infinite, because loops in the PDG generate infinitely many paths. We address this problem by restricting $\Pi(s,t)$ to a finite number of paths and by modifying the generation rules for $\mathsf{PC}(\pi)$ such that we still generate a necessary condition for the influence. Our aim is to restrict $\Pi(s,t)$ to cycle-free paths. However, we will encounter a case where we need to weaken this objective to cycle-disjoint paths.

6.3.1 Extended rules for generating path conditions

In order to write transformation rules which can be applied to the path condition of a given path and modify it to incorporate the changes we want to make, we introduce extra propositional variables in path conditions. This is why we replace

the rules defined in equations 6.5 (p. 52), 6.6 (p. 52), 6.7 (p. 54), 6.8 (p. 54), and 6.9 (p. 54) by the following ones (in this order):

1. Let ${}^{u_1}\beth_x^{u_2} \in \mathcal{W}$ be a propositional variable which we identify with the dependence edge $u_1 \xrightarrow[x]{\mathsf{fd}\triangleright} u_2$ or $u_1 \xrightarrow[x]{\mathsf{dd}\triangleright} u_2$. More precisely, if we generate a path condition for path π and we come along the edge $u_1 \xrightarrow[x]{\mathsf{fd}\triangleright} u_2$ or $u_1 \xrightarrow[x]{\mathsf{dd}\triangleright} u_2$, we use a new variable ${}^{u_1}\beth_x^{u_2}$. If π passes through this edge multiple times, we use different propositional variables each time.[17]

$$\begin{aligned}\delta'(y,u_1 \xrightarrow[x]{\mathsf{fd}\triangleright} u_2,\eta) := (\mathsf{E}_\delta\big(u_1 \xrightarrow[x]{\mathsf{fd}\triangleright} u_2\big) \wedge {}^{u_1}\beth_x^{u_2})\, \mathcal{U}\, (\\ (y == i_2) \wedge \delta_l\big(u_1 \xrightarrow[x]{\mathsf{fd}\triangleright} u_2\big) \wedge {}^{u_1}\Phi_x^{u_2} \wedge \eta)\end{aligned} \tag{6.13}$$

2. Let ${}^{u_1}\beth_x^{u_2} \in \mathcal{W}$ be as before.

$$\begin{aligned}\delta'(y,(u_1 \xrightarrow[x]{\mathsf{dd}\triangleright} u_2,\rho),\eta) := (\mathsf{E}_\delta\big(u_1 \xrightarrow[x]{\mathsf{dd}\triangleright} u_2\big) \wedge {}^{u_1}\beth_x^{u_2})\, \mathcal{U}\, (\\ (y! = i_2) \wedge \delta_l\big(u_1 \xrightarrow[x]{\mathsf{dd}\triangleright} u_2\big) \wedge {}^{u_1}\Phi_x^{u_2} \wedge \mathsf{E}(u_2) \wedge \delta'(y,\rho,\eta))\end{aligned} \tag{6.14}$$

3. Let $\beth \in \mathcal{W}$ be a propositional variable not used before. (We will always use $\beth$ as the propositional variable at the end of an information flow path.)

$$\mathsf{PC}(\pi) := \mathsf{E}(t) \wedge \beth \tag{6.15}$$

4. Let $\beth_e \in \mathcal{W}$ be a propositional variable that we identify with the control dependence edge e like before.

$$\mathsf{PC}(\pi) := \mathsf{E}(s) \wedge \beth_e \wedge \mathsf{PC}(\pi') \tag{6.16}$$

5. Let ${}^{s}\beth_x^{v} \in \mathcal{W}$ be as above.

$$\mathsf{PC}(\pi) := \mathsf{E}(s) \wedge (\mathsf{E}_\delta\big(s \xrightarrow[x]{\mathsf{fd}\triangleright} v\big) \wedge {}^{s}\beth_x^{v})\, \mathcal{U}\, (\delta_l\big(s \xrightarrow[x]{\mathsf{fd}\triangleright} v\big) \wedge {}^{s}\Phi_x^{v} \wedge \delta_x^{v} \wedge \mathsf{PC}(\pi')) \tag{6.17}$$

These additional propositional variables enable us to use both substitution of propositional variables (cf. definition 8 (p. 11)) and of subformulae (cf. definition 9 (p. 12)) to modify such a formula.

Even though $\overline{\mathsf{PC}(\pi)}$[18] is not the same formula as the original one, one can easily see that they are congruent. In particular, we have that if (π^*, π') is an information flow path in PDG_p and both π^* and π' are themselves information flow paths, then

$$\mathsf{PC}((\pi^*, \pi')) \Longleftrightarrow \mathsf{PC}(\pi^*)[\mathsf{PC}(\pi')/\beth].$$

Moreover, none of the generation rules uses the operators $\neg$ and $\rightarrow$, so no propositional variable occurs negatively in path conditions of the above form.

[17] We identify ${}^{u_1}\beth_x^{u_2}$ with the occurrence of $u_1 \xrightarrow[x]{\mathsf{fd}\triangleright} u_2$ or $u_1 \xrightarrow[x]{\mathsf{dd}\triangleright} u_2$. If we then generate path conditions for subpaths of π, we want to use the same propositional variables that we (would) have used to generate the path condition for the whole path.

[18] As defined in section 3.2.2, the bar over an LTL formula means that we substitute true for all propositional variables in the formula.

6.3.2 Eliminating cycles in paths

Our most urgent task is to avoid the infinite disjunction of LTL formulae which is still possible with influence conditions as in definition 42 (p. 57). We distinguish three types of cycles:

- Cycles composed only of control dependence edges,
- Cycles all of whose data dependence edges are flow dependence edges with respect to scalar variables, and
- Cycles that contain non-scalar data dependence edges.

We will address them in this order.

6.3.2.1 Control dependence cycles

Although in PDGs for IMP programs control dependence edges form a tree and thus cannot themselves generate cycles in the PDG, they could easily be dealt with in a more general setting:

Lemma 8 (Control dependence cycles)
Let $\pi : s \xrightarrow[\mathrm{PDG}_p]{}\!\!\!\blacktriangleright^* t$ *be an information flow path from* s *to* t *that contains a cycle*

$$\rho := (v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$$

of control dependence edges. Let π^* *denote the subpath of* π *before this cycle and* π' *the subpath of* π *after it. Then* $(\pi^*, \pi') \in \Pi(s, t)$ *and* $\mathsf{PC}(\pi) \Longrightarrow \mathsf{PC}((\pi^*, \pi'))$.

Proof. The claim $(\pi^*, \pi') \in \Pi(s, t)$ is obvious. For ρ, π' we have $\mathsf{PC}((\rho, \pi')) \Longleftrightarrow \bigwedge_{i=0}^{n-1} \left(\mathsf{E}(v_i) \wedge \beth_{(v_i, v_{i+1})}\right) \wedge \mathsf{PC}(\pi')$. Since $\aleph \wedge \beth \Longrightarrow \beth$ (cf. table 3.2 (p. 17)), we have

$$\bigwedge_{i=0}^{n-1} \left(\mathsf{E}(v_i) \wedge \beth_{(v_i, v_{i+1})}\right) \wedge \mathsf{PC}(\pi') \Longrightarrow \mathsf{PC}(\pi').$$

Therefore,

$$\mathsf{PC}(\pi) \Longleftrightarrow \mathsf{PC}(\pi^*)[\mathsf{PC}((\rho, \pi'))/\beth] \Longrightarrow \mathsf{PC}(\pi^*)[\mathsf{PC}(\pi')/\beth] \Longleftrightarrow \mathsf{PC}((\pi^*, \pi')).$$

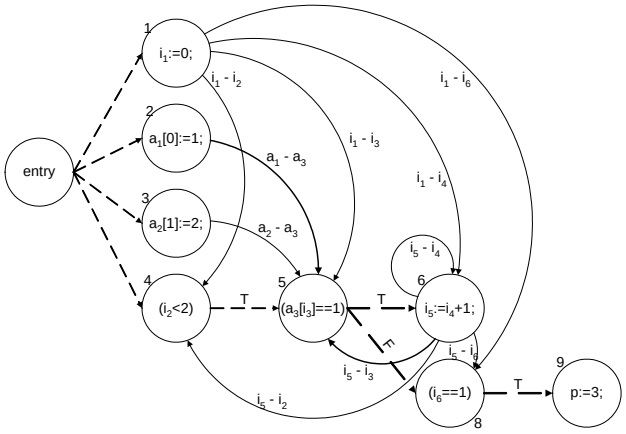
□

6.3.2.2 Scalar flow dependence cycles

Next, we consider loops $e_1, \ldots, e_n$ in $\pi : s \xrightarrow[\mathrm{PDG}_p]{}\!\!\!\blacktriangleright^* t$ where each e_i is either a control dependence edge (v_{i-1}, v_i) or a flow dependence edge $v_{i-1} \xrightarrow[x_i]{\mathsf{fd}}\!\!\triangleright\ v_i$ with respect to the scalar variable x_i, i. e. $(\!|x_i|\!) \in \{\, \mathtt{int}, \mathtt{bool} \,\}$, for $1 \leq i \leq n$, and $v_n = v_0$. The next lemma shows that we can get rid of this type of cycles by introducing an additional until operator.

Figure 6.4 An example program and its PDG to demonstrate the until operator being necessary for handling cycles with scalar flow dependences and its PDG.

```
 1  i:=0;
 2  a[0]:=1;
 3  a[1]:=2;
 4  while (i<2) {
 5      if (a[i]==1) {
 6          i:=i+1;
 7      } else {
 8          if (i==1) {
 9              p:=3;
10          }
11      }
12  }
```



Lemma 9 (Scalar flow dependence cycles)
Let $\pi \in \Pi(s,t)$ be an information flow path from s to t in PDG_p such that $\varrho := e_1, \ldots, e_n$ is a cycle in π where e_i is either a control dependence edge (v_{i-1}, v_i) or a flow dependence edge $v_{i-1} \xrightarrow[x_i]{fd}\!\triangleright\, v_i$ with respect to a scalar variable x_i $(1 \le i \le n)$, and $v_n = v_0$. Let π^* denote the subpath of π before ϱ and π' the subpath of π after ϱ. Let

$$E_\varrho := \bigvee_{\substack{v \in \mathrm{V}(\varrho) \\ \mathbf{def}(v) \neq \emptyset}} \mathsf{E}(v) \vee \bigvee_{v \xrightarrow[x]{fd}\!\triangleright\, w \in \mathrm{E}(\varrho) \cap D} \mathsf{E}_\delta\big(v \xrightarrow[x]{fd}\!\triangleright\, w\big)$$

denote the joint execution condition for all nodes and dependence edges in ϱ. Then $(\pi^*, \pi') \in \Pi(s,t)$, and

$$\mathsf{PC}(\pi) \rightarrowtail\!\!\!\Longrightarrow \mathsf{PC}(\pi^*)[E_\varrho \,\mathcal{U}\, (\mathsf{PC}(\pi'))/\beth]$$

and

$$\mathsf{PC}((\pi^*, \pi')) \rightarrowtail\!\!\!\Longrightarrow \mathsf{PC}(\pi^*)[E_\varrho \,\mathcal{U}\, (\mathsf{PC}(\pi'))/\beth].$$

Proof. The claim $(\pi^*, \pi') \in \Pi(s,t)$ is obvious. Let $W = \mathcal{W}(\mathsf{PC}(\pi))$. Suppose $\Xi \vDash \mathsf{PC}(\pi)$ where $\Xi = (\xi_i)_{i \in \mathbb{N}} \in \mathcal{M}^{\mathbf{A}_{\mathsf{IMP}}}_{\mathbb{V},W}$. Then, there is a $k \in \mathbb{N}$, such that $(\Xi, k) \vDash \mathsf{E}(v_0) \wedge \mathsf{PC}((\varrho, \pi'))$ and there is a $k' \in \mathbb{N}$, $k' \ge k$, such that $(\Xi, k') \vDash \mathsf{PC}(\pi')$. For every node $v \in \mathrm{V}(\varrho)$ such that $\mathbf{def}(v) \neq \emptyset$ we have $\mathsf{E}(v) \rightarrowtail E_\varrho$ and for every data dependence edge $v \xrightarrow[x]{\mathsf{fd}}\!\triangleright\, w$ we also have $\mathsf{E}_\delta\big(v \xrightarrow[x]{\mathsf{fd}}\!\triangleright\, w\big) \rightarrowtail E_\varrho$. Since by construction of $\mathsf{PC}(\rho)$, for $j \in \{\, k, \ldots, k'-1 \,\}$, we have $\xi_j \vDash \mathsf{E}_\delta\big(v \xrightarrow[x]{\mathsf{fd}}\!\triangleright\, w\big)$ for some $v \xrightarrow[x]{\mathsf{fd}}\!\triangleright\, w \in \mathrm{E}(\rho)$ or $\xi_j \vDash \mathsf{E}(v)$ for some $v \in \mathrm{V}(\rho)$ with $\mathbf{def}(v) \neq \emptyset$, we have $(\Xi, j) \vDash E_\varrho$ for $j \in \{\, k, \ldots, k'-1 \,\}$. Hence, $(\Xi, k) \vDash E_\varrho \,\mathcal{U}\, \mathsf{PC}(\pi')$ and as $\mathcal{V}(E_\varrho)$ does not contain rigid variables, we also have $\Xi \vDash \mathsf{PC}(\pi^*)[E_\varrho \,\mathcal{U}\, \mathsf{PC}(\pi')/\beth]$.

Since $\aleph \rightarrowtail\!\!\!\Longrightarrow \beth \,\mathcal{U}\, \aleph$ (cf. table 3.2 (p. 17)), we obtain with lemmata 4 (p. 21) and 5 (p. 22):

$$\mathsf{PC}((\pi^*, \pi')) \Longleftrightarrow \mathsf{PC}(\pi^*)[\mathsf{PC}(\pi')/\beth] \rightarrowtail\!\!\!\Longrightarrow \mathsf{PC}(\pi^*)[E_\varrho \,\mathcal{U}\, (\mathsf{PC}(\pi'))/\beth].$$

□

Example 14
This example is to show that, in general, we must not omit the additional $\mathcal{U}$ operator. Consider the program p and its PDG in figure 6.4. We are interested in statement 2 (`a[0]:=1;`) influencing statement 9 (`p:=3;`).

We look at the path $\pi := 2 \xrightarrow[a_1-a_3]{\mathsf{fd}}\triangleright\ 5, (5,6), 6 \xrightarrow[i_5-i_3]{\mathsf{fd}}\triangleright\ 5, (5,8), (8,9)$ which contains the cycle $(5,6), 6 \xrightarrow[i_5-i_3]{\mathsf{fd}}\triangleright\ 5$. The path condition $\mathsf{PC}(\pi)$ for π is given by

$$\begin{aligned}
\mathsf{PC}(\pi) = {} & \underbrace{\mathsf{true}}_{=\mathsf{E}(2)} \wedge (y == 0) \wedge \Big(\underbrace{\mathsf{true}}_{=\mathsf{E}_\delta\left(2 \xrightarrow[a_1-a_3]{\mathsf{fd}}\triangleright 5\right)} \wedge\ {}^2\sqsupset^5_{a_1-a_3} \Big)\ \mathcal{U}\ \Big(\\
& (y == i_3) \wedge \underbrace{\mathsf{true}}_{=\delta_l\left(2 \xrightarrow[a_1-a_3]{\mathsf{fd}}\triangleright 5\right)} \wedge \underbrace{(a_1[y] == a_3[y])}_{={}^2\Phi^5_{a_1-a_3}} \wedge \underbrace{\mathsf{true}}_{=\delta^5_{a_3}} \wedge \underbrace{(i_2 < 2)}_{=\mathsf{E}(5)} \wedge \sqsupset_{(5,6)} \wedge \\
& \underbrace{((i_2 < 2) \wedge (a_3[i_3] == 1))}_{=\mathsf{E}(6)} \wedge \Big(\underbrace{(i_2 < 2)}_{=\mathsf{E}_\delta\left(6 \xrightarrow[i_5-i_3]{\mathsf{fd}}\triangleright 5\right)} \wedge\ {}^6\sqsupset^5_{i_5-i_3} \Big)\ \mathcal{U}\ \Big(\underbrace{\mathsf{true}}_{=\delta_l\left(6 \xrightarrow[i_5-i_3]{\mathsf{fd}}\triangleright 5\right)} \wedge \\
& \underbrace{(i_5 == i_3)}_{={}^6\Phi^5_{i_5-i_3}} \wedge \underbrace{\mathsf{true}}_{=\delta^5_{i_3}} \wedge \underbrace{(i_2 < 2)}_{=\mathsf{E}(5)} \wedge \sqsupset_{(5,8)} \wedge \underbrace{((i_2 < 2) \wedge !(a_3[i_3] == 1))}_{=\mathsf{E}(8)} \wedge \sqsupset_{(8,9)} \wedge \\
& \underbrace{((i_2 < 2) \wedge !(a[3i_3] == 1) \wedge (i_6 == 1))}_{=\mathsf{E}(9)} \wedge \sqsupset)\Big)
\end{aligned}$$

where y is rigid. $\overline{\mathsf{PC}(\pi)}$ simplifies to

$$\begin{aligned}
\overline{\mathsf{PC}(\pi)} \Longleftrightarrow {} & (y == 0) \wedge \Diamond((y == i_3) \wedge (a_1[y] == a_3[y]) \wedge (a_3[i_3] == 1) \wedge \\
& (i_2 < 2)\ \mathcal{U}\ ((i_2 < 2) \wedge (i_5 == i_3) \wedge !(a_3[i_3] == 1) \wedge (i_6 == 1)))
\end{aligned} \tag{6.18}$$

If we look at the path $\varrho := 2 \xrightarrow[a_1-a_3]{\mathsf{fd}}\triangleright\ 5, (5,8), (8,9)$ which is π without the cycle $(5,6), 6 \xrightarrow[i_5-i_3]{\mathsf{fd}}\triangleright\ 5$, we obtain the following path condition for ϱ:

$$\begin{aligned}
\mathsf{PC}(\varrho) = {} & \mathrm{true} \wedge (y == 0) \wedge (\mathrm{true} \wedge\ {}^2\sqsupset^5_{a_1-a_3})\ \mathcal{U}\ ((y == i_3) \wedge \mathrm{true} \wedge \\
& (a_1[y] == a_3[y]) \wedge \mathrm{true} \wedge (i_2 < 2) \wedge \sqsupset_{(5,8)} \wedge ((i_2 < 2) \wedge !(a_2[i_3] == 1)) \wedge \\
& \sqsupset_{(8,9)} \wedge ((i_2 < 2) \wedge !(a_3[i_3] == 1) \wedge (i_6 == 1)) \wedge \sqsupset)
\end{aligned}$$

Simplifying $\overline{\mathsf{PC}(\varrho)}$ yields

$$\begin{aligned}
\overline{\mathsf{PC}(\varrho)} \Longleftrightarrow {} & (y == 0) \wedge \Diamond((y == i_3) \wedge (a_1[y] == a_3[y]) \wedge (i_2 < 2) \wedge \\
& !(a_3[i_3] == 1) \wedge (i_6 == 1))
\end{aligned} \tag{6.19}$$

The important point is the missing $\mathcal{U}$ operator in equation 6.19. Let Ξ be the state sequence that corresponds to the path $1, 2, 3, 4, 5, 6, 4, 5, 8, 9$ in the CFG (or in terms of the transition graph: to the path $1, 2, 3, 6, 9$.) Then $\Xi \vDash \overline{\mathsf{PC}(\pi)}$, but not $\Xi \vDash \overline{\mathsf{PC}(\varrho)}$.

6.3.2.3 Cycles with array variables

So far, we have presented how to avoid having to generate path conditions for paths that contain control or scalar flow dependence-only cycles. For cycles with

data dependence edges with respect to array variables, things are more difficult. If we look again at the factorial program in figure 6.3 (p. 55) and the path condition $\mathsf{PC}(\rho)$ from equation 6.11 (p. 57) we find that the path ρ contains the cycle $4 \xrightarrow[a_3-a_2]{\mathsf{fd}\triangleright} 4$ which is a flow dependence edge with array variable $\mathtt{a}$. The problem there is that in each iteration different array cells are affected. As every array has only a finite number of cells we could solve this problem by distinguishing all array cells and then treat them like scalar variables and thus handle these cycles similarly, but this approach is not feasible. To address this issue, we distinguish three cases for paths with cycles containing array dependences:

Let $\pi \in \Pi(s,t)$ be an information flow path in PDG_p from s to t that contains neither control nor scalar flow dependence cycles. Let $\varrho := e_1, \ldots, e_n$ be a cycle[19] in π which contains a dependence edge for an array variable a. As usual, let π^* denote the subpath of π before ϱ and π' the subpath of π after ϱ. Then, one of the following cases applies:

1. ϱ is of the form $v_0 \xrightarrow[a]{\mathsf{dd}\triangleright} v_1, \ldots, v_{n-1} \xrightarrow[a]{\mathsf{dd}\triangleright} v_n$, i. e. all edges of ϱ are def-def dependences with respect to a.

2. ϱ is of the form

$$e_1, \ldots, e_k, v_k \xrightarrow[a]{\mathsf{dd}\triangleright} v_{k+1}, \ldots, v_{m-2} \xrightarrow[a]{\mathsf{dd}\triangleright} v_{m-1}, v_{m-1} \xrightarrow[a]{\mathsf{fd}\triangleright} v_m, e_{m+1}, \ldots, e_n$$

   where $1 \leq k < m \leq n$ and e_k is not of the form $v_{k-1} \xrightarrow[a]{\mathsf{dd}\triangleright} v_k$, i. e. the array dependence cycle is completely contained in ϱ.

3. ϱ is of the form

   (a) $v_0 \xrightarrow[a]{\mathsf{dd}\triangleright} v_1, \ldots, v_{m-2} \xrightarrow[a]{\mathsf{dd}\triangleright} v_{m-1}, v_{m-1} \xrightarrow[a]{\mathsf{fd}\triangleright} v_m, e_{m+1}, \ldots, e_n$ or

   (b) $e_1, \ldots, e_k, v_k \xrightarrow[a]{\mathsf{dd}\triangleright} v_{k+1} \ldots, v_{m-1} \xrightarrow[a]{\mathsf{dd}\triangleright} v_m$

   where $0 < m \leq n$, $k > 0$, and e_k is not of the form $v_{k-1} \xrightarrow[a]{\mathsf{dd}\triangleright} v_k$, i. e. ϱ starts or continues with an array dependence edge sequence which ends inside ϱ, or ϱ ends with a def-def dependence edge sequence that starts inside ϱ.

In the first case, we can ignore the cycle and add one more until operator like we did this for scalar flow dependences in lemma 9 (p. 60):

Lemma 10 (def-def dependence cycles)
Let the identifiers be as in the first case. Then (π^*, π') is an information flow path from s to t. Let E_ϱ be as in lemma 9 (p. 60) and let $v_n \xrightarrow[a]{dd\triangleright} w$ or $v_n \xrightarrow[a]{fd\triangleright} w$ be the first edge in π'. Let ${}^{v_n}\sqsupset_a^w$ be the propositional variable associated with this edge. Let $(\eta, \eta') := \mathsf{PC}((\pi^*, \pi'))\,\langle\!\langle \mathcal{U}, {}^{v_n}\sqsupset_a^w \rangle\!\rangle$. Let y denote the rigid variable used for the dependence $v_n \xrightarrow[a]{fd\triangleright} w$ or $v_n \xrightarrow[a]{dd\triangleright} w$ and i_0 the index expression for array a in v_0.

If π^* is empty or its last edge is not of the form $v \xrightarrow[a]{dd\triangleright} v_0$, set

$$\kappa := E_\varrho \,\mathcal{U}\, (\mathsf{E}(v_0) \wedge \eta \,\mathcal{U}\, \eta').$$

[19] We assume that ϱ is a single cycle, i. e. except for the first and last node in ϱ, all nodes in ϱ are pairwise different.

Otherwise, set

$$\kappa := E_\varrho\, \mathcal{U}\ ((y! = i_0) \wedge {}^{v}\Phi_a^{v_0} \wedge \mathsf{E}(v_0) \wedge \eta\, \mathcal{U}\ \eta').$$

Set $\theta := \mathsf{PC}((\pi^*, \pi'))\ \langle\!\langle \mathcal{U},\ {}^{v_n}\sqsupseteq_a^{w} \mid \kappa \rangle\!\rangle$. *Then,* $\mathsf{PC}((\pi^*, \pi')) \rightarrowtail\!\!\!\Longrightarrow \theta$ *and* $\mathsf{PC}(\pi) \rightarrowtail\!\!\!\Longrightarrow \theta$.

Proof. Although the notation is quite different from lemma 9 (p. 60), basically the same thing happens, an additional $\mathcal{U}$ operator is introduced.

If π^* is empty or its last edge is not of the form $v \xrightarrow[a]{\mathsf{dd}\triangleright} v_0$, then ϱ is a def-def dependence cycle at the beginning of a sequence of array dependence edges, so the start and end node of the cycle ϱ coincides with the node where the rigid variable y is bound to the index value. To ensure that θ is a correct path condition for (π^*, π'), we must not include the constraint that the array cell i_0 is not overwritten at the end of the cycle.

Otherwise, we know that both before and after the cycle, we can safely include the constraint that the array cell is not overwritten as even the path without cycles contains this constraint.

The main difference to lemma 9 (p. 60) is that this notation allows us to use the same rigid variable in π^* and π' which would not be possible if we wrote something like $\mathsf{PC}(\pi^*)[E_\varrho\ \mathcal{U}\ \mathsf{PC}(\pi')/\sqsupseteq]$. Nevertheless, the proof goes along the same lines as the one for lemma 9 (p. 60). □

Hence, in the following, we may assume without loss of generality that π does not contain pure def-def cycles. For the second case, things are very similar: We can eliminate this cycle as we did it with scalar dependence cycles.

Lemma 11 (Complete array dependence cycles)
Let the identifiers be as in the second case and in lemma 9 (p. 60). Then $(\pi^*, \pi') \in \Pi(s, t)$ *is an information flow path such that*

$$\mathsf{PC}((\pi^*, \pi')) \rightarrowtail\!\!\!\Longrightarrow \mathsf{PC}(\pi^*)[E_\varrho\, \mathcal{U}\ \mathsf{PC}(\pi')/\sqsupseteq] \text{ and } \mathsf{PC}(\pi) \rightarrowtail\!\!\!\Longrightarrow \mathsf{PC}(\pi^*)[E_\varrho\, \mathcal{U}\ \mathsf{PC}(\pi')/\sqsupseteq].$$

In the last case, we can not be sure whether (π^*, π') is an information flow path at all, for example, if the last edge in π^* is of the form $v \xrightarrow[y]{\mathsf{dd}\triangleright} v_0$ and $m < n$. Even if (π^*, π') is an information flow path, we can not always apply the same trick as before because this cycle might describe a completely different type of influence, in case 3b, for instance.

- If ϱ is of type 3a and π^* is an empty path or the last edge of π^* is not of the form $v \xrightarrow[a]{\mathsf{dd}\triangleright} v_0$, then the subformula for $\mathsf{PC}(\varrho)$ in $\mathsf{PC}(\pi)$ is not related (via rigid variables) to the rest of $\mathsf{PC}(\pi)$. Hence we obtain the same entailments as in lemma 11.
- If, however, the last edge of π^* is of the form $v \xrightarrow[a]{\mathsf{dd}\triangleright} v_0$ or ϱ is of type 3b, we are not able to eliminate this cycle.[20]

  In this case, we say that ϱ is an **open** array dependence cycle in π.

[20] If we had used a non-killing definition for array flow dependences, this type of cycles would not have come up because the whole sequence of def-def dependences followed by one flow dependence would have been subsumed in a single flow dependence edge.

Definition 43 (Influence condition for sets of paths)
Let $\Pi^*(s,t) \subseteq \Pi(s,t)$ denote the set of all information flow paths $\pi : s \xrightarrow[\mathrm{PDG}_o]{}\!\!\!\!\blacktriangleright^* t$ which contain

- neither control dependence cycles,
- nor scalar flow dependence cylces,
- nor def-def dependence cycles,
- nor cycles that are not open array dependence cycles.

We call the paths in $\Pi^*(s,t)$ **influence paths** from s to t. For every path $\pi \in \Pi^*(s,t)$, we consider the set of paths

$$\Pi(\pi) := \left\{ \pi' \in \Pi(s,t) \;\middle|\; \pi \text{ contained in } \pi' \right\}.$$

Let $f \in \mathrm{E}(\pi) \cup \{\epsilon\}$ and let $e_\pi \in \mathrm{E}(\pi)$ be the first edge in π (after f if $f \neq \epsilon$) such that there exists a cycle $\varrho : \odot\!\!\succ(e_\pi) \xrightarrow[\mathrm{PDG}_p]{}\!\!\!\!\blacktriangleright^* \odot\!\!\succ(e_\pi)$ such that, when we insert ϱ before e_π into π, we obtain a path $\pi' \in \Pi(\pi)$ and ϱ is not an open array dependence cycle in π'. Let $P^f_{e_\pi}$ denote the set of all such cycles ϱ and set

$$\mathsf{E}\left(P^f_{e_\pi}\right) := \bigvee_{\varrho \in P^f_{e_\pi}} \left(\bigvee_{\substack{v \in \mathrm{V}(\varrho) \\ \mathbf{def}(v) \neq \emptyset}} \mathsf{E}(v) \vee \bigvee_{e \in \mathrm{E}(\varrho) \cap D} \mathsf{E}_\delta(e) \right).^{21}$$

Let π^* denote the subpath of π before e_π and π' the subpath of π from e_π on.

- If no such e_π exists, we set

$$\mathsf{IC}^f(\pi) := \mathsf{PC}(\pi).$$

- If all paths in $P^f_{e_\pi}$ are def-def dependence-only cycles of the form

$$v_0 \xrightarrow[a]{\mathsf{dd}}\!\!\triangleright\, v_1, \ldots, v_{n-1} \xrightarrow[a]{\mathsf{dd}}\!\!\triangleright\, v_n,$$

  let ${}^{v_n}\!\sqsupset^w_a$ and κ be as in lemma 10 (p. 62) with $E_\varrho := \mathsf{E}\left(P^f_{e_\pi}\right)$. We set

$$\mathsf{IC}^e(\pi) := \mathsf{IC}^{e_\pi}(\pi)\, \langle\!\langle \mathcal{U},\, {}^{v_n}\!\sqsupset^w_a \mid \kappa \rangle\!\rangle$$

- If none of the paths in $P^f_{e_\pi}$ is a def-def dependence only cycle, we set:

$$\mathsf{IC}^f(\pi) := \mathsf{PC}(\pi^*)[\mathsf{E}\left(P^f_{e_\pi}\right) \,\mathcal{U}\, \mathsf{IC}(\pi')/\sqsupset]$$

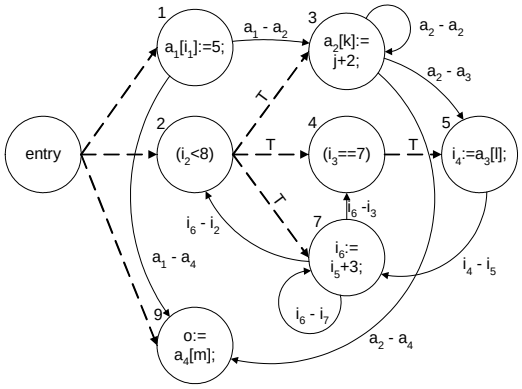
If $f = \epsilon$ is the empty word, we omit writing f.

Proposition 12
Let $\pi \in \Pi^*(s,t)$. Then, $\mathsf{PC}(\pi') \Longrightarrow \mathsf{IC}(\pi)$ for every $\pi' \in \Pi(\pi)$.

[21] Although $P^f_{e_\pi}$ is usually infinite, $\mathsf{E}\left(P^f_{e_\pi}\right)$ can always be constructed in such a way that the disjunction remains finite. In what follows, we will always write simplified versions of $\mathsf{E}\left(P^f_{e_\pi}\right)$.

Figure 6.5 An example program with cycles in its PDG.

```
1  a[i] := 5;
2  while (i < 8) {
3      a[k] := j + 2;
4      if (i == 7) {
5          i := a[l];
6      }
7      i := i + 3;
8  }
9  o := a[m];
```



Proof. Let $\pi \in \Pi^*(s,t)$. The claim follows by induction on the positions in π at which we can add a cycle such that we obtain a path in $\Pi(\pi)$ where we use the entailments from lemmata 8 (p. 59), 9 (p. 60), 10 (p. 62), and 11 (p. 63) together with the distribution laws from table 3.1 (p. 16) and the implication $\aleph \,\mathcal{U}\, \beth \vee \neg \,\mathcal{U}\, \beth \rightarrowtail (\aleph \vee \neg) \,\mathcal{U}\, \beth$ from table 3.2 (p. 17). □

Corollary 13

$$\bigvee_{\pi \in \Pi(s,t)} PC(\pi) \Longrightarrow \bigvee_{\pi \in \Pi^*(s,t)} IC(\pi)$$

From now on, we want the influence condition $\mathsf{IC}(s,t)$ to denote $\bigvee_{\pi \in \Pi^*(s,t)} \mathsf{IC}(\pi)$.

Lemma 14
Let $\pi \in \Pi^*(s,t)$. If there exist two influence paths π^*, π' such that $\pi = (\pi^*, \pi')$, then

$$IC(\pi) \Longleftrightarrow IC(\pi^*)[IC(\pi')/\beth]$$

Proof. Suppose $\pi = (\pi^*, \pi') \in \Pi^*(s,t)$. Let u be the target node of the last edge of π^* and source node of the first edge in π'. Then $\pi^* \in \Pi^*(s,u)$ and $\pi' \in \Pi^*(u,t)$. The only difference between $\mathsf{IC}(\pi)$ and $\mathsf{IC}(\pi^*)[\mathsf{IC}(\pi')/\beth]$ is that the execution condition for node u appears twice in a conjunction.[22] Thus, the claim follows from the associativity and idempotence laws for conjunctions from table 3.1 (p. 16). □

Example 15 (An example with cycles)
This example shows how the additional until operators are incorporated into path conditions to capture all information flow paths. Figure 6.5 shows the program and its PDG. We want to generate an influence condition for line 1 influencing line 9.

[22] If we can insert a nonempty cycle $\rho : u \xrightarrow{\mathrm{PDG}_p}{}^{*} u$ into π between π^* and π' such that $(\pi^*, \rho, \pi') \in \Pi(s,t)$, we assume that we take care of these cycles in either $\mathsf{IC}(\pi^*)$ or $\mathsf{IC}(\pi')$.

The restricted path set $\Pi^*(1,9)$ contains three paths:

- $\pi_1 := 1 \xrightarrow[a_1-a_4]{\mathsf{fd}} 9$
- $\pi_2 := 1 \xrightarrow[a_1-a_2]{\mathsf{dd}} 3,3 \xrightarrow[a_2-a_4]{\mathsf{fd}} 9$
- $\pi_3 := 1 \xrightarrow[a_1-a_2]{\mathsf{dd}} 3,3 \xrightarrow[a_2-a_3]{\mathsf{fd}} 5,5 \xrightarrow[i_4-i_5]{\mathsf{fd}} 7,7 \xrightarrow[i_6-i_2]{\mathsf{fd}} 2,(2,3),3 \xrightarrow[a_2-a_4]{\mathsf{fd}} 9$

Here, π_3 contains the open array dependence cycle $3 \xrightarrow[a_2-a_3]{\mathsf{fd}} 5,5 \xrightarrow[i_4-i_5]{\mathsf{fd}} 7,7 \xrightarrow[i_6-i_2]{\mathsf{fd}} 2,(2,3)$.

π_1: Since there are no cycles in the PDG starting at node 1 or 9, the influence condition for π_1 is simply the ordinary path condition augmented with two propositional variables:

$$\begin{aligned}\mathsf{IC}(\pi_1) = \text{true} \wedge (y == i_1) \wedge (\text{true} \wedge {}^1\beth^9_{a_1-a_4})\, \mathcal{U}\, ((y == m) \wedge \text{true} \wedge \\ (a_1[y] == a_4[y]) \wedge \text{true} \wedge !(i_2 < 8) \wedge \beth)\end{aligned}$$

where y is a rigid variable.

π_2: When we look at π_2, we note that all paths in $\Pi(\pi_2)$ are of the form $1 \xrightarrow[a_1-a_2]{\mathsf{dd}} 3,3 \xrightarrow[a_2-a_2]{\mathsf{dd}} 3,\ldots,3 \xrightarrow[a_2-a_2]{\mathsf{dd}} 3,3 \xrightarrow[a_2-a_4]{\mathsf{fd}} 9$. Hence, all cycles we have to consider are def-def dependence cycles which are not the beginning of an array dependence sequence because $1 \xrightarrow[a_1-a_2]{\mathsf{dd}} 3$ is already a def-def dependence with respect to a. The common execution condition $\mathsf{E}\big(P_{e_{\pi_2}}\big)$ for the cycle is $(i_2 < 8)$. Thus, we have

$$\begin{aligned}\mathsf{IC}(\pi_2) = \text{true} \wedge (y == i_1) \wedge (\text{true} \wedge {}^1\beth^3_{a_1-a_2})\, \mathcal{U}\, ((y! = k) \wedge \text{true} \wedge \\ (a_1[y] == a_2[y]) \wedge (i_2 < 8) \wedge \\ (i_2 < 8)\, \mathcal{U}\, ((y! = k) \wedge (a_1[y] == a_2[y]) \wedge (i_2 < 8) \wedge \\ (\text{true} \wedge {}^3\beth^9_{a_2-a_4})\, \mathcal{U}\, ((y == m) \wedge !(i_2 < 8) \wedge (a_2[y] == a_4[y]) \wedge \\ \text{true} \wedge !(i_2 < 8) \wedge \beth)))\end{aligned} \tag{6.20}$$

where y is a rigid variable. The third line is what we have inserted to account for the def-def dependence cycles.

π_3: The influence condition for path π_3 must cover cycles at nodes 3, 5, 7, and 2. The cycles at node 3 are the same as those for π_3, hence

$$\mathsf{E}\big(P_{e_{\pi_3}}\big) = \mathsf{E}\Big(P^{(2,3)}_{e_{\pi_3}}\Big) = (i_2 < 8).$$

Note that the first time we reach node 3 we are already inside an array dependence sequence whereas the second time we start it. Hence, both cases of lemma 10 (p. 62) can be seen here. For the cycles at nodes 5, 7, and 2, we obtain

$$\mathsf{E}\left(P^{3 \xrightarrow[a_2-a_3]{\mathsf{fd}} 5}_{e_{\pi_3}}\right) = \mathsf{E}\left(P^{5 \xrightarrow[i_4-i_5]{\mathsf{fd}} 7}_{e_{\pi_3}}\right) = \mathsf{E}\left(P^{7 \xrightarrow[i_6-i_2]{\mathsf{fd}} 2}_{e_{\pi_3}}\right) = \text{true}$$

because $\mathsf{E}(2) = \text{true}$ and cycles with edge $7 \xrightarrow[i_6 - i_2]{\text{fd}} \triangleright\, 2$ exist for all three start nodes. If we plug everything together, we get:

$$
\begin{aligned}
\mathsf{IC}(\pi_3) = {} & \underbrace{\text{true}}_{=\mathsf{E}(1)} \wedge (y_1 == i_1) \wedge \big(\underbrace{\text{true}}_{=\mathsf{E}_\delta\left(1 \xrightarrow[a_1 - a_2]{\text{dd}} \triangleright 3\right)} \wedge {}^{1}\sqsupset^{3}_{a_1 - a_2} \big)\, \mathcal{U}\, \big((y_1! = k) \wedge \\
& \underbrace{\text{true}}_{=\delta_l\left(1 \xrightarrow[a_1 - a_2]{\text{dd}} \triangleright 3\right)} \wedge \underbrace{(a_1[y_1] == a_2[y_1])}_{={}^{1}\Phi^{3}_{a_1 - a_2}} \wedge \underbrace{(i_2 < 8)}_{=\mathsf{E}(3)} \wedge \\
& \underbrace{(\boldsymbol{i_2 < 8})}_{=\mathsf{E}\left(\boldsymbol{P_{e_{\pi_3}}}\right)}\, \boldsymbol{\mathcal{U}}\, \big((\boldsymbol{y_1! = k}) \wedge \underbrace{(\boldsymbol{a_1[y_1] == a_2[y_1]})}_{=\boldsymbol{{}^{1}\Phi^{3}_{a_1 - a_2}}} \wedge \underbrace{(\boldsymbol{i_2 < 8})}_{=\mathsf{E}(\boldsymbol{3})} \wedge \\
& \big(\underbrace{(i_2 < 8)}_{=\mathsf{E}_\delta\left(3 \xrightarrow[a_2 - a_3]{\text{fd}} \triangleright 5\right)} \wedge {}^{3}\sqsupset^{5}_{a_2 - a_3} \big)\, \mathcal{U}\, \big((y_1 == l) \wedge \underbrace{\text{true}}_{=\delta_l\left(3 \xrightarrow[a_2 - a_3]{\text{fd}} \triangleright 5\right)} \wedge \\
& \underbrace{(a_2[y_1] == a_3[y_1])}_{={}^{3}\Phi^{5}_{a_2 - a_3}} \wedge \underbrace{\text{true}}_{=\delta^{5}_{a_3}} \wedge \underbrace{((i_2 < 8) \wedge (i_3 == 7))}_{=\mathsf{E}(5)} \wedge \\
& \underbrace{\boldsymbol{\text{true}}}_{=\mathsf{E}\left(\boldsymbol{P^{3 \xrightarrow[a_2 - a_3]{\text{fd}} \triangleright 5}_{e_{\pi_3}}}\right)}\, \boldsymbol{\mathcal{U}}\, \big(\underbrace{\boldsymbol{((i_2 < 8) \wedge (i_3 == 7))}}_{=\mathsf{E}(\boldsymbol{5})} \wedge \\
& \big(\underbrace{(i_2 < 8)}_{=\mathsf{E}_\delta\left(5 \xrightarrow[i_4 - i_5]{\text{fd}} \triangleright 7\right)} \wedge {}^{5}\sqsupset^{7}_{i_4 - i_5} \big)\, \mathcal{U}\, \big(\underbrace{\text{true}}_{=\delta_l\left(5 \xrightarrow[i_4 - i_5]{\text{fd}} \triangleright 7\right)} \wedge \underbrace{(i_4 == i_5)}_{={}^{5}\Phi^{7}_{i_4 - i_5}} \wedge \underbrace{\text{true}}_{=\delta^{7}_{i_5}} \wedge \\
& \underbrace{(i_2 < 8)}_{=\mathsf{E}(7)} \wedge \underbrace{\boldsymbol{\text{true}}}_{=\mathsf{E}\left(\boldsymbol{P^{5 \xrightarrow[i_4 - i_5]{\text{fd}} \triangleright 7}_{e_{\pi_3}}}\right)}\, \boldsymbol{\mathcal{U}}\, \big(\underbrace{\boldsymbol{(i_2 < 8)}}_{=\mathsf{E}(\boldsymbol{7})} \wedge \\
& \big(\underbrace{\text{true}}_{=\mathsf{E}_\delta\left(7 \xrightarrow[i_6 - i_2]{\text{fd}} \triangleright 2\right)} \wedge {}^{7}\sqsupset^{2}_{i_6 - i_2} \big)\, \mathcal{U}\, \big(\underbrace{\text{true}}_{=\delta_l\left(7 \xrightarrow[i_6 - i_2]{\text{fd}} \triangleright 2\right)} \wedge \underbrace{(i_6 == i_2)}_{={}^{7}\Phi^{2}_{i_6 - i_2}} \wedge \underbrace{\text{true}}_{=\delta^{2}_{i_2}} \wedge \\
& \underbrace{\text{true}}_{=\mathsf{E}(2)} \wedge \underbrace{\boldsymbol{\text{true}}}_{=\mathsf{E}\left(\boldsymbol{P^{7 \xrightarrow[i_6 - i_2]{\text{fd}} \triangleright 2}_{e_{\pi_3}}}\right)}\, \boldsymbol{\mathcal{U}}\, \big(\underbrace{\boldsymbol{\text{true}}}_{=\mathsf{E}(\boldsymbol{2})} \wedge \sqsupset_{(2,3)} \wedge \underbrace{(i_2 < 8)}_{=\mathsf{E}(3)} \wedge (y_2 == k) \wedge \\
& \underbrace{\boldsymbol{(i_2 < 8)}}_{=\mathsf{E}\left(\boldsymbol{P^{(2,3)}_{e_{\pi_3}}}\right)}\, \boldsymbol{\mathcal{U}}\, \big(\underbrace{\boldsymbol{(i_2 < 8)}}_{=\mathsf{E}(\boldsymbol{3})} \wedge \\
& \big(\underbrace{\text{true}}_{=\mathsf{E}_\delta\left(3 \xrightarrow[a_2 - a_4]{\text{fd}} \triangleright 9\right)} \wedge {}^{3}\sqsupset^{9}_{a_2 - a_4} \big)\, \mathcal{U}\, \big((y_2 == m) \wedge \\
& \underbrace{!(i_2 < 8)}_{=\delta_l\left(3 \xrightarrow[a_2 - a_4]{\text{fd}} \triangleright 9\right)} \wedge \underbrace{(a_2[y_2] == a_4[y_2])}_{={}^{3}\Phi^{9}_{a_2 - a_4}} \wedge \underbrace{\text{true}}_{=\delta^{9}_{a_4}} \wedge \underbrace{!(i_2 < 8)}_{=\mathsf{E}(9)} \wedge \sqsupset) \big) \big) \big) \big) \big) \big) \big) \big) \big) \big) \big)
\end{aligned}
$$

where y_1 and y_2 are rigid variables. What we have added to account for the cycles is set in bold face.

Obviously, these formulae can be simplified and need to be. We obtain:

$$\begin{aligned}\overline{\mathsf{IC}(1,9)} \Longleftrightarrow &(y == i_1) \wedge \Diamond((y == m) \wedge (a_1[y] == a_4[y]) \wedge !(i_2 < 8) \vee (y! = k) \wedge \\ &(a_1[y] == a_2[y]) \wedge (i_2 < 8) \wedge \Diamond((y == m) \wedge !(i_2 < 8) \wedge (a_2 == a_4[y]))) \vee \\ &(y_1 == i_1) \wedge \Diamond((y_1! = k) \wedge (a_1[y_1] == a_2[y_1]) \wedge \\ &(i_2 < 8)\, \mathcal{U}\, ((y_1 == l) \wedge (a_2[y_1] == a_3[y_1]) \wedge (i_2 < 8) \wedge (i_3 == 7) \wedge \\ &\Diamond((i_3 == 7) \wedge (i_2 < 8)\, \mathcal{U}\, ((i_4 == i_5) \wedge (i_2 < 8) \wedge \Diamond((i_6 == i_2) \wedge \\ &\Diamond((i_2 < 8) \wedge (y_2 == k) \wedge \Diamond((y_2 == m) \wedge !(i_2 < 8) \wedge \\ &(a_2[y_2] == a_4[y_2])))))))))\end{aligned}$$

6.4 Simplifying influence conditions

So far we have presented rules for generating both path and influence conditions. However they tend to become very long, even for small programs. Even though there is a number of general congruences and entailments listed in tables 3.1 (p. 16) and 3.2 (p. 17) which can be used to simplify both path and influence conditions, they do not help a lot in making the formulae shorter. What is more we are mostly interested in whether it is over $\mathcal{M}_p$ that such a formula is satisfiable. Hence, we can try to exploit the structure of the program p and, thus, of the sequences in $\mathcal{M}_p$ to further simplify the condition.

In this section, we present three different aspects of simplification:

- Eliminating rigid variables,
- Using fewer temporal operators, and
- Having to generate influence conditions for fewer paths.

Of course, the ideas presented here are not exhaustive in the sense that they cover all simplification options. In particular, we have not included simplification with congruences and entailments, although some useful patterns can be found in tables 3.1 (p. 16) and 3.2 (p. 17)

6.4.1 Eliminating rigid variables

First, we look at rigid variables. These are introduced whenever there is a data dependence edge sequence of the form $v_0 \xrightarrow[a]{\mathsf{dd}} v_1, \ldots, v_{n-2} \xrightarrow[a]{\mathsf{dd}} v_{n-1}, v_{n-1} \xrightarrow[a]{\mathsf{fd}} v_n$ with respect to some array variable a. We present some conditions under which we can remove the rigid variable in the influence condition.

6.4.1.1 Constant index expressions

If the index expression for the array variable a in either node v_0 or node v_n is constant, i. e. does not contain any program variables, then we can substitute the constant expression for the rigid variable in the influence condition:

Lemma 15
Let $\pi \in \Pi^*(s,t)$ be an influence path which contains a maximal subpath $\rho = v_0 \xrightarrow[a]{dd} v_1, \ldots, v_{n-2} \xrightarrow[a]{dd} v_{n-1}, v_{n-1} \xrightarrow[a]{fd} v_n$ where the index expression i_0 for the array access a in v_0 does not contain any variables, i. e. $\mathcal{V}(i_0) = \emptyset$. Let y denote the rigid variable used when we have generated the condition for ρ. Then

$$IC(\pi) \rightarrowtail IC(\pi)[i_0/y].^{23}$$

Proof. Let $W := \mathcal{W}(\mathsf{IC}(\pi))$ and let $\Xi = (\xi_i)_{i\in\mathbb{N}} \in \mathcal{M}^{\mathbf{A}_{\mathsf{IMP}}}_{\mathbb{V},W}$. Suppose $\Xi \vDash \mathsf{IC}(\pi)$. Then there is a $j \in \mathbb{N}$ such that $\xi_j \vDash (y == i_0)$. Since $\mathcal{V}(i_0) = \emptyset$, $i \mapsto \xi_i(i_0), i \in \mathbb{N}$ is a constant function, say $c := \xi_0(i_0)$. Hence, $\xi_i(y) = c$ for all $i \in \mathbb{N}$, because y is rigid, and $c \notin \bot$. Therefore, $\Xi \vDash \mathsf{IC}(\pi)[c/y]$, and consequently $\Xi \vDash \mathsf{IC}(\pi)[i_0/y]$. □

Lemma 16
Let $\pi \in \Pi^*(s,t)$ be an influence path which contains a maximal subpath $\rho = v_0 \xrightarrow[a]{dd} v_1, \ldots, v_{n-2} \xrightarrow[a]{dd} v_{n-1}, v_{n-1} \xrightarrow[a]{fd} v_n$ where the index expression i_n for the array access a in v_n does not contain any variables, i. e. $\mathcal{V}(i_n) = \emptyset$. Let y denote the rigid variable used when we have generated the condition for ρ. Then

$$IC(\pi) \rightarrowtail IC(\pi)[i_n/y].$$

Proof. If we substitute i_n for i_0 in the proof of lemma 15, we obtain a proof for this lemma. □

Example 16
Reconsider the factorial program in figure 6.3 (p. 55). In particular, we look at the path $\pi := 2 \xrightarrow[a_1-a_2]{\mathsf{fd}} 4, 4 \xrightarrow[a_3-a_4]{\mathsf{fd}} 7 \in \Pi^*(2,7)$. First, we compute the standard influence condition $\overline{\mathsf{IC}(\pi)}$. The only position to add a cycle to π is at node 4. The execution condition for this cycle is true as $\mathsf{E}(3) = \text{true}$. Hence, we obtain:

$$\begin{aligned}\overline{\mathsf{IC}(\pi)} \Longleftrightarrow &(y_1 == 1) \wedge \Diamond((y_1 == i_3) \wedge (i_4 + 1! = 0) \wedge (a_1[y_1] == a_2[y_1]) \wedge \\ &(i_2 < 6) \wedge (y_2 == i_5 + 1) \wedge \Diamond((i_2 < 6) \wedge \Diamond((y_2 == 5) \wedge \\ &(a_3[y_2] == a_4[y_2]) \wedge !(i_2 < 6))))\end{aligned}$$

where y_1 and y_2 are rigid variables.

We can apply lemma 15 for y_1 and lemma 16 for y_2. Then, we obtain:

$$\begin{aligned}\overline{\mathsf{IC}(\pi)} \rightarrowtail \Diamond((1 == i_3) \wedge (i_4 + 1! = 0) \wedge (a_1[1] == a_2[1]) \wedge (i_2 < 6) \wedge \\ (5 == i_5 + 1) \wedge \Diamond((i_2 < 6) \wedge \Diamond((a_3[5] == a_4[5]) \wedge !(i_2 < 6))))\end{aligned}$$

6.4.1.2 Array dependences of no return

We have just seen that we can eliminate a rigid variable if the first or last index expression is constant. Next we see that we can do the same - even if it does contain variables - when control flow cannot return to the first node in the sequence of array dependence edges.

[23] Note that we can not expect congruence here because $\mathsf{IC}(\pi)[i_0/y]$ does not impose any constraints on the rigid variable y whereas $\mathsf{IC}(\pi)$ does.

Lemma 17
Let $\pi \in \Pi^*(s,t)$ be an influence path from s to t which contains a maximal subpath ρ of the form the form $v_0 \xrightarrow[a]{dd} v_1, \ldots, v_{n-2} \xrightarrow[a]{dd} v_{n-1}, v_{n-1} \xrightarrow[a]{fd} v_n$ for some array variable a such that $v_0 \neq v_j$ for $0 < j \leq n$. Let i_0 be the index expression for array a in v_0 and let y be the rigid variable used when we generated the condition for ρ. Let Π denote the set of all information flow paths from v_0 to v_n that start with the same edge as ρ does in π:

$$\Pi := \Big\{ \pi' : v_0 \xrightarrow[\mathrm{PDG}_p]{}{}^{*} v_n \in \Pi(\rho) \Big| (\pi' = v_0 \xrightarrow[a]{dd} v_1, \pi'' \vee \pi' = v_0 \xrightarrow[a]{fd} v_1) \wedge \pi \text{ contained in } \pi' \wedge \pi' \text{ contains only data dependence edges with respect to } a \Big\}$$

If $v_0 \notin \rightarrowtail\!\!\!\!\circledcirc(\bigcup \mathrm{E}(\Pi))$, then

$$IC(\pi) \overset{\mathcal{M}_p}{\Longrightarrow} IC(\pi)[i_0/y].$$

Proof. Let $W := \mathcal{W}(\mathsf{IC}(\pi))$ and let $\Xi = (\xi_i)_{i\in\mathbb{N}} \in \mathcal{M}_p^W$. Suppose $(\Xi, l) \vDash \mathsf{IC}(\pi)$. Let π^* denote the subpath of π before ρ and π' the subpath after it. Then there is a $j \in \mathbb{N}$, $j \geq l$ such that $\xi_j \vDash (y == i_0)$ and there is a $k \in \mathbb{N}$, $k \geq j$ such that $\xi_k \vDash \mathsf{IC}(\pi')$. Since $v_0 \notin \rightarrowtail\!\!\!\!\circledcirc(\bigcup \mathrm{E}(\Pi))$ there is no $i \in \mathbb{N}$ with $j < i \leq k$ such that ξ_i corresponds to node v_0. Therefore, for all $i \in \{ j, \ldots, k \}$, $\xi_i(i_0) = \xi_i(y) = \xi_j(i_0)$. Neither does y occur in $\mathsf{IC}(\pi^*)$, nor in $\mathsf{IC}(\pi')$, so we have

$$(\Xi, l) \vDash \mathsf{IC}(\pi')[(\mathsf{IC}(\rho)[i_0/y])[\mathsf{IC}(\pi')/\beth]/\beth] \iff \mathsf{IC}(\pi)[i_0/y]$$

□

Example 17
Let us look again at the example program from figure 6.5 (p. 65) and the path $\pi_3 = 1 \xrightarrow[a_1-a_2]{\mathsf{dd}} 3, 3 \xrightarrow[a_2-a_3]{\mathsf{fd}} 5, 5 \xrightarrow[i_4-i_5]{\mathsf{fd}} 7, 7 \xrightarrow[i_6-i_2]{\mathsf{fd}} 2, (2,3), 3 \xrightarrow[a_2-a_4]{\mathsf{fd}} 9$. It is easy to see that we can apply this lemma for the maximal subpaths $1 \xrightarrow[a_1-a_2]{\mathsf{dd}} 3, 3 \xrightarrow[a_2-a_3]{\mathsf{fd}} 5$ and $3 \xrightarrow[a_2-a_4]{\mathsf{fd}} 9$. Hence, we obtain

$$\begin{aligned}
\overline{\mathsf{IC}(\pi_3)} \overset{\mathcal{M}_p}{\Longrightarrow} & (i_1 == i_1) \wedge \Diamond((i_1 != k) \wedge (a_1[i_1] == a_2[i_1]) \wedge \\
& (i_2 < 8)\, \mathcal{U}\, ((i_1 == l) \wedge (a_2[i_1] == a_3[i_1]) \wedge (i_2 < 8) \wedge (i_3 == 7) \wedge \\
& \Diamond((i_3 == 7) \wedge (i_2 < 8)\, \mathcal{U}\, ((i_4 == i_5) \wedge (i_2 < 8) \wedge \Diamond((i_6 == i_2) \wedge \\
& \Diamond((i_2 < 8) \wedge (k == k) \wedge \Diamond((k == m) \wedge !(i_2 < 8) \wedge \\
& (a_2[k] == a_4[k])))))))) \\
\Longrightarrow & \Diamond((i_1 != k) \wedge (a_1[i_1] == a_2[i_1]) \wedge \\
& (i_2 < 8)\, \mathcal{U}\, ((i_1 == l) \wedge (a_2[i_1] == a_3[i_1]) \wedge (i_2 < 8) \wedge (i_3 == 7) \wedge \\
& \Diamond((i_3 == 7) \wedge (i_2 < 8)\, \mathcal{U}\, ((i_4 == i_5) \wedge (i_2 < 8) \wedge \Diamond((i_6 == i_2) \wedge \\
& \Diamond((i_2 < 8) \wedge \Diamond((k == m) \wedge !(i_2 < 8) \wedge (a_2[k] == a_4[k])))))))))
\end{aligned}$$

6.4.2 Moving temporal operators

Next, we try to avoid using more temporal operators than necessary. State formulae over atomic formulae are relatively easy to handle. Therefore, if we are able to find a state subformula which is not satisfiable over $\mathcal{S}^{\mathbf{A}_{\mathsf{IMP}}}_{\mathbb{V},W}$, we can be sure that the LTL formula as a whole is not satisfiable either. Chances for doing so are the greater the longer such subformulae are. Hence, if there is no need to include a temporal operator, we have good reason to avoid it. There are two cases when we introduce temporal operators in information flow conditions:

1. The $\mathcal{U}$ operator when we pass along a flow or def-def dependence edge.

2. The $\mathcal{U}$ operator when we account for cycles in the PDG.

We have seen in example 14 (p. 61) with program shown in figure 6.4 (p. 60) that in general we can not get rid of the second type. However, in the first case we do not always have to include them. The key idea is that in every state we can refer to former variable values. More precisely, we have the variable value at our disposal for every statement when this has been executed the last time. Therefore, if we can make sure that there is no return to such a statement between two nodes in the path for which we generate a path condition we can move forward the until operator in data dependences. Then, we end up with a subformula of the form $\theta \mathcal{U} (\eta \mathcal{U} \zeta)$ and can apply the entailment $\aleph \mathcal{U} (\beth \mathcal{U} \urcorner) \Longrightarrow (\aleph \vee \beth) \mathcal{U} \urcorner$ from table 3.2 (p. 17) to get rid of one until operator.

We get there in two steps. First, we show that we may include at no extra cost the state formula "before" the until operator in question in the second operand of this operator. Second, we use the entailment $\aleph \wedge \beth \Longrightarrow \beth$ to remove the state formula "before" the until operator.

Lemma 18
Let $\pi \in \Pi^*(s,t)$ be an influence path and let $v, w \in \mathrm{V}(\pi)$ such that $v \prec_\pi w$, $\mathbf{def}(v) \neq \emptyset$, and $\mathbf{def}(w) \neq \emptyset$. Let π' denote the subpath of π from v to w, e_v denote the first edge and e_w the last edge of π'. If $|\pi'| = 1$, set $\Pi_{\pi'} := \{ \pi' \}$, otherwise set

$$\Pi_{\pi'} := \left\{ \rho \in \Pi(\pi') \;\middle|\; \exists \rho' : \multimap\!\!\circ(e_v) \xrightarrow[\mathrm{PDG}_p]{}{}^{*} \circ\!\!\to(e_w) : \rho = (e_v, \rho', e_w) \right\}$$

Let

$$\mathbb{V}_{\pi'} := \Big\{ (x,a,u) \in \mathbb{V}_p \Big| \exists \rho : v \xrightarrow[\mathrm{CFG}_p]{}{}^{*} w : \rho \text{ corresponds to a path } v \xrightarrow[\mathrm{PDG}_p]{}{}^{*} w \in \Pi_{\pi'} \text{ and } u \in \multimap\!\!\circ(\mathrm{E}(\rho)) \Big\}$$

denote the set of program variables in $\mathbb{V}_p$ whose value can possibly change in a state sequence corresponding to π when passing from v to w.

Let $W \subseteq \mathcal{W}$ be finite and let $\theta \in \mathsf{SF}^{\mathcal{L}_{IMP}}_{\mathbb{V}-(\mathbb{V}_{\pi'} \cup V_p),W}$ be a state formula. Suppose that $\Xi = (\xi_i)_{i \in \mathbb{N}} \in \mathcal{M}_p^W$ is a sequence of states corresponding to an execution

along π and suppose that ξ_j and ξ_k correspond to nodes v and w, respectively.[24] *Then, $(\Xi, j) \vDash \theta$ iff $(\Xi, k) \vDash \theta$.*

Proof. Let π, π', $\mathbb{V}_{\pi'}$, θ, Ξ, j, and k be as above. By definition of Ξ, j and k we know that $(\xi_j)\!\upharpoonright_{\mathbb{V}-(\mathbb{V}_{\pi'} \cup V_p)} = (\xi_k)\!\upharpoonright_{\mathbb{V}-(\mathbb{V}_{\pi'} \cup V_p)}$. Since θ is a state formula with $\mathcal{V}(\theta) \subseteq \mathbb{V} - \mathbb{V}_{\pi'}$, we have that $(\Xi, j) \vDash \theta$ iff $\xi_j \vDash \theta$ iff $\xi_k \vDash \theta$ iff $(\Xi, k) \vDash \theta$. □

Corollary 19 (Loop-independent flow and def-def dependences)
Let $v \xrightarrow[x]{fd\triangleright} w$ ($v \xrightarrow[x]{dd\triangleright} w$) be a loop-independent flow (def-def) dependence edge with respect to a variable x within a path $\pi \in \Pi^*(s,t)$. Suppose that $v \xrightarrow[x]{fd\triangleright} w$ ($v \xrightarrow[x]{dd\triangleright} w$) does not leave a loop, i. e. $\mathrm{L}^x\left(v \xrightarrow[x]{fd\triangleright} w\right) = \emptyset$ ($\mathrm{L}^x\left(v \xrightarrow[x]{dd\triangleright} w\right) = \emptyset$).

Let $(\eta, \eta') := \mathit{IC}(\pi)\,\langle\!\langle \mathcal{U}, {}^{v}\sqsupset_x^{w} \rangle\!\rangle$, let $\urcorner \in \mathcal{W} - \mathcal{W}(\mathit{IC}(\pi))$, and set

$$\theta := \mathit{IC}(\pi)\,\langle\!\langle \mathcal{U}, {}^{v}\sqsupset_x^{w} \mid \urcorner \rangle\!\rangle\,.$$

Let $(\zeta, \zeta') := \theta\,\langle\!\langle \mathcal{U}, \urcorner \rangle\!\rangle$. If $(\zeta, \zeta') = (\epsilon, \epsilon)$, set $\beta := \theta$, otherwise set $\beta := \zeta'$. Then β is a state formula.

If there is a data dependence edge before $v \xrightarrow[x]{fd\triangleright} w$ ($v \xrightarrow[x]{dd\triangleright} w$) in π let e denote the last such edge in π.[25] *For every $u \in \mathrm{L}^x(e)$ with $u \xrightarrow{cd\triangleright}{}^* w$, i. e. loop u is left by e and w is control dependent on u, replace the condition $!(u)$ for u being left by* true *in β. If e is loop-carried and there is a CFG path $\rho : v \xrightarrow[\mathrm{CFG}_p]{}{}^* w$ with $\odot\!\!\succ(e) \in \mathrm{V}(\rho)$ such that $v \xrightarrow[x,\rho]{fd\triangleright} w$ ($v \xrightarrow[x,\rho]{dd\triangleright} w$), then replace the Φ constraint for e in β by* true.

Set $\kappa := \eta\,\mathcal{U}\,(\beta[\eta'/\urcorner]))$. Then, $\theta[\kappa/\urcorner]$ is a correct influence condition for π.

Proof. We apply lemma 18. Since we only consider a path $\pi' := v \xrightarrow[x]{\mathsf{fd}\triangleright} w$ ($\pi' := v \xrightarrow[x]{\mathsf{dd}\triangleright} w$) of length 1, we do not have to care about loops. Thus, we need to show that $\mathcal{V}(\beta) \cap \mathbb{V}_{\pi'} = \emptyset$. Since $v \xrightarrow[x]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x]{\mathsf{dd}\triangleright} w$) is loop-independent, $(\mathbf{use}(v) \cup \mathbf{def}(v)) \cap \mathbb{V}_{\pi'} = \emptyset$.

First, suppose that if such an edge e exists, e does not leave a loop. Then by construction, β can contain

- rigid variables (for array dependences), but rigid variables are not contained in $\mathbb{V}_{\pi'}$;
- intrastatement conditions, but these contain only variables $y \in \mathbf{use}(v)$;
- Φ constraints, but if these are not true, e is loop independent, so with $v \xrightarrow[x]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x]{\mathsf{dd}\triangleright} w$) being loop independent, too, the Φ constraints do not contain variables in $\mathbb{V}_{\pi'}$.
- a condition for leaving loops, but by definition of β, this condition is true; and

[24] If π passes through v or w multiple times, we distinguish each occurrence of v and w in this context. If Ξ passes through the occurrence of v multiple times, we take j to be the last state of these. Conversely, if Ξ passes through the occurrence of w multiple times, we take k to be the first such state.

[25] Note that in this case $\zeta' \neq \epsilon$.

- execution conditions. Since IMP programs have structured control flow, all execution conditions for v are before v in p. Thus, if there was a variable $y \in \mathbb{V}_{\pi'}$ in these execution conditions, there would be a path $\rho : v \xrightarrow[\mathrm{CFG}_p]{}\!\!\!\blacktriangleright^* w \in \Pi_{\pi'}$ corresponding to π' that contains a back edge e to a loop predicate node u such that $u \xrightarrow{\mathsf{cd}}\!\triangleright^* v$. (If we had not $u \xrightarrow{\mathsf{cd}}\!\triangleright^* v$, then e would not help in the control flow reaching the node for variable y.) Since there is no u such that not $u \xrightarrow{\mathsf{cd}}\!\triangleright^* w$, either, we must have $u \xrightarrow{\mathsf{cd}}\!\triangleright^* w$. This is a contradiction to $v \xrightarrow[x]{\mathsf{fd}}\!\triangleright\, w$ ($v \xrightarrow[x]{\mathsf{dd}}\!\triangleright\, w$) being loop-independent.

Hence, we have that $\mathcal{V}(\beta) \cap \mathbb{V}_{\pi'} = \emptyset$. Let $U := \left\{ u \in \mathrm{L}^x(e) \;\middle|\; \text{not } u \xrightarrow{\mathsf{cd}}\!\triangleright^* w \right\}$ denote the set of loop predicate nodes left by e on which w is not control dependent. Suppose that $U \neq \emptyset$. Then β contains the nontrivial condition

$$\gamma := \bigwedge_{u \in U} !(u)$$

If we remove this condition from β, the above case applies. If not, we have to show that if $\Xi = (\xi_i)_{i \in \mathbb{N}} \in \mathcal{M}_p$ is a state sequence that corresponds to the information flow path π and if ξ_j corresponds to node w, we have that $\xi_j(\gamma) = \mathtt{T}$.

Let Ξ and ξ_j be this way. Then, there exists a state ξ_k with $k < j$ that corresponds to node v. By assumption, we have that β holds in ξ_k. Then, in particular, γ holds in ξ_k. Also $v \notin U$ because $\mathbf{def}(v) \neq \emptyset$ and $\bigcup \mathbf{def}(U) = \emptyset$.

Suppose that $\Psi = \xi_k, \ldots, \xi_j$ corresponds to a CFG path $v \xrightarrow[\mathrm{CFG}]{}\!\!\!\blacktriangleright^* w$ that passes through at least one node $u \in U$. By assumption, we have that not $u' \xrightarrow{\mathsf{cd}}\!\triangleright^* w$ for all $u' \in U$. Since u is a loop predicate node and in IMP loops can only be left if the loop predicate evaluates to `F` we have that $!(u)$ holds in ξ_j.

Note that lemma 18 (p. 71) requires that $\mathbf{def}(w) \neq \emptyset$. If we have $\mathbf{def}(w) = \emptyset$, then w corresponds to a `while` or an `if` statement in p. Since we do not have jump statements like `goto` in IMP, the next assignment statement to execute after w inevitably comes after w in p.[26] Also, v comes before w in p because $v \xrightarrow[x]{\mathsf{fd}}\!\triangleright\, w$ (or $v \xrightarrow[x]{\mathsf{dd}}\!\triangleright\, w$) is loop independent. Therefore we do not need the requirement $\mathbf{def}(w) \neq \emptyset$ here. □

Example 18

We now look at the example in figure 6.6 to see in more detail what actually happens in corollary 19. For the sake of simplicity, we consider only scalar variables and an acyclic PDG. We are interested in paths from line 2 (`x:=5;`) to line 12 (`v:=w;`). Obviously, there is only one path π in the PDG from node 2 to node 12, namely

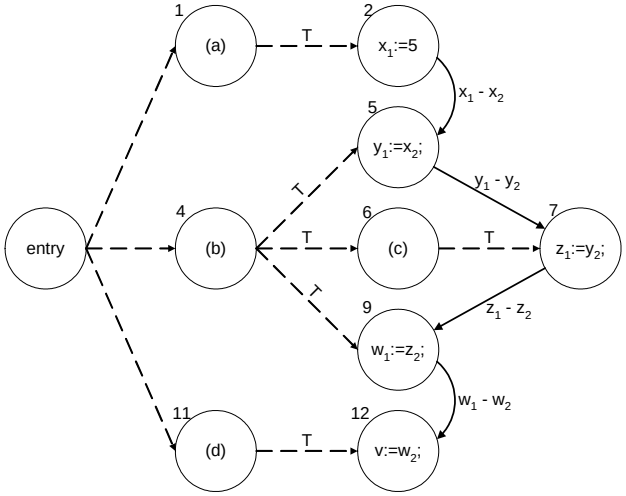
$$\pi := 2 \xrightarrow[x_1-x_2]{\mathsf{fd}}\!\triangleright\, 5, 5 \xrightarrow[y_1-y_2]{\mathsf{fd}}\!\triangleright\, 7, 7 \xrightarrow[z_1-z_2]{\mathsf{fd}}\!\triangleright\, 9, 9 \xrightarrow[w_1-w_2]{\mathsf{fd}}\!\triangleright\, 12.$$

As PDG_p is acyclic, we do not have to account for any cycles when generating the influence condition for π. Since the program does not make use of array

[26] If π continues after $v \xrightarrow[x]{\mathsf{fd}}\!\triangleright\, w$ ($v \xrightarrow[x]{\mathsf{dd}}\!\triangleright\, w$), this is trivially fulfilled because all outgoing edges from w are control dependence edges and `if` and `while` statements must not have empty bodies. If it is the last edge, we assume without loss of generality that we have an assignment statement each for both evaluation results of w after w.

Figure 6.6 Example program to illustrate temporal operators being moved.

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



variables, the simplifications for rigid variables from 6.4.1 do not apply here either. Hence, we obtain:

$$
\begin{aligned}
\mathsf{IC}(\pi) = {} & \underbrace{(a)}_{=\mathsf{E}(2)} \wedge (\underbrace{((a) \vee (b))}_{=\mathsf{E}_\delta\left(2 \xrightarrow[x_1-x_2]{\mathsf{fd}} \triangleright 5\right)} \wedge {}^{2}\beth^{5}_{x_1-x_2}) \,\mathcal{U}\, (\underbrace{\mathrm{true}}_{=\delta_l\left(2 \xrightarrow[x_1-x_2]{\mathsf{fd}} \triangleright 5\right)} \wedge \underbrace{(x_1 == x_2)}_{={}^{2}\Phi^{5}_{x_1-x_2}} \wedge \underbrace{\mathrm{true}}_{=\delta^{7}_{x_2}} \wedge \\
& \underbrace{(b)}_{=\mathsf{E}(5)} \wedge (\underbrace{(b)}_{=\mathsf{E}_\delta\left(5 \xrightarrow[y_1-y_2]{\mathsf{fd}} \triangleright 7\right)} \wedge {}^{5}\beth^{7}_{y_1-y_2}) \,\mathcal{U}\, (\underbrace{\mathrm{true}}_{=\delta_l\left(5 \xrightarrow[y_1-y_2]{\mathsf{fd}} \triangleright 7\right)} \wedge \underbrace{(y_1 == y_2)}_{={}^{5}\Phi^{7}_{y_1-y_2}} \wedge \underbrace{\mathrm{true}}_{=\delta^{7}_{y_2}} \wedge \\
& \underbrace{((b) \wedge (c))}_{=\mathsf{E}(7)} \wedge (\underbrace{(b)}_{=\mathsf{E}_\delta\left(7 \xrightarrow[z_1-z_2]{\mathsf{fd}} \triangleright 9\right)} \wedge {}^{7}\beth^{9}_{z_1-z_2}) \,\mathcal{U}\, (\underbrace{\mathrm{true}}_{=\delta_l\left(7 \xrightarrow[z_1-z_2]{\mathsf{fd}} \triangleright 9\right)} \wedge \underbrace{(z_1 == z_2)}_{={}^{7}\Phi^{9}_{z_1-z_2}} \wedge \\
& \underbrace{\mathrm{true}}_{=\delta^{9}_{z_2}} \wedge \underbrace{(b)}_{=\mathsf{E}(9)} \wedge (\underbrace{((b) \vee (d))}_{=\mathsf{E}_\delta\left(9 \xrightarrow[w_1-w_2]{\mathsf{fd}} \triangleright 12\right)} \wedge {}^{9}\beth^{12}_{w_1-w_2}) \,\mathcal{U}\, (\underbrace{!(b)}_{=\delta_l\left(9 \xrightarrow[w_1-w_2]{\mathsf{fd}} \triangleright 12\right)} \wedge \\
& \underbrace{(w_1 == w_2)}_{={}^{9}\Phi^{12}_{w_1-w_2}} \wedge \underbrace{\mathrm{true}}_{=\delta^{12}_{w_2}} \wedge \underbrace{((d) \wedge !(b))}_{=\mathsf{E}(12)} \wedge \beth))))
\end{aligned}
$$

Even though we could still simplify this condition, it contains far more $\mathcal{U}$ operators than necessary.

Let us first consider the edge $2 \xrightarrow[x_1-x_2]{\mathsf{fd}} \triangleright 5$. Then

$$
\begin{aligned}
& (\eta, \eta') := \\
\mathsf{IC}(\pi)\,\langle\!\langle \mathcal{U}, {}^{2}\beth^{5}_{x_1-x_2} \rangle\!\rangle = {} & (((a) \vee (b)) \wedge {}^{2}\beth^{5}_{x_1-x_2}, \\
& \mathrm{true} \wedge (x_1 == x_2) \wedge \mathrm{true} \wedge (b) \wedge ((b) \wedge {}^{5}\beth^{7}_{y_1-y_2}) \,\mathcal{U}\, (\mathrm{true} \wedge \\
& (y_1 == y_2) \wedge \mathrm{true} \wedge ((b) \wedge (c)) \wedge ((b) \wedge {}^{7}\beth^{9}_{z_1-z_2}) \,\mathcal{U}\, (\mathrm{true} \wedge \\
& (z_1 == z_2) \wedge \mathrm{true} \wedge (b) \wedge (((b) \vee (d)) \wedge {}^{9}\beth^{12}_{w_1-w_2}) \,\mathcal{U}\, (!(b) \wedge \\
& (w_1 == w_2) \wedge \mathrm{true} \wedge ((d) \wedge !(b)) \wedge \beth))))
\end{aligned}
$$

and

$$\mathsf{IC}(\pi)\left\langle\!\left\langle \mathcal{U},\ {}^{2}\beth^{5}_{x_1-x_2} \;\middle|\; \daleth \right\rangle\!\right\rangle = (a) \wedge \daleth$$

Hence, we set

$$\kappa := ((a) \vee (b)) \wedge {}^{2}\beth^{5}_{x_1-x_2})\ \mathcal{U}\ ((a) \wedge \eta')$$

and the altered version of $\mathsf{IC}(\pi)$ is $(a) \wedge \kappa$.

Although the corollary does not state that we can do this multiple times for different data dependence edges, it is easy to see that we can add additional constraints even for multiple flow dependences in a row if all of them satisfy the conditions from above and there are no cycles we have to take care of in between. (See corollary 20 below.) Hence, we can apply this kind of transformation to edges $5 \xrightarrow[y_1-y_2]{\mathsf{fd}}\triangleright\ 7$ and $7 \xrightarrow[z_1-z_2]{\mathsf{fd}}\triangleright\ 9$ as well and obtain:

$$\begin{aligned}\theta :=& (a) \wedge (((a) \vee (b)) \wedge {}^{2}\beth^{5}_{x_1-x_2})\ \mathcal{U}\ ((a) \wedge \mathrm{true} \wedge (x_1 == x_2) \wedge \mathrm{true} \wedge (b) \wedge \\ & ((b) \wedge {}^{5}\beth^{7}_{y_1-y_2})\ \mathcal{U}\ ((a) \wedge \mathrm{true} \wedge (x_1 == x_2) \wedge \mathrm{true} \wedge (b) \wedge \mathrm{true} \wedge (y_1 == y_2) \wedge \\ & \mathrm{true} \wedge ((b) \wedge (c)) \wedge ((b) \wedge {}^{7}\beth^{9}_{z_1-z_2})\ \mathcal{U}\ ((a) \wedge \mathrm{true} \wedge (x_1 == x_2) \wedge \mathrm{true} \wedge (b) \wedge \\ & \mathrm{true} \wedge (y_1 == y_2) \wedge \mathrm{true} \wedge ((b) \wedge (c)) \wedge \mathrm{true} \wedge (z_1 == z_2) \wedge \mathrm{true} \wedge (b) \wedge \\ & (((b) \vee (d)) \wedge {}^{9}\beth^{12}_{w_1-w_2})\ \mathcal{U}\ (!(b) \wedge (w_1 == w_2) \wedge \mathrm{true} \wedge ((d) \wedge !(b)) \wedge \beth))))\end{aligned}$$

Note that we can not apply this scheme to the last flow dependence edge $9 \xrightarrow[w_1-w_2]{\mathsf{fd}}\triangleright\ 12$, because the execution condition $\mathsf{E}(12) = (d) \wedge !(b)$ for node 12 requires that the loop has already been left whereas the execution condition $\mathsf{E}(9) = (b)$ requires that the loop predicate still holds. This is because $9 \xrightarrow[w_1-w_2]{\mathsf{fd}}\triangleright\ 12$ leaves the loop with loop predicate node 4.

Now we can proceed to the second step mentioned above in that we remove some parts of the conjunctions that now appear twice in the formula. This will make the until operators move forward. For θ, we obtain:

$$\begin{aligned}\theta \Longrightarrow \theta' :=& (((a) \vee (b)) \wedge {}^{2}\beth^{5}_{x_1-x_2})\ \mathcal{U}\ (((b) \wedge {}^{5}\beth^{7}_{y_1-y_2})\ \mathcal{U}\ (((b) \wedge {}^{7}\beth^{9}_{z_1-z_2})\ \mathcal{U}\ (\\ & (a) \wedge (x_1 == x_2) \wedge (b) \wedge (y_1 == y_2) \wedge ((b) \wedge (c)) \wedge (z_1 == z_2) \wedge (b) \wedge \\ & (((b) \vee (d)) \wedge {}^{9}\beth^{12}_{w_1-w_2})\ \mathcal{U}\ (!(b) \wedge (w_1 == w_2) \wedge ((d) \wedge !(b)) \wedge \beth))))\end{aligned}$$

Hence we need to have two temporal operator to spread the condition over two states. Simplifying θ' yields

$$\begin{aligned}\overline{\theta'} \Longleftrightarrow ((a) \vee (b))\ \mathcal{U}\ ((a) \wedge (b) \wedge (c) \wedge (x_1 == x_2) \wedge (y_1 == y_2) \wedge (z_1 == z_2) \wedge \\ ((b) \vee (d))\ \mathcal{U}\ (!(b) \wedge (d) \wedge (w_1 == w_2)))\end{aligned}$$

If we remember that, ultimately, we will look at $\Diamond\,\overline{\theta'}$ which is congruent to $\Diamond((a) \wedge (b) \wedge (c) \wedge (x_1 == x_2) \wedge (y_1 == y_2) \wedge (z_1 == z_2) \wedge ((b) \vee (d))\ \mathcal{U}\ (!(b) \wedge (d) \wedge (w_1 == w_2)))$. We see that we were able to eliminate all but one until operator, which can not be sensibly removed.

Corollary 20 (Multiple loop-independent data dependences)
Let π' be a subpath of an influence path $\pi \in \Pi^*(s,t)$. Let $\Pi_{\pi'}$ be defined as in lemma 18 (p. 71). Suppose that $\Pi = \{\, \pi' \,\}$ and every data dependence edge

$e \in \mathrm{E}(\pi') \cap D$ is loop independent and does not leave a loop, i. e. $\mathrm{L}(e) = (\emptyset, \emptyset)$. Suppose that the first and last edge of π' are data dependence edges e^* and e'.

Let θ be an influence condition for π, possibly with any of the above transformations applied to it. Set $(\eta, \eta') := \theta \langle\!\langle \mathcal{U}, \beth_{e'} \rangle\!\rangle$ where $\beth_{e'}$ is the propositional variable used for e'. Set $\vartheta := \theta \langle\!\langle \mathcal{U}, \beth_{e'} \mid \daleth \rangle\!\rangle$ where $\daleth \in \mathcal{W} - \mathcal{W}(\theta)$ is a new propositional variable. Set $(\zeta, \zeta') := \theta \langle\!\langle \mathcal{U}, \beth_{e^*} \rangle\!\rangle$ where $\beth_{e^*}$ is the propositional variable used for e^*. Set $(\beta, \beta') := (\theta \langle\!\langle \mathcal{U}, \beth_{e^*} \mid \daleth \rangle\!\rangle) \langle\!\langle \mathcal{U}, \daleth \rangle\!\rangle$. If $(\beta, \beta') = (\epsilon, \epsilon)$, set $\gamma := \vartheta \langle\!\langle \mathcal{U}, \beth_{e^*} \mid \daleth \rangle\!\rangle$. Otherwise, set $\gamma := \beta'$.

If there is a data dependence edge in π before π', let f denote the last such edge in π. For every $u \in \mathrm{L}^x(f)$ such that $u \xrightarrow{cd}{}^* \multimap\!\!\circledcirc(e')$, replace the condition $!(u)$ for loop u being left by f in γ by true*. If f is loop-carried and there is a data dependence edge $e^\sharp \in \mathrm{E}(\pi') \cap D$, say $e^\sharp = \odot\!\!\succ(e^\sharp) \xrightarrow[y]{fd} \multimap\!\!\circledcirc(e^\sharp)$ or $e^\sharp = \odot\!\!\succ(e^\sharp) \xrightarrow[y]{dd} \multimap\!\!\circledcirc(e^\sharp)$, such that there is a CFG path $\rho : \odot\!\!\succ(e^\sharp) \xrightarrow[\mathrm{CFG}_p]{}{}^* \multimap\!\!\circledcirc(e^\sharp)$ with $\odot\!\!\succ(f) \in \mathrm{V}(\rho)$ and $\odot\!\!\succ(e^\sharp) \xrightarrow[y,\rho]{fd} \multimap\!\!\circledcirc(e^\sharp)$ or $\odot\!\!\succ(e^\sharp) \xrightarrow[y,\rho]{dd} \multimap\!\!\circledcirc(e^\sharp)$, then replace the Φ constraint for f in γ by* true*.*

Set $\kappa := \eta\ \mathcal{U}\ (\gamma[\eta'/\daleth])$. Then $\vartheta[\kappa/\daleth]$ is a correct influence condition for π. This corollary can be applied to it again if neither $\multimap\!\!\circledcirc(f)$ nor $\odot\!\!\succ(e^*)$ have been $\multimap\!\!\circledcirc(e')$ in a previous application.

Proof. The key point of this corollary is it saying that we may include propagation conditions, execution conditions, conditions for loops being left, and flow dependence conditions of the forms $(y == i)$ and $(y! = i)$ not only one until operator later, but any number of dependences later as long as there are no cycles which can be attached inside π' and as long as none of the until operators is caused by a loop-carried edge.

If $|\pi'| = 1$, we essentially obtain corollary 19 (p. 72). For $|\pi'| > 1$, the argument for applying lemma 18 (p. 71) goes the same way: Since CDG_p is a tree for IMP programs and all data dependences in π' are loop independent, we have that all nodes in π' appear in p in the same order as in π. If we abstract from conditions for loops being left, γ can only contain conditions of the other types mentioned above, so $\mathcal{V}(\gamma) \cap \mathbb{V}_{\pi'} = \emptyset$ where $\mathbb{V}_{\pi'}$ is as in lemma 18 (p. 71), but we can safely include these conditions by the same arguments as in corollary 19 (p. 72). Similarly, regarding $\mathbf{def}(\multimap\!\!\circledcirc(e')) \neq \emptyset$, the same argument applies as in the proof of corollary 19 (p. 72). □

Example 19

If we look at example 18 (p. 73) again, we are now able to apply this corollary to obtain a simpler version of θ

$$
\begin{aligned}
\theta :=& (a) \wedge (((a) \vee (b)) \wedge {}^{2}\beth^{5}_{x_1 - x_2})\ \mathcal{U}\ (\mathrm{true} \wedge (x_1 == x_2) \wedge \mathrm{true} \wedge (b) \wedge \\
& ((b) \wedge {}^{5}\beth^{7}_{y_1 - y_2})\ \mathcal{U}\ (\mathrm{true} \wedge (y_1 == y_2) \wedge \mathrm{true} \wedge ((b) \wedge (c)) \wedge \\
& ((b) \wedge {}^{7}\beth^{9}_{z_1 - z_2})\ \mathcal{U}\ ((a) \wedge \mathrm{true} \wedge (x_1 == x_2) \wedge \mathrm{true} \wedge (b) \wedge \mathrm{true} \wedge (y_1 == y_2) \wedge \\
& \mathrm{true} \wedge ((b) \wedge (c)) \wedge \mathrm{true} \wedge (z_1 == z_2) \wedge \mathrm{true} \wedge (b) \wedge \\
& (((b) \vee (d)) \wedge {}^{9}\beth^{12}_{w_1 - w_2})\ \mathcal{U}\ (!(b) \wedge (w_1 == w_2) \wedge \mathrm{true} \wedge ((d) \wedge !(b)) \wedge \beth))))
\end{aligned}
$$

where we do not have to repeat execution conditions for nodes 2 and 5 after every $\mathcal{U}$ operator. However, if we simplify this new version with the same rules as above, we end up with the very same influence condition $\overline{\theta'}$.

6.4.3 Reducing the number of influence paths

In the previous subsections, we have presented some lemmata we can use to simplify an influence condition for a single path. However, in many cases, we are interested in an influence condition for which a state sequence exists whenever there is an influence, i. e. in an influence condition between statements and not for a single influence path. In this case, we want to compute $\Diamond\,\mathsf{IC}(s,t)$ for two statements s and t, which is the disjunction of influence conditions for all influence paths between s and t. Here, we show how to exploit the absorption law for disjunctions (cf. table 3.1 (p. 16)) to get rid of some influence paths.

Lemma 21
Let π, ρ be two influence paths. Suppose that there is a prefix π^* of π such that π^* is an information flow path and $\Diamond\, \mathit{IC}(\pi) \iff \Diamond\, \mathit{IC}(\pi^*)[\mathit{IC}(\rho)/\sqsupseteq]$. Then $\Diamond\, \mathit{IC}(\pi) \Longrightarrow\!\!\!\!\!= \Diamond\, \mathit{IC}(\rho)$.

Proof. Let $W = \mathcal{W}(\mathsf{IC}(\pi)) \supseteq \mathcal{W}(\mathsf{IC}(\pi^*)) \cup \mathcal{W}(\mathsf{IC}(\rho))$ and $\Xi = (\xi_i)_{i\in\mathbb{N}} \in \mathcal{M}_{\mathbb{V},W}^{\mathbf{A}_{\mathsf{IMP}}}$. Suppose that $\Xi \vDash \Diamond\,\mathsf{IC}(\pi)$. Then also $\Xi \vDash \Diamond\,\mathsf{IC}(\pi^*)[\mathsf{IC}(\rho)/\sqsupseteq]$. Since π^* is an information flow path and by construction of $\mathsf{IC}(\pi^*)$, there is a $k \in \mathbb{N}$ such that $(\Xi, k) \vDash \mathsf{IC}(\rho)$. Hence $\Xi \vDash \Diamond\,\mathsf{IC}(\rho)$. □

Lemma 22
Let π, ρ be two influence paths such that ρ is a prefix of π. Then $\mathit{IC}(\pi)[\text{true}\,/\sqsupseteq] \Longrightarrow\!\!\!\!\!= \mathit{IC}(\rho)[\text{true}\,/\sqsupseteq]$.

Proof. Let π' denote the suffix of π such that $\pi = \rho, \pi'$. Then π' is an influence path and $\mathsf{IC}(\pi) \iff \mathsf{IC}(\rho)[\mathsf{IC}(\pi')/\sqsupseteq]$. Thus

$$\mathsf{IC}(\pi)[\text{true}\,/\sqsupseteq] \iff (\mathsf{IC}(\rho)[\mathsf{IC}(\pi')/\sqsupseteq])[\text{true}\,/\sqsupseteq] = \mathsf{IC}(\rho)[(\mathsf{IC}(\pi')[\text{true}\,/\sqsupseteq])/\sqsupseteq].$$

Since $\mathsf{IC}(\pi')[\text{true}\,/\sqsupseteq] \Longrightarrow\!\!\!\!\!= \text{true}$, we have

$$\mathsf{IC}(\rho)[(\mathsf{IC}(\pi')[\text{true}\,/\sqsupseteq])/\sqsupseteq] \Longrightarrow\!\!\!\!\!= \mathsf{IC}(\rho)[\text{true}\,/\sqsupseteq].$$

□

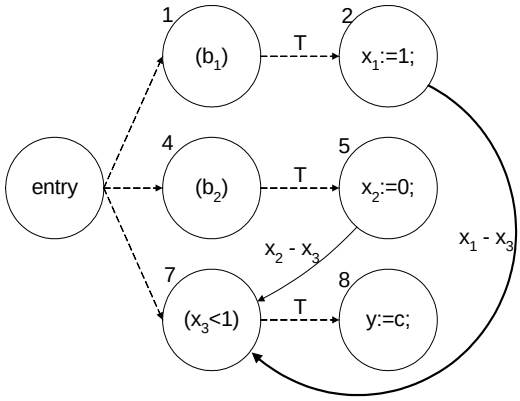
These two lemmata allow us to omit some influence paths when we generate $\mathsf{IC}(s,t)$. For examples, see sections 8.2 and 8.3 below.

6.5 Examples

In this section we provide some examples to explain why we used some of the constructions above the way we did even though they may not always be obvious at a first glance. In particular, we look at the following issues:

Figure 6.7 Example for Φ constraints.

```
1  if (b) {
2      x := 1;
3  }
4  if (b) {
5      x := 0;
6  }
7  if (x < 1) {
8      y := c;
9  }
```



- Φ constraints for data dependence edges,
- true as Φ constraint for cyclic flow dependence edges, and
- Having the values $\perp_a$, $\perp_b$, and $\perp_i$ for typed variables.

Example 20 (Φ constraints for data dependence edges)
Let us look at the program in figure 6.7 and its PDG. If we are interested in whether line 2 influences line 8, we see that this can possibly happen only if this influence passes along $2 \xrightarrow[x_1 - x_3]{\text{fd}} 7$. Hence, we require that $x_1 == x_3$. Note however that if line 2 is executed then line 5 will also be executed, so the value 1 can never reach line 7 since it will inevitably be overwritten in line 5.

If we did not include the Φ constraint in path conditions, the path condition for lines 2 and 8 would have read

$$\theta \Longleftrightarrow (b_1) \wedge \Diamond(x_3 < 1).$$

Note that there are state sequences in $\mathcal{M}_p$ that model θ even though there is no such influence. For example, take any initial state ξ with $\xi(b) = \texttt{F}$ and $\xi(x) = 1$ and construct the state sequence as described in 6.1.2. If, however, we include the

Figure 6.8 Example program for true as Φ constraint for cyclic flow dependence edges and its PDG.

```
1  i := 0;
2  while (i < 2) {
3      i := i + 1;
4  }
```

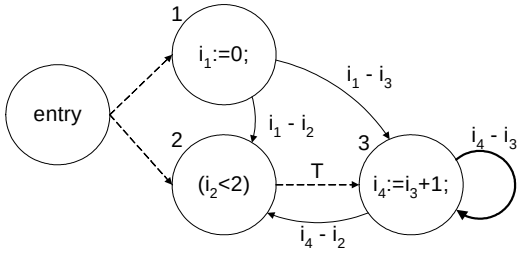
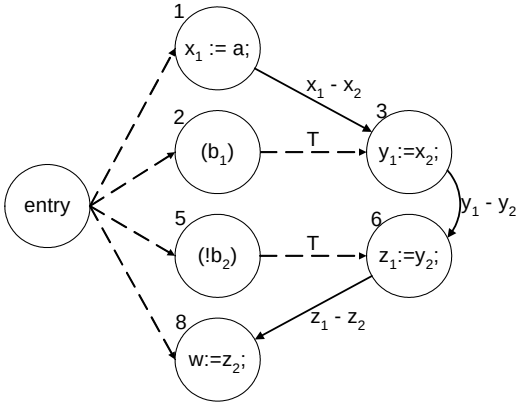


Figure 6.9 Example program and its PDG for $\perp$ values in $\mathbf{A}_{\mathsf{IMP}}$.

```
1  x := a;
2  if (b) {
3      y := x;
4  }
5  if (!b) {
6      z := y;
7  }
8  w := z;
```



additional constraint, we obtain

$$\mathsf{PC}(2,8) \Longleftrightarrow (b_1) \wedge \Diamond((x_1 == x_3) \wedge (x_3 < 1))$$

Now, there does not exist a state sequence in $\mathcal{M}_p$ that models $\mathsf{PC}(2,8)$.

Hence, Φ constraints do make path and influence conditions stronger.

Example 21 (true **as Φ constraint for cyclic flow dependence edges)**
Let us look at the loop carried flow dependence edge $3 \xrightarrow[i_4 - i_3]{\mathsf{fd}} 3$ of the program p shown in figure 6.8.

If we used the ordinary Φ constraint $(i_4 == i_3)$ for this edge, then any path condition that contains $(i_4 == i_3)$ would not be satisfiable because $(i_4 == i_3) \overset{\mathcal{M}_p}{\Longleftrightarrow}$ false. While it is safe to use Φ constraints for noncyclic flow dependence edges (because the source node is not revisited after we left it before we reach the target node which is different from the source node), we are not able to refer to the former value of the variable occurrence i_4 because i_4 is updated whenever i_3 is also updated. For cyclic def-def dependences, the array cell in question must not be modified, hence the ordinary Φ constraint is correct. Note that this issue does not affect influence conditions because they do not contain Φ constraints for cyclic flow dependence edges.

Example 22 (It is sensible to have $\perp_i$)
In this example, we demonstrate that initializing every variable in the PDG with an uninitialized value is sensible. Consider the program p shown in figure 6.9.

Here, the influence condition for lines 1 and 8 is given by

$$\overline{\mathsf{IC}(1,8)} \Longleftrightarrow \Diamond((b_1) \wedge (x_1 == x_2) \wedge \Diamond((!b_2) \wedge (y_1 == y_2) \wedge \Diamond(z_1 == z_2)))$$

We can simplify this condition with corollary 20 (p. 75) to

$$\overline{\mathsf{IC}(1,8)} \overset{\mathcal{M}_p}{\Longleftrightarrow} \Diamond((b_1) \wedge (x_1 == x_2) \wedge (!b_2) \wedge (y_1 == y_2) \wedge (z_1 == z_2))$$

If we did not have each variable in $\mathcal{V}(\mathsf{IC}(1,4))$ initialized to its undefined value in $\perp$, but if we allowed every variable to have any possible initial value,[27] then there would also be a state sequence in $\mathcal{M}_p$ whose first state ξ would satisfy $\xi(x_1) = \xi(x_2)$, $\xi(y_1) = \xi(y_2)$, $\xi(z_1) = \xi(z_2)$, $\xi(b_1) = \mathtt{T}$, and $\xi(b_2) = \mathtt{F}$. Any such state sequence with ξ as its first state satisfies $\overline{\mathsf{IC}(1,4)}$.

If, however, we do initialize all variables to the undefined value until they are first used, we obtain $\overline{\mathsf{IC}(1,4)} \overset{\mathcal{M}_p}{\Longleftrightarrow}$ false.

6.6 Comparison with Boolean path conditions

We now want to look at the relation between Boolean path conditions as proposed by Snelting, Krinke and Robschink in [Sne96, RS02, Rob04, SRK] (cf. chapter 5) and the LTL influence conditions from above.

In this section, we first see that they are not more precise than LTL path conditions. Then, we look at some examples that show how LTL formulae improve the precision of path conditions.

6.6.1 From temporal to Boolean path conditions

We now show that Boolean path conditions can be derived from LTL path conditions. More precisely, we show that for every information flow path $\pi \in \Pi^*(s,t)$ and an appropriate LTL influence condition $\mathsf{IC}(\pi)$ there is a cycle-free path $\pi' \in \Pi^{**}(s,t)$ that is contained in π such that, if we have a state sequence $\Xi \in \mathcal{M}_p$ that carries information along π and thus satisfies $\mathsf{IC}(\pi)$, we can construct a satisfying assignment for the appropriate Boolean path condition $\mathsf{BPC}(\pi')$ for π' from Ξ where we assume that we apply the approach of variables being separated whenever necessary (cf. sections 5.4 and 5.6).

Theorem 23
Let $\pi \in \Pi^*(s,t)$ be an influence path from s to t. Suppose $\Xi = (\xi_i)_{i\in\mathbb{N}} \in \mathcal{M}_p$ is a state sequence that carries information along π and satisfies the influence condition $\theta \in \mathit{LTL}^{\mathcal{L}_{IMP}}_{\mathbb{V},W}$ (for some finite $W \subseteq \mathcal{W}$ which has been simplified with lemma 18 (p. 71) and corollaries 19 (p. 72) and 20 (p. 75) as far as possible. Let θ' denote the formula obtained from $\overline{\theta}$ if we remove all "unnecessary" until operators like we have shown in example 18 (p. 73).

Let π' denote the path we obtain if we remove all cycles from π. Let η denote the Boolean path condition for π' where we have separated variables to account for loop-carried data dependences and extra cycles.

Then, $\Xi \vDash \theta'$ and there is a satisfying assignment for η.

[27] Note that it is sensible to assign every variable a value in every state. Otherwise, a state would be a partial function on the set $\mathbb{V}$ of variables which is equivalent to total functions with a special image value representing "undefined". Neither can we force all variables to be initialized to a single value because we would like to compute the values of input variables, which are not specifically marked, that prove the influence being possible.

Proof. $\Xi \vDash \theta'$ follows from the definition of θ and θ'.

Now, we show that if we assign the variable values from some states in Ξ to the variables in η, we get a satisfying assignment for η. η contains three different types of constraints:

- Execution conditions
- Φ constraints
- Conditions for information flow along dependence edges (global array constraints)

In the following we look at each of them in this order.

First, we take care of the **execution conditions**. Since $\mathrm{V}(\pi') \subseteq \mathrm{V}(\pi)$, we have that every execution condition in η also appears in θ'. Moreover, for each execution condition β in η that corresponds to node v_β in π', we can identify one execution condition β' in θ' that was generated from v_β in π. By construction, we have that $\beta' \rightarrowtail \beta$. Suppose ξ_i belongs to node v_β. Then $\xi_i(\beta') = \mathtt{T}$ because $\Xi \vDash \theta'$. Hence, $\xi_i(\beta) = \mathtt{T}$, too, so we can take the assignment of ξ_i for this execution condition. Moreover, for $x \in \mathcal{V}(\beta') \supseteq \mathcal{V}(\beta)$, we have $\xi_i(x) \notin \bot$, so ξ_i is a valid assignment to $\mathbb{V}$ over $\mathbf{A}'_{\mathsf{IMP}}$.

What remains to be shown is that these assignments can be combined to a satisfying assignment for the conjunction of all execution conditions. If two execution conditions are part of the same maximal conjunction in θ', we have that ξ_i satisfies both of them, so both assignments we obtain from ξ_i do not contradict each other.

So suppose that we need to combine the assignments of two execution conditions γ and γ' that are not part of the same maximal disjunction. Then, by construction of θ', we have that there is at least one until operator between γ and γ' in θ'. Let κ denote the smallest subformula of θ' that contains both γ and γ' as subformulae. Without loss of generality, we assume that κ is a subformula of the form $\gamma \wedge \ldots \wedge \ldots \mathcal{U} (\ldots \mathcal{U} (\ldots \wedge \gamma' \wedge \ldots)$. Then, there are two states ξ_i and ξ_j in Ξ with $i \leq j$ such that $\xi_i(\gamma) = \mathtt{T} = \xi_j(\gamma')$.

By definition, every $\mathcal{U}$ operator in κ corresponds to one or more of the following cases:

1. A loop-carried data dependence edge,
2. A cycle that can be inserted in π at that node,
3. A data dependence edge leaving a loop.

In the first and second case, the variables in η have been separated at the appropriate positions. Since, we can obviously combine the assignments for execution conditions that do not have common variables, we can combine those for γ and γ' obtained from ξ_i and ξ_j.

In the third case, γ and γ' can have common variables. Let v denote the node corresponding to γ and v' the one for γ'. Then, for every $x \in \mathcal{V}(\gamma) \cap \mathcal{V}(\gamma')$ we

have that $x \in \mathbf{use}(u)$ for some node u with $u \xrightarrow{\mathsf{cd}\triangleright}{}^{*} v$ and $u \xrightarrow{\mathsf{cd}\triangleright}{}^{*} v'$ because γ and γ' are execution conditions from control dependence.

Suppose for contradiction that such an x occurs in two different atomic formulae in γ and γ'. We know that IMP allows only structured control flow and that in the CDG $(V', C, v_e, \overline{\alpha})$ for every $\Lambda \in \overline{\alpha}(C)$ we have $|\Lambda| \leq 1$. Since x identifies u, by construction of $\mathsf{E}_{\mathsf{cd}}(u)$, x can appear in two different atomic formulae iff these are $\cdot(u)$ and $\cdot(!(u))$. This can only happen if u is an `if` predicate node and v and v' are in the then and else part of the `if` statement. However, every CFG path from one branch of an `if` statement to the other contains a back edge, hence there must be a loop-carried data dependence between v and v' in π' and thus γ and γ' cannot have common variables. Hence, every variable occurs in only one atomic formula, so the assignments for execution conditions can be combined.

Next, we look at Φ **constraints**. We have introduced two types of Φ constraints:

1. Φ constraints that originate from Φ functions

2. Φ constraints generated by data dependence edges

Φ constraints of the first type do not appear in LTL path conditions because they are integral part of the model set $\mathcal{M}_p$. By definition of SSA and state sequences, we have for every state sequence $\Psi \in \mathcal{M}_p$ that every initialized variable obeys these Φ constraints in every state of Ψ and between states in such a sequence. Hence, assignments constructed from Ξ always satisfy Φ constraints from Φ functions.

As for the second type, these are only generated for data dependence edges that are not loop carried, cf. equation 5.8 (p. 38).

Let $(x_v == x_w)$[28] denote such a Φ constraint for a data dependence edge $v \xrightarrow[x]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x]{\mathsf{dd}\triangleright} w$). Let $e_1, \ldots, e_m$ denote the maximal subpath of π' such that we have not separated the variables along $e_1, \ldots, e_m$ and $e_i = v \xrightarrow[x]{\mathsf{fd}\triangleright} w$ or $e_i = v \xrightarrow[x]{\mathsf{dd}\triangleright} w$ for some $1 \leq i \leq m$. Then, no variable in any subformula of η that corresponds to $e_1, \ldots, e_m$ is related with variables that originate from parts of π' other than $e_1, \ldots, e_m$, except via Φ constraints from Φ functions.

By definition, $x_v \in \mathbf{def}(v)$. In particular, there is no control dependence edge $v \xrightarrow{\mathsf{cd}\triangleright} u$ in the PDG, i. e. v (and thus x_v) does not occur in execution conditions. Moreover, Φ constraints from Φ functions do not impose constraints on x_v that are not automatically satisfied by Ξ.

We know that there is a state ξ_k in Ξ such that $\xi_k(x_v == x_w) = \mathtt{T}$ because $(x_v == x_w)$ is an atomic formula in θ'. Without loss of generality, let ξ_k be the state that corresponds to the node which the maximal conjunction that contains $(x_v == x_w)$ belongs to.

Since all e_j, $1 \leq j \leq m$, are loop independent, the nodes $(v_j)_{0 \leq j \leq m}$, where $v_0 = \odot\!\!\succ(e_1)$ and $v_j = \prec\!\!\odot(e_j)$ $(1 \leq j \leq m)$, are pairwise disjoint and occur in this order in the program p. Since x_v strictly belongs to node v, x_v does not occur

[28] Although not all Φ constraints are of the form $e_1 == e_2$ for some expressions $e_1, e_2 \in \mathfrak{E}$, e. g. if $(\!|x_v|\!) = \mathtt{bool}$, we write also $x_v == x_w$ for these.

in another Φ constraint from data dependence edges in η. Nor does x_v occur in execution conditions. Hence, we can assign to both x_v and x_w the value $\xi_k(x_w)$ without creating a contradiction in the common assignment.

At last, we show that we can also incorporate the δ **constraints** for array variables into the assignments. As we have seen in example 9 (p. 41) above, if we restrict ourselves to cycle-free information flow paths when generating Boolean path conditions, we have to include global array constraints (cf. equation 5.14 (p. 40)) in our path condition. In order to avoid confusion with variable renamings due to variable separation in paths, we do not follow equation 5.15 (p. 40), but we use the distributivity law of the $\vee$ operator to include an appropriate array constraint for every path, i. e. we assume that the Boolean path condition for π' has been generated by equation 5.16 (p. 41) and

$$\mathsf{BPC}(\pi') := \bigwedge_{v \in \mathrm{V}(\pi')} \mathsf{E}_{\mathsf{cd}}(v) \wedge \bigwedge \Phi_{\pi'} \wedge \bigwedge_{e \in \mathrm{E}(\pi') \cap D} \Phi(e) \wedge \bigwedge_{v \xrightarrow[a]{\mathsf{fd}} w \in A'_{\pi'}} \delta_G(v \xrightarrow[a]{\mathsf{fd}} w) \tag{6.21}$$

where $A'_{\pi'} := \left\{ v \xrightarrow[a]{\mathsf{fd}} w \in \mathrm{E}(\pi) \;\middle|\; (\!|a|\!) = \texttt{array} \right\}$ is the set of all flow dependence edges with respect to array variables in π'.

Suppose we consider $\delta_G(v \xrightarrow[a]{\mathsf{fd}} w)$ for some $v \xrightarrow[a]{\mathsf{fd}} w \in A'_{\pi'}$. Let $\rho := v_0 \xrightarrow[a]{\mathsf{dd}} v_1, \ldots, v_{m-2} \xrightarrow[a]{\mathsf{dd}} v_{m-1}, v_{m-1} \xrightarrow[a]{\mathsf{fd}} v_m$ denote the maximal subpath in π such that $v_{m-1} \xrightarrow[a]{\mathsf{fd}} v_m = v \xrightarrow[a]{\mathsf{fd}} w$. Then, by definition of $\delta_G(v \xrightarrow[a]{\mathsf{fd}} w)$ we have that $\delta_\pi(\rho) \rightarrowtail \delta_G(v \xrightarrow[a]{\mathsf{fd}} w)$[29] where $\delta_\pi(\rho)$ is the δ constraint from equation 5.10 (p. 39). We now construct a satisfying assignment for $\delta_\pi(\rho)$ from Ξ. Note that every atomic formula of the conjunction $\gamma := \bigwedge_{k=1}^{n-1}(y! = i_k) \wedge (y == i_n)$ is also an atomic formula in θ and θ' where y is the rigid variable used in θ' and i_j is as in equation 5.10 (p. 39) for $0 \leq j \leq n$. Hence, we have states $\xi_{k_1}, \ldots, \xi_{k_n} \in \Xi$ that correspond to these atomic formulae such that $\xi_{k_j}(y! = i_j) = \mathtt{T}$ for $1 \leq j < n$, and $\xi_{k_n}(y == i_n) = \mathtt{T}$ and $k_1 \leq k_2 \leq \ldots \leq k_n$.

We use each of these states to obtain a satisfying assignment for γ: First, let ψ_j denote the assignment obtained from ξ_{k_j} $(1 \leq j \leq n)$. Suppose that $\xi_{k_j} \neq \xi_{k_{j'}}$. Then $\mathcal{V}\left(i_{k_j}\right) \cap \mathcal{V}\left(i_{k_{j'}}\right) = \emptyset$ because if $v_{k_j} = v_{k_{j'}}$ then the variables in θ' have been separated when we generated η (if $v_{k_j} \neq v_{k_{j'}}$, then by definition of the PDG). Since y is rigid, i. e. $\xi_{k_j}(y) = \xi_{k_{j'}}(y)$, it follows that we can combine ψ_j and $\psi_{j'}$ to an assignment that satisfies both atomic formulae. By induction we obtain an assignment ψ' that satisfies all atomic formulae in γ. As $(y == i_0)$ is an atomic formula in θ', too, we have that there is also a state $\xi_{k_0} \in \Xi$ such that $k_0 < k_1$ and $\xi_{k_0}(y) = \xi_{k_0}(i_0) \neq \bot_i$. With the same argument as before we have that we can incorporate the corresponding assignment ψ_{k_0} into ψ'. Let ψ denote this new assignment. Hence, we have a satisfying assignment for $\delta_\pi(\rho)$ and thus also for $\delta_G(v \xrightarrow[a]{\mathsf{fd}} w)$.

Next, we show that the assignment ψ is compatible with all other assignments we constructed so far. Regarding other assignments for global δ constraints, we note that they do not share common variables. Otherwise, we would have to pass

[29] We assume here that $\delta_\pi(\rho)$ uses the same variables that have been used in $\delta_G(v \xrightarrow[a]{\mathsf{fd}} w)$.

along the def-use edge a second time, but then, since ρ_λ in equation 5.13 (p. 40) is cycle-free for $\lambda \in \Lambda$, there would have to be a loop-carried edge in between which causes the variables to be separated.

With respect to execution conditions, they can only share variables from $\mathcal{V}(i_n)$ because the **def** sets of control flow statements are empty. Since the values for these variables are obtained from the same state as those in the assignment for the execution conditions, they do not contradict each other. As ψ has been constructed out of states from a state sequence in $\mathcal{M}_p$, ψ automatically satisfies the Φ constraints of the first type. For the other Φ constraints we have already seen that they are also part of the temporal influence condition. Again, since we used the same states to construct the assignment ψ as we did for these Φ constraints, the assignments must be compatible.

Consequently, we have shown that we can construct a satisfying assignment for η from Ξ. □

6.6.2 Examples

As we have seen in the last section, if there is an influence between two statements we are able to derive a satisfying assignment for the Boolean path condition from the state sequence that models the corresponding LTL formula. Now, we show that temporal path conditions are stronger than Boolean path conditions in the sense that they improve precision, i. e. for cases in which in fact no influence is possible, LTL path conditions can sometimes be unsatisfiable whereas the Boolean path condition for this case is satisfiable. We present some examples for this.

6.6.2.1 Additional constraints in temporal path conditions

Boolean path conditions concentrate mainly on execution conditions from control dependence and, in our setting, δ constraints for arrays. Temporal path conditions as presented in the last chapter exploit a number of additional constraints such as intrastatement conditions and conditions for leaving loops. Clearly, these extra constraints make the conditions stronger.

Example 23 (Intrastatement conditions)
We consider the program p and its PDG in figure 6.10, which is already in SSA form and has an acyclic PDG.

The influence condition $\mathsf{IC}(1,6)$ yields

$$\overline{\mathsf{IC}(1,6)} \Longleftrightarrow \Diamond(((b_1 \&\& b_2) || (!(b_1) \&\& !(b_2))) \wedge \overbrace{(x_2 < 5)}^{=\delta^3_{b_2}} \wedge ((x_2 < 5) \&\& !(b_2)) \wedge \Diamond((y_1 == y_2) \wedge \underbrace{(0! = 0)}_{=\delta^6_{y_2}}))$$

Clearly $(0! = 0) \Longleftrightarrow$ false, hence $\overline{\mathsf{IC}(1,6)} \nrightarrow$ false. If we look only at $\mathsf{IC}(1,3)$, we get:

$$\overline{\mathsf{IC}(1,3)} \Longleftrightarrow \Diamond(((b_1 \&\& b_2) || (!(b_1) \&\& !(b_2))) \wedge \overbrace{(x_2 < 5)}^{=\delta^3_{b_2}})$$

Figure 6.10 Example program for intrastatement conditions and its PDG.

```
1  b := true;
2  x := 8;
3  if ((x < 5) && !(b)) {
4       y := 1;
5  }
6  z := 0 * y;
```

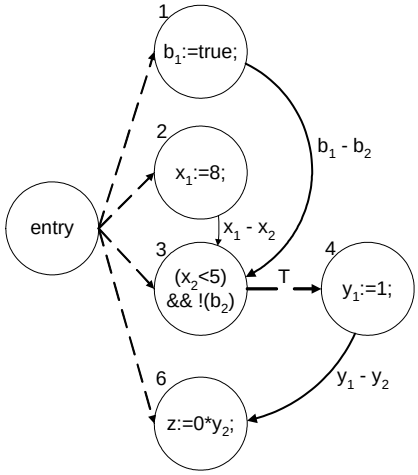
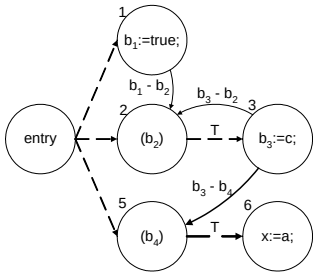


Figure 6.11 Example program (left), its SSA transform (center) and its PDG (right) for loop termination constraints in execution conditions.

```
1 b := true;          1 b1 := true;
2 while (b) {         2 while (b2:=Φ(b1,b3), b2) {
3     b := c;         3     b3 := c;
4 }                   4 }
5 if (b) {            5 if (b4:=Φ(b2), b4) {
6     x := a;         6     x := a;
7 }                   7 }
```



We know that $\Diamond(x_2 < 5) \nprec^{\mathcal{M}_p} \Diamond(x_1 < 5) \nprec^{\mathcal{M}_p}$ false. Hence $\overline{\mathsf{IC}(1,3)} \nprec^{\mathcal{M}_p}$ false. The Boolean path condition as defined in equation 5.7 (p. 37) gives

$$\mathsf{BPC}(1,6) \nleftrightarrow ((x_2 < 5)\&\&!(b_2)) \wedge (x_1 == x_2) \wedge (((b_1)\&\&(b_2))||(!(b_1)\&\&!(b_2))) \wedge (y_1 == y_2)$$

On first sight, this condition is satisfiable. If we include that $x_1 = 8$ and $b_1 = \text{true}$, then we have

$$\mathsf{BPC}(1,6)[8/x_1, \text{true}/b_1] \nleftrightarrow \text{false}$$

However, $\mathsf{BPC}(1,3) \nleftrightarrow ((b_1\&\&b_2)||(!(b_1)\&\&!(b_2)))$ which is satisfiable. Of course, intrastatement conditions could be included in Boolean path conditions, too.

Example 24 (Conditions for leaving loops)
We start again with a small IMP program p, this time the one shown in figure 6.11.

If we look at lines 3 and 6, we obtain for $\mathsf{IC}(3,6)$:

$$\overline{\mathsf{IC}(3,6)} \Longleftrightarrow (b_2) \wedge ((b_2) \vee (b_4)) \,\mathcal{U}\, (((b_3\&\&b_4)||(!(b_3)\&\&!(b_4))) \wedge \overbrace{(b_4)}^{=\mathsf{E}_{\mathsf{cd}}(6)} \wedge \overbrace{!(b_2)}^{=\mathsf{E}_{\circlearrowleft}(6)})$$

Suppose for contradiction there was a state sequence $\Xi = (\xi_i)_{i\in\mathbb{N}} \in \mathcal{M}_p$ such that $\Xi \vDash \Diamond\,\overline{\mathsf{IC}(3,6)}$. Then, there would have to be a state ξ_k in Ξ for some $k > 0$ such that $\xi_k(b_4) = \mathtt{T}$ and $\xi_k(b_2) = \mathtt{F}$. From the construction of Ξ we also would have that $\xi_k(b_4) = \xi_k(b_2)$, a contradiction. Thus, $\overline{\mathsf{IC}(3,6)}$ is not satisfiable over $\mathcal{M}_p$. Consequently, there is no influence from line 3 to line 6.

The Boolean path condition for lines 3 and 6 is

$$\begin{aligned}\mathsf{BPC}(3,6) \nleftrightarrow &(b_2) \wedge (b_4) \wedge ((b_2 \equiv b_1) \vee (b_2 \equiv b_3)) \wedge (b_4 \equiv b_2) \wedge \\ &((b_3 \&\& b_4) || (!(b_3) \&\& !(b_4))) \\ \nleftrightarrow &(b_2) \wedge (b_3) \wedge (b_4)\end{aligned}$$

Clearly, $\mathsf{BPC}(3,6)$ is satisfiable even though no influence is possible.

In contrast to intrastatement conditions, loop termination conditions, both in execution conditions and along data dependence edges, can not be easily included in Boolean path conditions because they refer to variable values at different program states.

Example 25 (Arrays and def-def dependences)
We look again at example 7 (p. 40). Clearly, there is no influence from line 1 to 4 because the value of x can never reach y for it is either in the wrong array cell ($i \neq 1$) or overwritten in line 3. Nevertheless, the Boolean path condition is equivalent to true (subject to Φ constraints) because all execution conditions are trivial and the constraint ($1 == 1$) in the δ constraint (cf. equation 5.12 (p. 40)) makes it equivalent to true, too.

However, the temporal path condition gives

$$\overline{\mathsf{IC}(1,4)} \overset{\mathcal{M}_p}{\Longleftrightarrow} \Diamond((1! = 1) \wedge (i == 1)) \Longleftrightarrow \text{false}$$

This is due to LTL path conditions handling arrays more precisely which could also be done for Boolean path conditions. Unfortunately, this way the path conditions become larger.

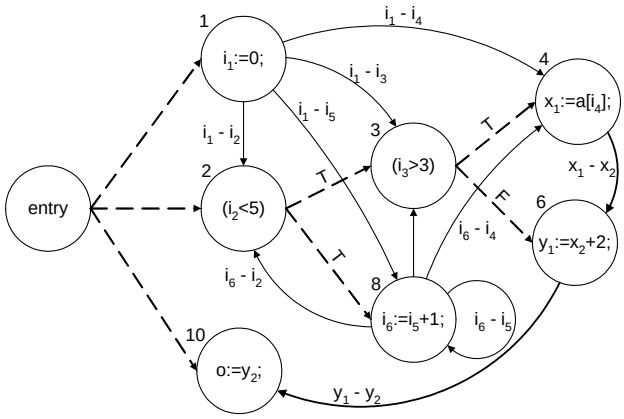
6.6.2.2 Exploiting temporal aspects

Even if we include only those constraints in temporal path conditions that are also used in Boolean path conditions, temporal path conditions are still strictly stronger than their Boolean counterparts. The reason for this is that whenever in Boolean path conditions variables are separated (or some constraints omitted), the path condition becomes less precise. This separation is done by the $\mathcal{U}$ operators in LTL formulae. However, they do not completely separate the variables. We always have our set of possible state sequences $\mathcal{M}_p$ that imposes some constraints on the variables so that they do remain related.

Example 26 (Loop-carried data dependences)
First, we look at loop-carried data dependence edges in the program shown in figure 6.12. We are interested in an influence from line 4 to line 10.

Figure 6.12 Loop-carried data dependences example program and its PDG.

```
 1  i := 0;
 2  while (i < 5) {
 3      if (i > 3) {
 4          x := a[i];
 5      } else {
 6          y := x + 2;
 7      }
 8      i := i + 1;
 9  }
10  o := y;
```



The only information flow path in $\Pi(4,10)$ is $\pi := 4 \xrightarrow[x_1-x_2]{\mathsf{fd}} 6,6 \xrightarrow[y_1-y_2]{\mathsf{fd}} 10$ where $4 \xrightarrow[x_1-x_2]{\mathsf{fd}} 6$ is loop-carried by loop 2 and $6 \xrightarrow[y_1-y_2]{\mathsf{fd}} 10$ leaves loop 2. Hence, we obtain

$$\begin{aligned}\overline{\mathsf{IC}(4,10)} \Longleftrightarrow &(i_2 < 5) \wedge (i_3 > 3) \wedge (i < 5)\, \mathcal{U}\, ((i_2 < 5) \wedge !(i_3 > 3) \wedge (x_1 == x_2) \wedge \\ &\Diamond(!(i_2 < 5) \wedge (y_1 == y_2)))\end{aligned}$$

Note that for all state sequences $\Xi = (\xi_i)_{i\in\mathbb{N}} \in \mathcal{M}_p$ we have that $\xi_j(i_3) \leq \xi_k(i_3)$ for all $0 \leq j < k$. Hence, if we had $\Xi \vDash \Diamond\, \overline{\mathsf{IC}(4,10)}$, then there would be a $j \in \mathbb{N}$ such that $\xi_j(i_3 > 3) = \mathtt{T}$, i. e. $\xi_j(i_3) > 3$, and a $k \geq j$ such that $\xi_k(!(x_3 > 3)) = \mathtt{T}$, i. e. $\xi_k(i_3) \leq 3$, a contradiction. Hence, $\Diamond\, \overline{\mathsf{IC}(4,10)} \rightarrowtail^{\mathcal{M}_p} \mathsf{false}$. Therefore, no influence between lines 4 and 10 is possible.

If we look at Boolean path conditions, then this temporal aspect can not be properly expressed. The Boolean path condition with variable separation along loop-carried flow dependence edges is given by

$$\begin{aligned}\mathsf{BPC}(4,10) \nrightarrow\!\!\!\!\!\!\leftarrow &(i_2^{(1)} < 5) \wedge (i_3^{(1)} > 3) \wedge (i_2^{(2)} < 5) \wedge !(i_3^{(2)} > 3) \wedge \\ &((i_2^{(1)} \equiv i_1^{(1)}) \vee (i_2^{(1)} \equiv i_6^{(1)})) \wedge ((i_3^{(1)} \equiv i_1^{(1)}) \vee (i_3^{(1)} \equiv i_6^{(1)})) \wedge \\ &((i_2^{(2)} \equiv i_1^{(2)}) \vee (i_2^{(2)} \equiv i_6^{(2)})) \wedge ((i_3^{(2)} \equiv i_1^{(2)}) \vee (i_3^{(2)} \equiv i_6^{(2)})) \wedge \\ &(y_1^{(2)} == y_2^{(2)})\end{aligned}$$

Although we have both $(i_3^{(1)} > 3)$ and $!(i_3^{(2)} > 3)$ as constraints in the formula, there is no way to incorporate the additional knowledge that i_3 must satisfy the former before the latter.

On the other hand, if we use the approach of omitting contradictory constraints, we get

$$\begin{aligned}\mathsf{BPC}(4,10) \nrightarrow\!\!\!\!\!\!\leftarrow &(i_2 < 5) \wedge !(i_3 > 3) \wedge \\ &((i_2 \equiv i_1) \vee (i_2 \equiv i_6)) \wedge ((i_3 \equiv i_1) \vee (i_3 \equiv i_6)) \wedge (y_1 == y_2)\end{aligned}$$

In this case, equation 5.9 (p. 38) tells us to omit the constraint $(i_3 > 3)$. Hence $\mathsf{BPC}(4,10)$ does not contain any hint at any influence being impossible any more.

Chapter 7

Model checking

So far, we have presented how to generate temporal and Boolean path conditions. For the latter we consider all variables to be existentially quantified and there exist constraint solvers that can help to simplify these constraints or even solve for input variables. Model checking can do a similar job for temporal path conditions. We use it to decide if there are state sequences in $\mathcal{M}_p$ that satisfy the path condition and – if so – to find such a state sequence. Moreover, we show how model checking can be used to "solve" an LTL formula for input variables, i. e. how to compute the set of initial states of a program from which satisfying computations start.

There are two approaches to model checking that are fundamentally different: Explicit state model checking represents programs as finite transition systems such as Kripke structures or Büchi automata which have a separate state for every possible program state. Verification is based on searching in the product transition graph of the program and the LTL formula automaton for a strongly connected subgraph that is reachable from an initial state. Symbolic model checking uses Boolean formulae over propositional variables, which are used to encode the program states, to represent sets of states and transition relations of transition systems. μ calculus formulae, to which we convert our LTL formulae, can then be recursively evaluated by applying a number of operators to the Boolean formulae.

One major drawback of LTL model checking is that it is known to be PSPACE-complete [Sis83]. However, this holds only in the length of the formula to be checked. There are algorithms (see below) that are linear in the number of program states.

In this chapter, we first present how to extract Kripke structures from transition graphs. Then, we briefly present explicit model checking and how to apply it for LTL formulae. Third, we give a short introduction to LTL symbolic model checking and explain how this can be also used to solve for input variables. We conclude with a tiny example program to which we apply these techniques.

7.1 The program model

Model checking traditionally operates on Kripke structures or Büchi automata which are finite transition systems. In order to be able to apply model checking to

IMP programs, we must convert them into an appropriate finite transition system. Similarly to section 6.1.2 where we have shown how to obtain state sequences from transition graphs we now present how to construct Kripke structures from transition graphs and how to obtain a Büchi automaton from a Kripke structure.

7.1.1 Kripke structures

Definition 44 (Kripke structure)
Let $W \subseteq \mathcal{W}$ be a finite set of propositional variables. A **Kripke structure** over W is a 4-tuple $\boldsymbol{\mathcal{M}} = (Q, \Upsilon, \Delta, L)$ where

- Q is a finite set of **states**,
- $\Upsilon \subseteq Q$ is the set of **initial states**,
- $\Delta \subseteq Q \times Q$ is the **transition relation**, and
- $L : Q \mapsto \mathfrak{P}(W)$ is a function that labels every state with the set of propositional variables from W that are true in this state.

The **transition graph** of a Kripke structure is the labelled graph $G_{\boldsymbol{\mathcal{M}}} := (Q, \Delta, L)$ with labelling function L.

A **computation** $C = (c_i)_{i \in \mathbb{N}}$ in a Kripke structure $\boldsymbol{\mathcal{M}} = (Q, \Upsilon, \Delta, L)$ over W is an infinite path in its transition graph that starts in an initial state. If $\theta \in \mathsf{LTL}_W$ is an LTL formula over propositional variables from W, we say that C satisfies θ iff the state sequence $\Xi = (\xi_i)_{i \in \mathbb{N}} \vDash \theta$ where $\xi_i(\beth) = \mathtt{T}$ iff $\beth \in L(c_i)$ for all $i \in \mathbb{N}$ and $\beth \in W$.

We now present how to convert an IMP program p to its Kripke structure. Let $TG_p = (N, E, \dot{\to}, \dot{\not\to}, \lambda)$ be the transition graph for p. Let θ be the LTL formula over $\mathcal{L}_{\mathsf{IMP}}$ to be checked. Let $V := \mathbb{V}_p \cup \mathcal{V}(\theta) \cup V_p$ denote the set of variables of interest where V_p is the set of program variable names (cf. equation 6.1 (p. 45)). We associate every atomic formula in θ with a propositional variable, i. e. we have that $\mathfrak{A}(\theta) \subseteq \mathcal{W} - \mathcal{W}(\theta)$.

Let $X := V_p \cup (\mathcal{V}(\theta) - \mathbb{V}_p)$ denote the set of all input variables and non-program variables in θ. The Kripke structure $\boldsymbol{\mathcal{M}}_p = (Q, \Upsilon, \Delta, L)$ over A for p is defined as

- $Q := N \times \mathcal{S}_V^{\mathbf{A}_{\mathsf{IMP}}}$,
- $\Upsilon := \{ (v_e, s|_V) \mid s \in \mathcal{IS}_p \}$ where v_e is the entry node in CFG_p,
- $((v, \xi), (w, \xi')) \in \Delta$ iff
  - $(v, \pi, w) \in E$ is a transition edge such that $\xi(\lambda((v, w))) = \mathtt{T}$, $\xi'|_{\mathbb{V}_p \cup V_p}$ is obtained from $\xi|_{\mathbb{V}_p \cup V_p}$ like ξ_{i+1} is obtained from ξ_i in equation 6.2 (p. 46), and $\xi(x) = \xi'(x)$ for all $x \in \mathcal{V}(\theta) \cap \mathbb{V}^r$, or
  - $(v, \pi, w) \in E$ is an idle edge and $\xi = \xi'$, and
- $L((v, \xi)) := \{ a \in \mathfrak{A}(\theta) \mid \xi(a) = \mathtt{T} \}$.

The problem with this construction is that $|Q|$ is exponential in the size of V, which contains $\mathbb{V}_p$. Since we distinguish every variable occurrence, $|\mathbb{V}_p|$ is linear in the size of p. However, many states in Q are not reachable from states in Υ, e. g. all those states that violate Φ constraints from Φ functions (cf. section 5.4). Since we are only interested in paths starting in Υ, we can safely remove from Q all states that are not reachable from Υ via Δ.

7.1.2 Büchi automata

Büchi automata [Büc60] are finite ω-automata with the acceptance condition proposed by Büchi: Every accepting run must infinitely often pass through an accepting state.

Definition 45 (Finite ω-automaton, Büchi automaton)
A **(finite) ω-automaton** (**Büchi automaton**) is represented by a 5-tuple $\boldsymbol{\mathcal{A}} = (Q, \Sigma, \Delta, \Upsilon, F)$ where

- Q is a finite set of **states**,
- Σ is a finite **alphabet**,
- $\Delta \subseteq (Q \times \Sigma) \times Q$ is the **transition relation**,
- $\Upsilon \subseteq Q$ is the nonempty set of **initial states**, and
- $F \subseteq Q$ is the set of **accepting states**.

The **transition graph** for a Büchi automaton $\boldsymbol{\mathcal{A}} = (Q, \Sigma, \Delta, \Upsilon, F)$ is the multigraph $G_{\boldsymbol{\mathcal{A}}} := (Q, \Delta, \odot\!\!\rightarrow, \rightarrow\!\!\odot)$ where $\odot\!\!\rightarrow((q, \varsigma, q')) := q$ and $\rightarrow\!\!\odot((q, \varsigma, q')) := q'$ for $(q, \varsigma, q') \in \Delta$.

An **ω-run** R of a finite ω-automaton $\boldsymbol{\mathcal{A}} = (Q, \Sigma, \Delta, \Upsilon, F)$ over an infinite word $a = (a_i)_{i \in \mathbb{N}} \in \Sigma^\omega$ is an infinite sequence of states $R = (q_i)_{i \in \mathbb{N}}$ where $q_0 \in \Upsilon$ and $(q_i, a_i, q_{i+1}) \in \Delta$ for all $i \in \mathbb{N}$.

An ω-run $R = (q_i)_{i \in \mathbb{N}}$ over $a = (a_i)_{i \in \mathbb{N}}$ of the Büchi automaton $\boldsymbol{\mathcal{A}}$ is **accepted** by $\boldsymbol{\mathcal{A}}$ iff there is a state $q \in F$ such that the set $\{\, i \in \mathbb{N} \mid q_i = q \,\}$ is infinite, i. e. the run passes infinitely often through an accepting state. An infinite word $a \in \Sigma^\omega$ is accepted by $\boldsymbol{\mathcal{A}}$ if there is an accepting run of $\boldsymbol{\mathcal{A}}$ over a.

The **language** $\mathfrak{L}(\boldsymbol{\mathcal{A}})$ of $\boldsymbol{\mathcal{A}}$ is the set of all infinite words accepted by $\boldsymbol{\mathcal{A}}$.

The Büchi automaton for an IMP program p is most easily defined in terms of the Kripke structure $\boldsymbol{\mathcal{M}}_p = (Q, \Upsilon, \Delta_M, L)$ over $W \subseteq \mathcal{W}$ for p. The Büchi automaton $\boldsymbol{\mathcal{A}}_p$ for p is defined as

$$\boldsymbol{\mathcal{A}}_p := (Q \cup \{\, \iota \,\}, \mathfrak{P}(W), \Delta, \{\, \iota \,\}, Q \cup \{\, \iota \,\})$$

where $(q, \varsigma, q') \in \Delta$ iff

- $q' \in Q$ and
  - $(q, q') \in \Delta_M$ or
  - $q = \iota$ and $q' \in \Upsilon$, and
- $\varsigma = L(q')$.

Notice that all states of $\boldsymbol{\mathcal{A}}_p$ are accepting. Hence, every run in $\boldsymbol{\mathcal{A}}_p$ is accepting.

7.2 Explicit state model checking

Explicit state LTL model checking procedures view programs as Kripke structures or Büchi automata. Both approaches generate a finite "product graph" that combines the program structure with the LTL formula which has an infinite path subject to some constraints iff the LTL formula is satisfiable. More precisely, we obtain a satisfying state sequence for the formula from the infinite path. Hence, by searching the product graph for a cycle that is reachable from some initial state set and that meets the constraints, we are able to decide whether the formula is satisfiable or not (and if so, we get one satisfying sequence).

There exist two variants for doing this. We first present very briefly the one that operates on Kripke structures before outlining the ideas behind model checking with Büchi automata.

7.2.1 Kripke structures

The "product graph" in this approach is a so-called tableau which is constructed from the Kripke structure of a program and the LTL formula. It has been first proposed by Lichtenstein and Pnueli [LP85]. Although their algorithm exhibits very clearly the ideas behind LTL model checking, we do not go into detail here, because this algorithm is of no practical relevance. For more details, see [LP85] and [CGP00].

First, we compute the closure of the LTL formula θ, a set of LTL formulae which contains subformulae of θ and subformulae of θ with extra $\bigcirc$ operators, their negations, and the Boolean constants true and false. Next, for every state of the Kripke structure, we construct all atoms: An atom is a state s and a subset F of the closure of θ such that F is maximally consistent and consistent with the labelling of s. These atoms are the nodes of the atoms' graph that are joined by an edge whenever the two states are linked by a transition in the original Kripke structure and for every LTL formula of the form $\bigcirc(\eta)$ in the source state's formula set, then η must be in the one of target node.

The key insight is that all infinite paths in the atoms' graph that end up in a strongly connected subgraph thereof with no outgoing edges (terminal component) such that for every LTL formula $\eta \,\mathcal{U}\, \eta'$ in one of its atoms, there also exists in it an atom with η' in it satisfy all formulae in the first atom's formula set. We call such a component self-fulfilling. A simple algorithm for model checking a Kripke structure and LTL formula θ is then:

1. Compute the atoms graph and decompose it into strongly connected components.

2. Iteratively remove all terminal components until only self-fulfilling ones remain.

3. Check whether there is an atom (s, F) such that s is an initial state and $\theta \in F$.

7.2.2 Büchi automata

The main reason for the Kripke structure approach not being applied in practice is that we have to construct the complete atoms' graph before we can start analyzing it. When looking for satisfying assignments, we are not able to incorporate additional knowledge from the LTL formula in the generation process. By using Büchi automata, we often avoid constructing the whole Kripke structure and atoms' graph. Here, we briefly present how to use Büchi automata for model checking. For more details, see [CGP00, Hol03].

The idea is to construct a Büchi automaton that is the intersection automaton for the program's Büchi automaton and a model automaton for the LTL formula θ over propositional variables from W and that accepts exactly those sequences $C = (c_i)_{i\in\mathbb{N}}$ of subsets of W that - when viewed as state sequences via the identification $\xi_i(\beth) = \mathtt{T}$ iff $\beth \in c_i$ - are models for θ.

Again, we first have to compute the closure $\mathrm{Cl}(\theta)$ of θ which, this time, is the set of all subformulae of θ and their negations. The local automaton for θ, whose states, each of which is accepting, are all maximally consistent subsets of $\mathrm{Cl}(\theta)$ and whose alphabet is $\mathfrak{P}(\mathrm{Cl}(\theta))$, ensures that every transition is consistent in itself, i. e. the input letter must be the source state q, for every $\bigcirc(\eta) \in \mathrm{Cl}(\theta)$ we require $\bigcirc(\eta) \in q$ iff $\eta \in q'$, and for every $\eta \,\mathcal{U}\, \eta' \in \mathrm{Cl}(\theta)$ we demand that $\eta \,\mathcal{U}\, \eta'$ holds in q iff $\eta' \in q$, or $\eta \in q$ and $\eta \,\mathcal{U}\, \eta' \in q'$. All states that contain θ are initial states.

The eventuality automaton for θ ensures that all eventualities, i. e. all formulae of the form $\eta \,\mathcal{U}\, \eta' \in \mathrm{Cl}(\theta)$, are eventually satisfied. Its states are all eventuality subsets, its alphabet is again the power set of the closure of θ, and $\emptyset$ is the only initial and the only accepting state. In the state $\emptyset$ transitions are possible to all states that contain all eventualities $\eta \,\mathcal{U}\, \eta'$ of the input letter ς that are not immediately satisfied, i. e. $\eta' \notin \varsigma$. In other states, all target states must contain all eventualities of the origin state that are not satisfied by the input letter.

The model automaton is the intersection automaton for local and eventuality automaton where the input alphabet is restricted to W and every transition input letter is projected to W.

To search for models for θ in the program, we compute the intersection automaton for the program Büchi automaton and the model automaton for θ and use depth-first search to see whether there exists an infinite path that starts in an initial state, i. e. whether we can reach a cycle from an initial state. In this case, the language intersection of program and model automaton is nonempty, i. e. there is a satisfying sequence. By projecting the nodes on the infinite path to the state component we obtain a model for θ.

Note that the construction for the model automaton above is highly inefficient. By construction, the automaton's size is exponential in the length of the formula. Gerth, Peled, Vardi, and Wolper [GPVW95] propose a different construction method. They start with an automaton with only two states and then repeatedly expand nodes until a correct model automaton has been generated. Although this algorithm generates in the worst case automata whose size is exponential in the length of the LTL formula, it has a much better average case

performance. For details on this construction method, see [GPVW95] or [CGP00].

The main advantage of Büchi automata is that we can avoid to fully construct the Kripke structure in many cases. As we have seen, we can easily construct a Kripke structure and a Büchi automaton from the transition graph of a program. If we remember for each state the program variable assignments in it, we can use the transition graph to determine which successor nodes exist for it. Hence, if we use a (nested) depth-first search algorithm to find a run of the intersection automaton of the program and model automaton, we can find the successors of a state by computing "on the fly" the successor states in both automata and then find those that can be combined to form a new state in the intersection automaton.

This way, we may be lucky to find a cycle before all states have been explored. Moreover, some states of the program Büchi automaton may not have to be generated at all because they are only reachable by runs that violate the LTL formula. Further, the search can try to limit the number of randomly accessed states and memory cells to avoid unnecessary paging as, in most cases, memory requirements will exceed the available main memory. See e. g. [CVWY92] for algorithms that are designed to keep the randomly accessed memory small.

This model checking approach has been successfully implemented e. g. in SPIN [Hol03]. Besides the optimizations alluded to in this section, it makes use of a number of other optimizations that we will not discuss here because this is not the central topic of this thesis. SPIN is designed to verify concurrent programs, i. e. its focus lies on processes that involve communication and synchronization, but not on data intensive programs. In particular, SPIN assumes that the program can only start in a single initial state, i. e. there is only one successor state to the initial state ι in the Büchi automaton $\boldsymbol{\mathcal{A}}_p$. If we want to use SPIN to check the satisfiability of an LTL formula with respect to program p, we must use the nondeterministic features in ProMeLa to allow for multiple initial states. Conversely, if we assume that all variables are initialized in the program before they are used, i. e. p does not depend on input variables, we could have avoided all the work to generate the LTL formula by simply running or simulating p once to see whether an influence path is executed.

7.3 Symbolic model checking

In the last section we have seen how to apply traditional model checking methods to finding influence paths between two statements in a program p by constructing a finite state machine for p and an temporal influence condition whose satisfiability with respect to p is then checked. However, we always find only one satisfying state sequence, we do not get any information on all satisfying state sequences. Moreover, these finite state machines become very large very quickly, e. g. a single additional integer variable that is not bounded can increase the state space by the factor 2^{32}. Since they are generated from a much more compact description p, we might expect that they contain many similar substructures. The idea of symbolic model checking [McM92] is that we do no longer construct each state on its own, but to describe sets of states and transitions between states efficiently. Usually,

one uses Boolean formulae to encode the transitions and Boolean formulae with fixpoint operators for the temporal formulae. Ordered binary decision diagrams (OBDD) [Bry86] have been shown to be the first choice in order to keep these formulae's representations as small as possible.

In this section, we briefly introduce the propositional μ calculus to which we convert LTL formulae. We then show how to evaluate such a μ calculus formula over a program. Due to space limitations, we do not present how to implement evaluating a μ calculus formula using Boolean formulae and OBDDs. We refer the interested reader to [McM92] and [CGP00].

7.3.1 Propositional μ calculus

Propositional μ calculus has become popular in the context of model checking because a number of formal specification methods can be translated into μ calculus (e. g. CTL, LTL, observational equivalence [BCM$^+$90]) and efficient model checkers are available for it, e. g. SMV [McM92] and NuSMV [CCGR99]. Here, we present a variant of the propositional μ calculus dialect proposed by Kozen [Koz83]. Instead of using transition systems, which may contain a number of transition relations, we define μ calculus with respect to Kripke structures, which allow only one transition relation, because this is sufficient for LTL formulae being encoded in the μ calculus. This section follows the lines of [CGP00].

Apart from propositional variables, the μ calculus knows relational variables. Let $\mathcal{X}$ denote a set of relational variables. We usually use letters set in Gothic print (e. g. $\mathfrak{Q}, \mathfrak{R}$) to denote relational variables. Every relational variable $\mathfrak{R} \in \mathcal{X}$ stands for a set of states, i. e. $\mathfrak{R} \subseteq Q$.

Definition 46 (μ calculus formulae)
Let $\boldsymbol{\mathcal{M}} = (Q, \Upsilon, \Delta, L)$ be a Kripke structure over $W \subseteq \mathcal{W}$. The set of μ-calculus formulae $\mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ over $\boldsymbol{\mathcal{M}}$ is the smallest set of words over the alphabet $\mathcal{Z}_{\mathsf{MU}} := \mathcal{X} \cup \mathcal{O}_{\mathsf{MU}} \cup \{ (,), [,], ., \langle, \rangle \}$ with the operator set $\mathcal{O}_{\mathsf{MU}} := \{ \neg, \wedge, \vee, \mu, \nu \}$ that satisfies

- Every propositional variable in W is a μ-calculus formula: $W \subseteq \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$.
- Every relational variable is a μ-calculus formula: $\mathcal{X} \subseteq \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$.
- If $f, g \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ are μ-calculus formula, so are $\neg(f)$, $(f) \wedge (g)$, $(f) \vee (g) \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$.
- If $f \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ is a μ-calculus formula, then are $\langle\rangle(f), [](f) \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ μ-calculus formulae.
- If $\mathfrak{R} \in \mathcal{X}$ is a relational variable and $f \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ a μ calculus formula, then are $\mu\mathfrak{R}.f, \nu\mathfrak{R}.f \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ μ-calculus formulae if $\mathfrak{R}$ occurs only positively in f, i. e. every occurrence of $\mathfrak{R}$ in f falls under an even number of negations.[30]

[30] Positive and negative occurrence can be defined analogously to definition 18 (p. 22).

As before, we introduce precedence rules to save parentheses. We order the operators as

$$\mu \quad \nu \quad \neg \quad []\cdot \quad \langle\rangle\cdot \quad \wedge \quad \vee,$$

i. e. μ has highest priority and $\vee$ least. For example, we write $\mu\mathfrak{R}.\mathfrak{R}\wedge\beth\vee\nu\mathfrak{S}.\neg(\neg\mathfrak{S})$ for $((\mu\mathfrak{R}.(\mathfrak{R}))\wedge(\beth))\vee(\nu\mathfrak{S}.(\neg(\neg\mathfrak{S})))$.

Definition 47 (Free and bound relational variables)
Let $f \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ be a μ-calculus formula and $\mathfrak{R} \in \mathcal{X}$ a relational variable. $\mathfrak{R}$ occurs **freely** in f if $\mathfrak{R}$ occurs in f such that there is no formula $g \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ containing $\mathfrak{R}$ such that $\mu\mathfrak{R}.(g)$ or $\nu\mathfrak{R}.(g)$ is a subformula of f that contains the occurrence of $\mathfrak{R}$. $\mathfrak{R}$ occurs **bound** in f if there is a subformula $\mu\mathfrak{R}.(g)$ or $\nu\mathfrak{R}.(g)$ of f such that $\mathfrak{R}$ occurs in $g \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$.

f is said to be **closed** iff no relational variable occurs freely in f.

Definition 48 (Environment)
An **environment** for a Kripke structure $\boldsymbol{\mathcal{M}} = (Q, \Upsilon, \Delta, L)$ is a mapping $\omega : \mathcal{X} \mapsto \mathfrak{P}(Q)$. If $\omega \in \mathfrak{P}(Q)^{\mathcal{X}}$ is an environment, $\mathfrak{R} \in \mathcal{X}$ a relational variable, and $P \subseteq Q$ a set of states, then $\omega[\mathfrak{R} \leftarrow P]$ denotes the environment defined by

$$\omega[\mathfrak{R} \leftarrow P] := \mathfrak{S} \mapsto \begin{cases} P & \text{if } \mathfrak{S} = \mathfrak{R} \\ \omega(\mathfrak{S}) & \text{otherwise} \end{cases}.$$

Definition 49 (Semantics of μ-calculus formulae)
Let $\boldsymbol{\mathcal{M}} = (Q, \Upsilon, \Delta, L)$ be a Kripke structure over W. A μ-calculus formula $f \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ is interpreted as a set of states $[\![f]\!]^{\omega}_{\boldsymbol{\mathcal{M}}}$ in which f holds where ω is an environment. This set is defined inductively:

- $[\![\beth]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} := \{\, q \in Q \mid \beth \in L(q) \,\}$ for every propositional variable $\beth \in W$.
- $[\![\mathfrak{R}]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} := \omega(\mathfrak{R})$ for every relational variable $\mathfrak{R} \in \mathcal{X}$.
- $[\![\neg(f)]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} := Q - [\![f]\!]^{\omega}_{\boldsymbol{\mathcal{M}}}$ where $f \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$
- $[\![(f)\wedge(g)]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} := [\![f]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} \cap [\![g]\!]^{\omega}_{\boldsymbol{\mathcal{M}}}$ where $f, g \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$.
- $[\![(f)\vee(g)]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} := [\![f]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} \cup [\![g]\!]^{\omega}_{\boldsymbol{\mathcal{M}}}$ where $f, g \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$.
- $[\![\langle\rangle(f)]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} := \{\, q \in Q \mid \Delta(q) \cap [\![f]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} \neq \emptyset \,\}$ where $f \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$.
- $[\![[](f)]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} := \{\, q \in Q \mid \Delta(q) \subseteq [\![f]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} \,\}$ where $f \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$.
- $[\![\mu\mathfrak{R}.(f)]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} := \mu\mathfrak{R}.\tau_f(\mathfrak{R})$ is the least fixpoint of $\tau_f : \mathfrak{P}(Q) \mapsto \mathfrak{P}(Q)$, $\tau_f(P) := [\![f]\!]^{\omega[\mathfrak{R}\leftarrow P]}_{\boldsymbol{\mathcal{M}}}$ where $f \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ and $\mathfrak{R} \in \mathcal{X}$.[31]
- $[\![\nu\mathfrak{R}.(f)]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} := \nu\mathfrak{R}.\tau_f(\mathfrak{R})$ is the greatest fixpoint of $\tau_f : \mathfrak{P}(Q) \mapsto \mathfrak{P}(Q)$, $\tau_f(P) := [\![f]\!]^{\omega[\mathfrak{R}\leftarrow P]}_{\boldsymbol{\mathcal{M}}}$ where $f \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ and $\mathfrak{R} \in \mathcal{X}$.

[31] $P \subseteq Q$ is a fixpoint of τ_f iff $\tau_f(P) = P$. P is the least (greatest) fixpoint of τ_f iff for every other fixpoint $R \subseteq Q$ of τ_f we have $P \subseteq R$ ($R \subseteq P$). Since, by definition, $\mathfrak{R}$ occurs only positively in τ_f, τ_f is monotonic on the complete lattice $(\mathfrak{P}(Q), \cap, \cup)$. Hence, τ_f has least and greatest fixpoints [Tar55]. As Q is finite, τ_f is also $\cup$- and $\cap$-monotonic. Thus, we can use fixpoint iteration to compute the greatest and least fixpoint of τ_f in a finite number of steps [CGP00].

Intuitively, $[\![\langle\rangle(f)]\!]^{\omega}_{\boldsymbol{\mathcal{M}}}$ ($[\![[](f)]\!]^{\omega}_{\boldsymbol{\mathcal{M}}}$) is the set of states one (all) of whose successor states with respect to transitions in Δ satisfy f.

7.3.2 Encoding programs as Boolean formulae

We want to use Boolean formula to encode sets of states of a Kripke structure $\boldsymbol{\mathcal{M}} = (Q, \Upsilon, \Delta, L)$ over W. Let $U_Q \subseteq \mathcal{W}$ be a finite set of propositional variables. Let $\Gamma_Q : Q \mapsto \mathbb{B}^{U_Q}$ be an injective mapping that identifies every state $q \in Q$ with an assignment on U_Q such that $\Gamma_Q(q)(\beth) = \mathtt{T}$ iff $\beth \in L(q)$ for all $\beth \in U_Q \cap W$ and $q \in Q$. For every Boolean formula θ over U_Q we define the set of states modelled by θ as

$$[\![\theta]\!]_{\boldsymbol{\mathcal{M}}} := \{\, q \in Q \mid \Gamma_Q(q)(\theta) = \text{true} \,\}.$$

Conversely, for every set of states $P \subseteq Q$, we can find a Boolean formula θ_P over U_Q such that $[\![\theta]\!]_{\boldsymbol{\mathcal{M}}} = P$. Since the evaluation $[\![f]\!]^{\omega}_{\boldsymbol{\mathcal{M}}}$ of any μ-calculus formula f in a Kripke structure $\boldsymbol{\mathcal{M}}$ and an environment ω is a set of states, we can encode $[\![f]\!]^{\omega}_{\boldsymbol{\mathcal{M}}}$ as a Boolean formula over U_Q.

In order to be able to evaluate μ-calculus formulae by manipulating Boolean formula, we also have to encode the transition relation Δ of $\boldsymbol{\mathcal{M}}$ as a Boolean formula. Let $U'_Q \subseteq \mathcal{W}$ be a another set of propositional variables such that $|U_Q| = \left|U'_Q\right|$, $U'_Q \cap U_Q = \emptyset$. Let $\cdot' : U_Q \mapsto U'_Q$ be a bijection that assigns each $\beth \in U_Q$ its primed variant $\beth' \in U'_Q$. Then, we encode Δ as a Boolean formula $\theta(U_Q, U'_Q)$ over $U_Q \cup U'_Q$ such that the assignment ξ that is the combination of $\Gamma_Q(q') \circ \cdot'^{-1}$ and $\Gamma_Q(q)$ satisfies $\theta(U_Q, U'_Q)$ iff $(q, q') \in \Delta$, for all $q, q' \in Q$. For example, we can set

$$\theta(U_Q, U'_Q) := \bigvee_{(q,q') \in \Delta} \theta_{\{\, q \,\}} \wedge (\theta_{\{\, q' \,\}}[\beth'/\beth, \beth \in U_Q]) \tag{7.1}$$

where $\rho[\beth'/\beth, \beth \in U_Q]$ is the Boolean formula we obtain from ρ by substituting $\beth'$ for $\beth$ for all $\beth \in U_Q$. Note however that usually there are shorter formulae $\theta(U_Q, U'_Q)$ to encode Δ that exploit regularities in Δ.

If we want to encode the transition relation from a program, we can, of course, first convert the program to a Kripke structure and then use equation 7.1 to get a Boolean formula. However, we may get a simpler encoding if we do not make the detour over Kripke structures.

Let $\Gamma_Q : Q \mapsto U_Q$ be a reasonable encoding of Q. For example we might encode every variable b of type `bool` with two propositional variables $\beth_b$ and $\beth'_b$ where the three possible values $\perp_b$, false, and true are encoded as $(\mathtt{F}, \mathtt{F})$, $(\mathtt{F}, \mathtt{T})$, and $(\mathtt{T}, \mathtt{T})$ respectively. Similarly, we can represent every `int` variable in their 32-bit complement representation with 32 propositional variables plus one that stores whether it has been initialized or not. The same way, we can encode array variables and the current state $v \in N$ in the transition graph.

Since all transitions in Δ originate from edges in the transition graph $TG_p = (V', E, \multimap, \multimap\!\!\circ, \lambda)$ of p we can encode Δ by $\bigvee_{e \in E} \theta_e$ where θ_e is a conjunction of the following constraints:

- One that checks that the propositional variables for the transition graph state are correct (over U_Q).

- One that checks that the guard for the edge is satisfied (over U_Q).
- One that encodes the state transformation very much like a Boolean circuit would do (over $U_Q \cup U'_Q$).
- One that ensures that the propositional variables for the transition graph successor state are correct (over U'_Q).

7.3.3 Encoding LTL formulae in the μ calculus

In [CGL94], it is shown how LTL model checking is reduced to CTL model checking with fairness constraints. Similarly to Kripke structures and Büchi automata, the LTL formula is translated into a Kripke structure called tableau and – from an abstract point of view – a path in the model's Kripke structure is searched that satisfies the constraints imposed by the tableau, which works similarly to the local Büchi automaton. [CGP00] contains a modification of this construction which they claim to it producing smaller tableaus. Here, we sketch an adaption of [CGL94] that translates LTL formulae directly into μ-calculus formulae.

Let $\theta \in \mathsf{LTL}$ be an LTL formula over propositional variables $W \subseteq \mathcal{W}$ (W finite). Without loss of generality, we assume that θ does not contain $\Box$ or $\Diamond$ operators. The set of elementary formulae $\mathrm{el}(\theta)$ of θ contains all propositional variables in θ, all subformulae of θ that start with a $\bigcirc$ operator, and for every subformula of θ of the form $(\eta)\ \mathcal{U}\ (\eta')$ it contains the formula $\bigcirc((\eta)\ \mathcal{U}\ (\eta'))$. The tableau's state set is the power set of all elementary formulae of θ, i. e. $Q_\theta := \mathfrak{P}(\mathrm{el}(\theta))$. For the labelling function $L_\theta : Q_\theta \mapsto \mathfrak{P}(W)$, we set $L_\theta(q) := q \cap W$.

For every subformula η and elementary formula η of θ, let $\mathrm{sat}(\eta)$ denote the set of states in Q_θ that satisfy θ: We set $\mathrm{sat}(\mathrm{true}) := Q_\theta$, $\mathrm{sat}(\mathrm{false}) := \emptyset$, $\mathrm{sat}(\eta) := \{\, q \in Q_\theta \mid \eta \in q \,\}$ for $\eta \in \mathrm{el}(\theta)$, and

$$\mathrm{sat}\left((\eta)\ \mathcal{U}\ (\eta')\right) := \mathrm{sat}\left(\eta'\right) \cup \left(\mathrm{sat}\left(\eta\right) \cap \mathrm{sat}\left(\bigcirc((\eta)\ \mathcal{U}\ (\eta'))\right)\right).$$

All other operators are defined as one would expect it to be.

For the transition relation Δ_θ we want to have that every elementary formula $\eta \in q$ in state $q \in Q_\theta$ is true. In particular, we want to have that $\bigcirc(\eta) \in q$ iff all successor states satisfy η and that $\bigcirc(\eta) \notin q$ iff none of the successor states satisfies η. Hence, for the transition relation Δ_θ we set

$$(q, q') \in \Delta_\theta \text{ iff } (\forall \bigcirc(\eta) \in \mathrm{el}(\theta) : q \in \mathrm{sat}(\bigcirc(\eta)) \text{ iff } q' \in \mathrm{sat}(\eta))$$

The set of initial states is $\Upsilon_\theta := \mathrm{sat}(\theta)$. Then, the tableau for θ is $\boldsymbol{\mathcal{M}}_\theta = (Q_\theta, \Upsilon_\theta, \Delta_\theta, L_\theta)$. Notice that we can represent the tableau $\boldsymbol{\mathcal{M}}_\theta$ as a Boolean formula, too.

We now define the product Kripke structure $\boldsymbol{\mathcal{M}} = (Q, \Upsilon, \Delta, L)$ over W of the tableau $\boldsymbol{\mathcal{M}}_\theta = (Q_\theta, \Upsilon_\theta, \Delta_\theta, L_\theta)$ and the program Kripke structure $\boldsymbol{\mathcal{M}}_p = (Q_p, \Upsilon_p, \Delta_p, L_p)$. We set

- $Q := \{\, (q, q') \in Q_\theta \times Q_p \mid L_p(q') \cap W = L_\theta(q) \cap W \,\}$

- $\Upsilon := \Upsilon_\theta \times \Upsilon_p \cap Q$
- $((q, q'), (r, r')) \in \Delta$ iff $(q, r) \in \Delta_\theta$ and $(q', r') \in \Delta_p$ and $(q, q'), (r, r') \in Q$.
- $L((q, q')) := L_\theta(q)$.

All computations of $\boldsymbol{\mathcal{M}}$ correspond to computations in $\boldsymbol{\mathcal{M}}_p$, i. e. to an execution of p and locally respect the constraints imposed by θ. The μ calculus formula we now build checks that all eventuality constraints of θ are satisfied. We want to check if for every subformula $(\eta)\ \mathcal{U}\ (\eta')$ of θ every computation of $\boldsymbol{\mathcal{M}}$ passes infinitely often through a state (q, q') such that $q \in \mathrm{sat}\,(\neg((\eta)\ \mathcal{U}\ (\eta')) \vee \eta')$. If so, it can not be the case that $(\eta)\ \mathcal{U}\ (\eta')$ holds almost always while η' is not satisfied.

Let Θ denote the set of all subformulae of θ of the form $(\eta)\ \mathcal{U}\ (\eta')$. For all $\zeta \in \Theta$, say $\zeta = (\eta)\ \mathcal{U}\ (\eta')$, let ρ_ζ denote the Boolean formulae that encodes the set of states $\{\ (q, q') \in Q \mid q \in \mathrm{sat}\,(\neg((\eta)\ \mathcal{U}\ (\eta')) \vee \eta')\ \}$. The μ-calculus formula for these constraints is then given by

$$\nu\mathfrak{R}.\ \bigwedge_{\zeta\in\Theta} (\langle\rangle\,(\mu\mathfrak{S}.\,(\mathfrak{R} \wedge \rho_\zeta \vee \langle\rangle\,(\mathfrak{S})))) \tag{7.2}$$

Thus, to see whether an LTL formula θ is satisfiable over a program p, we build the Kripke structure $\boldsymbol{\mathcal{M}}$ for p and θ and then evaluate the closed μ-calculus formula from equation 7.2 over $\boldsymbol{\mathcal{M}}$. If an initial state is in the result, there exists a computation that starts in the corresponding state in $\boldsymbol{\mathcal{M}}_p$ that satisfies θ.[32] See e. g. [CGMZ95] on how to generate state sequences with symbolic model checking.

7.4 Model checking and temporal path conditions

So far, we have sketched how explicit and symbolic model checking works on Kripke structures. Hence, when applying model checking, either explicit state or symbolic, to temporal path conditions, we must do the following steps:

1. Compute the Kripke structure or Büchi automaton for program p,
2. Convert the LTL path condition into a tableau or automaton,
3. Construct the product structure or automaton for the program and the formula,
4. Find an appropriate infinite path therein.

In this section, we look at a trivial example program p, shown in figure 7.1 (p. 101), where all variables are of type `bool` and see how model checking can help us in checking the satisfiability of and solving an LTL path condition.

[32] Instead of checking whether the result contains an initial state, we can conjunctively add to equation 7.2 the Boolean formula that encodes the set Υ, too. All states in the result set are then starting state for computations satisfying θ.

Clearly, p does not yield very interesting path conditions, but, for illustration purpose, we look at

$$\Diamond\,\overline{\mathsf{IC}(1,3)} \Longleftrightarrow \Diamond(((b_1 \&\& b_2)||(!(b_1)\&\&!(b_2))) \wedge b_2) \Longleftrightarrow \mathrm{true}\,\mathcal{U}\,(b_1 \wedge b_2) =: \theta.$$

First, we construct the Kripke structure $\boldsymbol{\mathcal{M}}_p$ for p. For the variables, we have that $\mathbb{V}_p = \{\, b_1, b_2, b_3, c, d \,\}$ and $V_p = \{\, \mathtt{b}, \mathtt{c}, \mathtt{d} \,\}$. Hence Υ is the set of states 1 through 8 listed in table 7.1. The remaining states of $\boldsymbol{\mathcal{M}}_p$ that are reachable from Υ are listed in table 7.1 under numbers 9 through 14.

Let Q denote the states reachable from Υ, i. e. $Q = \{\, 1, \ldots, 14 \,\}$. For the transition relation Δ on Q we get:

$$\Delta = \{\, (i, i+8), (i+4, i+8), (i+10, i+10) \mid i \in \{\, 1, \ldots, 4 \,\} \,\} \cup \{\, (9, 13), (10, 14) \,\}$$

There are two atomic formulae in θ: $\mathfrak{A}(\theta) = \{\, b_1, b_2 \,\}$. For the labelling function L, we get:

$$L(i) = \begin{cases} \{\, b_1 \,\} & \text{if } i \in \{\, 9, 10 \,\} \\ \{\, b_1, b_2 \,\} & \text{if } i \in \{\, 13, 14 \,\} \\ \emptyset & \text{otherwise} \end{cases}$$

The Kripke structure $\boldsymbol{\mathcal{M}}_p$ is shown in figure 7.2. Every state is labelled with its state number from table 7.1 and the set of atomic formulae holding in it. Initial states (1 to 8) are marked with an arrow pointing to them. We now apply both explicit state and symbolic model checking to p and θ.

7.4.1 Model checking using Büchi automata

Since it is Büchi automata that are usually used when one does explicit state model checking, we use this approach in this example. The construction of the Büchi automaton $\boldsymbol{\mathcal{A}}_p$ for $\boldsymbol{\mathcal{M}}_p$ is straight forward, so we will not go into details. To keep notation simple, we give the state number 0 to the initial state ι.

We now construct the local and eventuality automaton. The closure of θ is

$$\mathrm{Cl}\,(\theta) = \{\theta, \neg\theta, \mathrm{true}\,, \neg\,\mathrm{true}\,, b_1 \wedge b_2, \neg(b_1 \wedge b_2), b_1, \neg(b_1), b_2, \neg(b_2)\}$$

The states $Q_L := \{\, q_1, \ldots, q_8 \,\}$ of the local automaton for θ

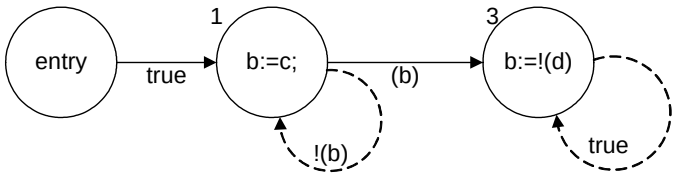
$$\boldsymbol{\mathcal{L}}_\theta = (Q_L, \mathfrak{P}\,(\mathrm{Cl}\,(\theta))\,, \Delta_L, \{\, q_1, q_3, q_5, q_7 \,\}\,, Q_L)$$

are shown in table 7.2 (p. 102). For the transition relation $\Delta_L \subseteq Q_L \times (\mathfrak{P}\,(\mathrm{Cl}\,(\theta)) \times Q_L)$ we get $\Delta_L(q_i) = \{\, q_i \,\} \times Q_L$ for $i \in \{\, 1, 2, 4, 6, 8 \,\}$ and $\Delta_L(q_i) = \{\, q_i \,\} \times \{\, q_1, q_3, q_5, q_7 \,\}$ for $i \in \{\, 3, 5, 7 \,\}$.

Regarding the eventuality automaton, we note that θ contains only one eventuality, namely $\mathrm{true}\,\mathcal{U}\,(b_1 \wedge b_2)$. We now construct the eventuality automaton. The set of eventualities in $\mathrm{Cl}\,(\theta)$ is $e(\theta) = \{\, \mathrm{true}\,\mathcal{U}\,(b_1 \wedge b_2) \,\}$. Figure 7.3 (p. 102) shows the eventuality automaton for θ where A denotes the set $\mathfrak{P}\,(\mathrm{Cl}\,(\theta))$ and B the set $\{\, \Theta \subseteq \mathrm{Cl}\,(\theta) \mid (b_1 \wedge b_2) \in \Theta \,\}$. Accepting states are drawn with a double circle, initial states are marked with an arrow pointing to them. Each edge (q, q')

Figure 7.1 An example program and its transition graph used to illustrate model checking with temporal path conditions.

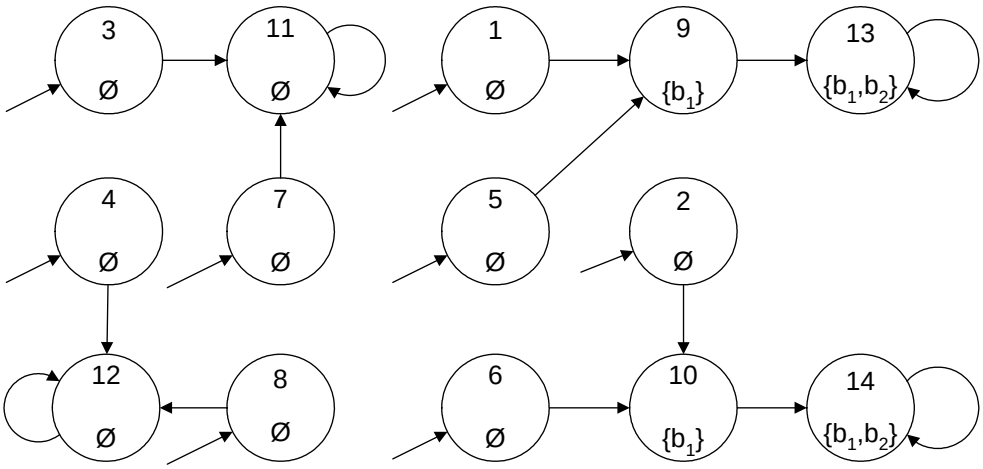
```
1  b := c;
2  if (b) {
3       b := !(d);
4  }
```



State	Node	b	c	d	b_1	b_2	b_3	c	d
1	v_e	T	T	T	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$
2	v_e	T	T	F	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$
3	v_e	T	F	T	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$
4	v_e	T	F	F	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$
5	v_e	F	T	T	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$
6	v_e	F	T	F	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$
7	v_e	F	F	T	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$
8	v_e	F	F	F	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$	$\perp_b$
9	1	T	T	T	T	$\perp_b$	$\perp_b$	T	$\perp_b$
10	1	T	T	F	T	$\perp_b$	$\perp_b$	T	$\perp_b$
11	1	F	F	T	F	$\perp_b$	$\perp_b$	F	$\perp_b$
12	1	F	F	F	F	$\perp_b$	$\perp_b$	F	$\perp_b$
13	3	F	T	T	T	T	F	T	T
14	3	T	T	F	T	T	T	T	F

Table 7.1: Reachable states of the Kripke structure for the program shown in figure 7.1.

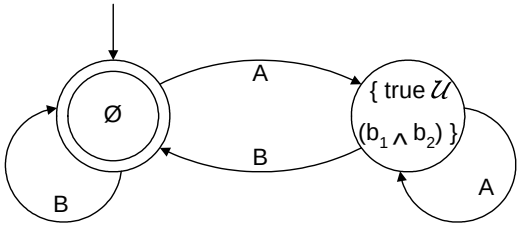
Figure 7.2 Kripke structure transition graph for the program that figure 7.1 shows.



State	true	b_1	b_2	$b_1 \wedge b_2$	true $\mathcal{U}$ $(b_1 \wedge b_2)$
q_1	X	X	X	X	X
q_2	X	X	X	X	
q_3	X	X			X
q_4	X	X			
q_5	X		X		X
q_6	X		X		
q_7	X				X
q_8	X				

Table 7.2: States of the local automaton for true $\mathcal{U}$ $(b_1 \wedge b_2)$. Every row describes one state. If a cell contains X, the formula η in the column header is contained in the state q listed at the row header. If the cell is empty, $\neg(\eta)$ is contained in q.

Figure 7.3 Eventuality automaton for true $\mathcal{U}$ $(b_1 \wedge b_2)$.



with label ς represents a tuple $(q, \varsigma, q') \in \Delta_L$. For simplicity, if there are multiple edges between two states we draw only a single edge with all labels collected in a set of labels.

Formally, the eventuality automaton is

$$\boldsymbol{\mathcal{E}}_\theta = (\{ \emptyset, \{ \theta \} \}, \mathfrak{P}(\mathrm{Cl}(\theta)), \Delta_E, \{ \emptyset \}, \{ \emptyset \})$$

where $\Delta_E := \{ \emptyset \} \times (B \times \{ \emptyset \} \cup A \times \{ \theta \}) \cup \{ \theta \} \times (A \times \{ \theta \} \cup B \times \{ \emptyset \})$.

Let $W = \{ b_1, b_2 \}$ be the set of atomic formulae in θ. We now build the model automaton

$$\boldsymbol{\mathcal{A}}_\theta = (Q_L \times \{ \emptyset, \{ \theta \} \}, \mathfrak{P}(W), \Delta, \{ q_1, q_3, q_5, q_7 \} \times \{ \emptyset \}, Q_L \times \{ \emptyset \}),$$

with $\Delta := \{ (q, P(\varsigma), q') \mid (q, \varsigma, q') \in \Delta^* \}$ where $P : \mathfrak{P}(\mathrm{Cl}(\theta)) \mapsto \mathfrak{P}(W)$, $P(\Theta) := \Theta \cap W$ is the projection map used to reduce the alphabet and Δ^* is the transition relation for the standard product automaton of $\boldsymbol{\mathcal{L}}_\theta$ and $\boldsymbol{\mathcal{E}}_\theta$, i. e. $((q, r), \varsigma, (q', r')) \in \Delta^*$ iff $(q, \varsigma, q') \in \Delta_L$ and $(r, \varsigma, r') \in \Delta_E$.

To decide if there is a sequence in $\boldsymbol{\mathcal{A}}_p$ that satisfies θ, we build the intersection automaton $\boldsymbol{\mathcal{A}}$ of both automata $\boldsymbol{\mathcal{A}}_p$ and $\boldsymbol{\mathcal{A}}_\theta$ and check whether $\boldsymbol{\mathcal{A}}$ has any accepting runs. If we do this, we can find, for instance, the run

$$(0, q_7, \emptyset), (6, q_3, \{ \theta \}), (10, q_1, \{ \theta \}), (14, q_1, \emptyset), (14, q_1, \emptyset), \ldots$$

over the word $\emptyset, \{ b_1 \}, \{ b_1, b_2 \}, \{ b_1, b_2 \}, \ldots$. Hence, we know that when run the program with the assignment to the input variables given by `b` := `F`, `c` := `T`, and `d` := `F`, information flows from `c` in line 1 to line 3.

q_1^θ	q_2^θ	q_3^θ	q_4^θ	q_5^θ	q_6^θ	q_7^θ	q_8^θ	η	$\mathrm{sat}(\eta)$
X	X	X	X					b_1	$\{ q_1^\theta, \ldots, q_4^\theta \}$
X	X			X	X			b_2	$\{ q_1^\theta, q_2^\theta, q_5^\theta, q_6^\theta \}$
X		X		X		X		$\bigcirc(\mathrm{true}\, \mathcal{U}\, (b_1 \wedge b_2))$	$\{ q_1^\theta, q_3^\theta, q_5^\theta, q_7^\theta \}$
								true	$\{ q_1^\theta, \ldots, q_8^\theta \}$
								$b_1 \wedge b_2$	$\{ q_1^\theta, q_2^\theta \}$
								$\mathrm{true}\, \mathcal{U}\, (b_1 \wedge b_2)$	$\{ q_1^\theta, \ldots, q_5^\theta, q_7^\theta \}$

Table 7.3: States q_1^θ to q_8^θ of the tableau for $\mathrm{true}\, \mathcal{U}\, (b_1 \wedge b_2)$ contain all the elementary formulae η marked with a X in their column. The column on the right lists the sat sets for all elementary formulae and subformulae of $\mathrm{true}\, \mathcal{U}\, (b_1 \wedge b_2)$.

7.4.2 Symbolic model checking

Next, we show how to apply symbolic model checking to p and θ, i. e. how to build the product Kripke structure $\boldsymbol{\mathcal{M}}$ and the μ-calculus formula that we afterwards evaluate over $\boldsymbol{\mathcal{M}}$. We do not present how to encode the Kripke structures in Boolean formulae and how to evaluate the μ-calculus formula by manipulating Boolean formulae.

We start by constructing the tableau $\boldsymbol{\mathcal{M}}_\theta = (Q_\theta, \Upsilon_\theta, \Delta_\theta, L_\theta)$ for $\theta = \mathrm{true}\, \mathcal{U}\, (b_1 \wedge b_2)$. The set of elementary formulae of θ is

$$\mathrm{el}\,(\theta) :== \{\, b_1, b_2, \bigcirc(\mathrm{true}\, \mathcal{U}\, (b_1 \wedge b_2)) \,\}\,.$$

Table 7.3 shows on the left the states in Q_θ of which $q_1^\theta, \ldots, q_5^\theta, q_7^\theta$ are initial states. The sat sets for elementary formulae and subformulae of θ are listed on the right-hand side of the same table.

The transition relation Δ_θ is given by

$$\Delta_\theta := \left\{ q_1^\theta, q_3^\theta, q_5^\theta, q_7^\theta \right\} \times \left\{ q_1^\theta, \ldots, q_5^\theta, q_7^\theta \right\} \cup \left\{ q_2^\theta, q_4^\theta, q_6^\theta, q_8^\theta \right\} \times \left\{ q_6^\theta, q_8^\theta \right\}$$

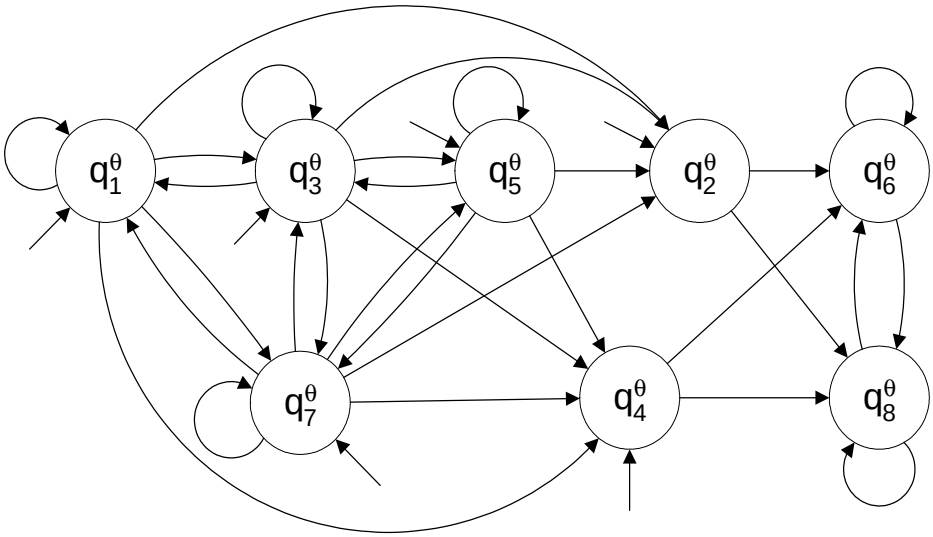
The labels for states q_i^θ can be seen in the first two rows of table 7.3: Iff the cell contains an X, the formula under η is part of the label for the column's state. Figure 7.4 shows the tableau where the labels to the nodes have been omitted.

Next, we construct the product Kripke structure $\boldsymbol{\mathcal{M}}$ of $\boldsymbol{\mathcal{M}}_p$ and $\boldsymbol{\mathcal{M}}_\theta$ which is given by:

- $Q := \{ q_7^\theta, q_8^\theta \} \times \{ 1, \ldots, 8, 11, 12 \} \cup \{ q_3^\theta, q_4^\theta \} \times \{ 9, 10 \} \cup \{ q_1^\theta, q_2^\theta \} \times \{ 13, 14 \}$,
- $\Upsilon := \{ q_7^\theta \} \times \{ 1, \ldots, 8 \}$,
- $L(q, q') := \{ b_1, b_2 \}$ for $q \in \{ q_1^\theta, q_2^\theta \}$, $L(q, q') := \{ b_1 \}$ for $q \in \{ q_3^\theta, q_4^\theta \}$, and $L(q, q') = \emptyset$ otherwise, and
- $\Delta := \{ (q_8^\theta, q_8^\theta) \} \otimes \{ 1, \ldots, 8, 11, 12 \} \cup \{ q_1^\theta, q_2^\theta \}^2 \otimes \{ (13, 13), (14, 14) \} \cup$
  $\{ (q_3^\theta, q_1^\theta), (q_3^\theta, q_2^\theta) \} \otimes \{ (9, 13), (10, 14) \} \cup$
  $\{ (q_7^\theta, q_3^\theta), (q_7^\theta, q_4^\theta) \} \otimes \{ (1, 9), (2, 10), (5, 9), (6, 10) \} \cup$
  $\{ (q_7^\theta, q_7^\theta) \} \otimes \{ (3, 11), (4, 12), (7, 11), (8, 12) \}$.[33]

[33] For $A \subseteq Q_\theta \times Q_\theta$ and $B \subseteq Q_p \times Q_p$, let $A \otimes B := \{ ((a, b), (a', b')) \mid (a, a') \in A, (b, b') \in B \}$.

Figure 7.4 Tableau for true $\mathcal{U}$ $(b_1 \wedge b_2)$ with states from table 7.3 without labels.



Let $\zeta := \neg(\text{true}\,\mathcal{U}\,(b_1 \wedge b_2)) \vee (b_1 \wedge b_2)$. With $\text{sat}\,(\zeta) = \{\, q_1^\theta, q_2^\theta, q_6^\theta, q_8^\theta \,\}$ and ρ_ζ being a Boolean formula that encodes the set $\text{sat}\,(\zeta)$, we obtain

$$f := \nu\mathfrak{R}.\,\langle\rangle(\mu\mathfrak{S}.(\mathfrak{R} \wedge \rho_\zeta \vee \langle\rangle(\mathfrak{S}))) \wedge \theta_\Upsilon$$

as the μ-calculus formula that we have to evaluate over $\boldsymbol{\mathcal{M}}$ where θ_Υ is the Boolean formula for Υ.

We now evaluate f over $\boldsymbol{\mathcal{M}}$. We start with the μ operator in f in an environment ω with $\omega(\mathfrak{R}) = Q$. Let $g := \mathfrak{R} \wedge \rho_\zeta \vee \langle\rangle(\mathfrak{S})$. We now compute the least fixpoint $\mu\mathfrak{S}.g$ of g with fixpoint iteration starting in the environment $\omega_0 := \omega[\mathfrak{S} \leftarrow \emptyset]$. We obtain

$$Q_0 := [\![g]\!]^{\omega_0}_{\boldsymbol{\mathcal{M}}} = \{\, q_1^\theta, q_2^\theta \,\} \times \{\, 13, 14 \,\} \cup \{\, q_8^\theta \,\} \times \{\, 1, \ldots, 8, 11, 12 \,\}$$

Set $\omega_1 := \omega_0[\mathfrak{S} \leftarrow Q_0]$. Then $Q_1 := [\![g]\!]^{\omega_1}_{\boldsymbol{\mathcal{M}}} = Q_0 \cup \{\, q_3^\theta \,\} \times \{\, 9, 10 \,\}$. Set $\omega_2 := \omega_1[\mathfrak{S} \leftarrow Q_1]$. Then, $Q_2 := [\![g]\!]^{\omega_2}_{\boldsymbol{\mathcal{M}}} = Q_1 \cup \{\, q_7^\theta \,\} \times \{\, 1, 2, 5, 6 \,\}$. Since $[\![g]\!]^{\omega_2[\mathfrak{S} \leftarrow Q_2]}_{\boldsymbol{\mathcal{M}}} = Q_2$, the fixpoint iteration terminates and we have $[\![\mu\mathfrak{S}.(g)]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} = Q_2$. Note that also $Q_2 = [\![\langle\rangle(\mu\mathfrak{S}.(g))]\!]^{\omega}_{\boldsymbol{\mathcal{M}}}$. This has been the first fixpoint iteration of $h := \langle\rangle(\mu\mathfrak{S}.(g))$ to compute the greatest fixpoint of h. For the second iteration, we have to rerun the fixpoint iteration for the inner μ operator. Thus, we set $\omega_3 := \omega[\mathfrak{R} \leftarrow Q_2]$ and evaluate $\mu\mathfrak{S}.(g)$ in the environment ω_3 again. This yields again Q_2. Hence, $[\![h]\!]^{\omega_3}_{\boldsymbol{\mathcal{M}}} = Q_2$, so $[\![\nu\mathfrak{R}.(h)]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} = Q_2$.

Thus, $[\![f]\!]^{\omega}_{\boldsymbol{\mathcal{M}}} = \{\, q_7^\theta \,\} \times \{\, 1, 2, 5, 6 \,\}$. Hence, we know that when we run our program from one of the initial states $1, 2, 5, 6$, line 1 influences line 3. Looking at table 7.1 (p. 101) we see we must initialize variable `c` with `T` for the influence to happen.

7.5 Comparison and conclusion

We have presented three different model checking techniques. All three techniques suffer from their complexity being exponential in the length of the formula. In most applications of model checking, LTL formulae are used to specify correct behaviour, i. e. all computations of a model should satisfy the formula. In these cases, the specification is negated and then one of the above techniques is applied. SPIN [Hol03] implements the Büchi automaton approach and processes a "never claim" to which the LTL formula can be automatically translated and for which it searches for a satisfying state sequence in the program's Büchi automaton. SMV [McM92] and NuSMV [CCGR99] are symbolic model checkers that try to falsify the input specification.[34] In order to use these model checkers for our purpose, we need to provide them with the negated LTL formula to obtain the desired results. This has also become a widespread technique in automatic test data generation with model checkers, see e. g. [HRV+03, GH99, HCL+03]. Besides, in order to apply model checkers such as SPIN and NuSMV in our setting, we have to convert an IMP program into the input language for it. Even though this is rather straight forward because all constructs of IMP have corresponding constructs in the model checkers' input languages, we have to translate the SSA representation of a program including Φ functions. In section 7.1, we have shown how to convert an IMP program into its transition graph and how to extract state sequences from such a transition graph. If we want to translate our program into the model checker's language, we have to encode this extraction in it. However, this has not yet been implemented, nor is there an implementation that can generate temporal influence conditions. All examples in this thesis have been done by hand. Hence, we do not have empirical data on whether the techniques presented here are feasible in practice. The examples in the next chapter at least suggest that for smallish programs LTL path conditions are tractable.

Nevertheless, we do want to mention some aspects of this translation. First of all, modern computer programs are written in programming languages with far more features than those IMP can offer. Most importantly, IMP lacks functions and procedures. However, in theory, if functions and procedures are not recursive, inlining is one way to compensate this.

What is much more important, is the state explosion problem. Model checkers have been successfully applied to parallel systems and network protocols being verified, where each component's state space is rather small. For our analysis data flow dependences are most interesting. However, this makes the state explosion problem much more urgent. For example, consider a program that has ten input variables of type `int`. Since every variable can take 2^{32} different values, we already have 2^{320} different initial program states. Clearly, for an explicit state model checker, which – since IMP programs are deterministic – basically simulates the program on every possible input value, this number is far too large to be tractable. If, however, there does exist a run of the program, that satisfies the influence condition, we may be lucky to find one early. Nevertheless, it will be impossible

[34] While SMV can only process CTL with fairness constraints (to which LTL can also be translated [CGL94]), NuSMV directly supports model checking LTL formulae.

to find all possible input values for which the program's runs satisfy the influence condition in reasonable time with explicit state model checkers. Symbolic model checking may be able to help in this case. Since most of the input values are handled uniformly, model checkers such as NuSMV may be able to do the job, but as we do not have any experimental data, we can only speculate.[35]

In the area of automatic test data generation, model checking has been successfully applied. In [GH99] model checkers have been run on specification models with a temporal formula designed to find an input sequence for the (deterministic) program such that a particular point in the specification is reached. In [HRV+03] they have applied such techniques to realistic applications using both explicit state (SPIN) and symbolic model checking (NuSMV). Their conclusion is that NuSMV finds the shortest test cases possible for almost every case whereas SPIN's depth first search often produces excessively long witnesses. For smaller specifications, SPIN and NuSMV have acceptable runtime requirements, but as soon as the models become larger, SPIN clearly outruns NuSMV.

We have presented temporal influence conditions in a very theoretic way. For instance, all arrays have 2^{32} array cells, a rather unrealistic assumption. Obviously, this blows up the state space and the number of state sequences. In practice however, we will almost never encounter arrays of that size. Similarly, we assume that every integer input variable can take 2^{32} different values. Even though this might be a realistic assumption, in many cases, an influence between two program statements will not be different for each input variable value. Therefore, in order to apply this approach in practice, we definitely have to abstract from concrete variable values to classes of variable values. For example, we could change the structure for $\mathcal{L}_{\mathsf{IMP}}$ to $\mathbf{A}^*_{\mathsf{IMP}}$ so that integer variables may only take values -5 to 5 plus two special values: The well-known "undefined" $\perp_i$ and "don't know" $?_i$. The latter is taken whenever an assignment would normally assign a value below -5 or above 5 to the variable. Similarly, we have a $?_b$ for the type `bool`. The interpreted function operators would then propagate the "don't know" value similarly to what they do with the undefined value. We extend the interpretation of $\cdot$ to $\cdot^{\mathbf{A}^*_{\mathsf{IMP}}}(?_b) := \mathtt{T}$. Notice that the negation operator $\neg$ does not occur in influence conditions. Thus, this is a conservative approximation of the program behaviour: For every state sequence $\Xi \in \mathcal{M}_p$ over $\mathbf{A}_{\mathsf{IMP}}$ of program p that satisfies an influence condition $\theta \in \mathsf{LTL}^{\mathcal{L}_{\mathsf{IMP}}}$, we have a state sequence $\Psi \in \mathcal{M}_p$ over $\mathbf{A}^*_{\mathsf{IMP}}$ that also satisfies θ.[36] We can then apply model checking for the reduced state space and see if the formula is satisfiable over the more abstract program model. If not, we know that it is not satisfiable by the program either. Otherwise, we check if there is a state sequence over $\mathbf{A}_{\mathsf{IMP}}$ that corresponds to the found state sequence over $\mathbf{A}^*_{\mathsf{IMP}}$ and that satisfies the formula as well. If such a state sequence does not exist, we may want to reduce our level of abstraction, e. g. increase the number of possible values for certain variables that have taken the "don't know" value, and restart model checking. The model checker FLAVERS [CCO02] is based on the idea of gradually refining the program model, in its case a trace flow graph for

[35] Unfortunately, NuSMV does not seem to provide the option to compute all initial states from which a state sequence that violates the LTL specification starts.

[36] Note that there is a homomorphism $h : \mathbf{A}_{\mathsf{IMP}} \mapsto \mathbf{A}^*_{\mathsf{IMP}}$. Then, we can set $\Psi := h \circ \Xi$.

multithreaded programs.

Another approach is to use other static analysis techniques to restrict the range of a variable. Then, we do not have to introduce such an artificial value which can have undesired effects, see e. g. section 8.1 below. It can also be possible to apply some kind of design abstraction method or to compute equivalence classes for variable values in order to reduce the state space. We can not present the details here because this is beyond the scope of this thesis. However, some ideas can be found in [Hol00, BMMR01, KPV03].

Once we have found a state sequence that satisfies the temporal influence condition in question, we can then mask all states that do not lie on the influence path and reconstruct how the influence actually happens, including possible values that flow along data flow dependence edges.

Chapter 8

Examples

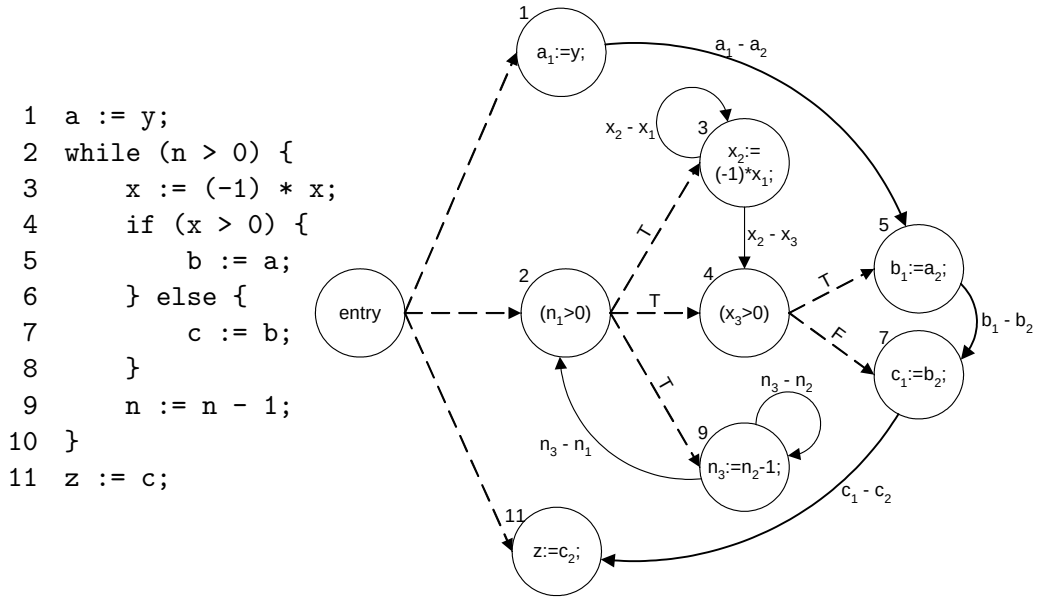
In this chapter we present three examples where we use temporal path conditions to analyze programs. As temporal path conditions have not yet been implemented, all examples are pencil-and-paper work and rather small. Nevertheless they demonstrate the potential of temporal path conditions, but obviously they can not be seen as a proof that temporal path conditions scale to larger programs. Even if we are able to generate temporal path conditions for larger programs (and in more realistic programming languages than IMP), the state explosion problem with model checking will still persist.

The first example is a purely artificial one, but it has been a standard example for path conditions. It is a simple program with one loop-carried data dependence. The next one is also a classic example, a fictitious measurement program from a cheese scale which contains a calibration path violation [Sne96]. The last example extends the second one in that setting the calibration factor can only be done after we have entered the service mode of the measurement program. This example has been motivated by many embedded systems being equipped with some sort of service mode in which maintenance features are available and which is activated by pressing a specific sequence of keys. For example, the service mode of VW cars in which internal state data is available on the air conditioning display [vwr]. In the first two examples, we are interested in proving that there exists an influence path. In the last, our aim is to validate that the weighing scale can only be calibrated after having pressed the correct sequence of states to enter the service mode.

8.1 Loop-carried dependences

Here, we present a standard example of path conditions (cf. e. g. [Kri03]) which we have already mentioned in example 6 (p. 37). Figure 8.1 shows the program p and its PDG, figure 8.2 (p. 111) shows its transition graph. We are interested in an influence between lines 1 and 11.

Figure 8.1 Loop-carried flow dependence example program with PDG.



We get for the influence condition $\mathsf{IC}(1, 11)$ (without Φ constraints):

$$\overline{\mathsf{IC}(1,11)} \Longleftrightarrow \Diamond((x_3 > 0) \wedge (n_1 > 0)\, \mathcal{U}\, ((n_1 > 0) \wedge !(x_3 > 0) \wedge \Diamond(!(n_1 > 0))))$$

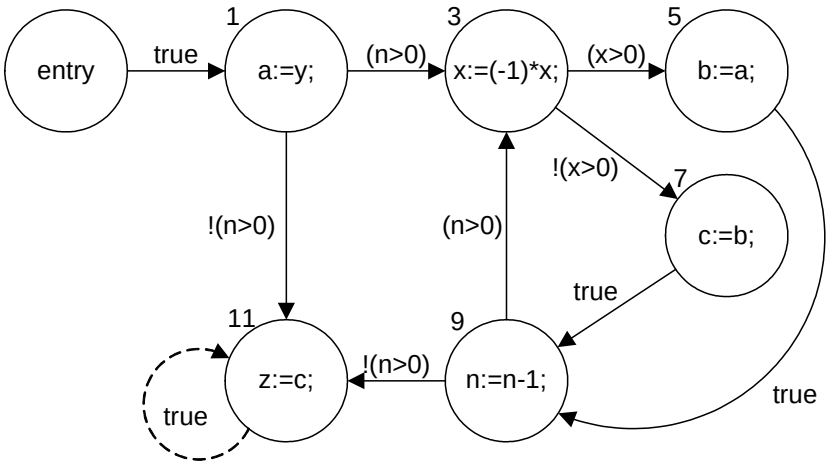
We are now able to apply a model checker like SPIN or NuSMV to the program and the influence condition to search for state sequences satisfying $\overline{\mathsf{IC}(1, 11)}$.[37] We have manually translated the transition graph shown in figure 8.2 with our semantics of how to extract state sequences thereof into SMV, the input language of NuSMV, (cf. section A.1.1 in the appendix).

In order to keep the state space small, we restricted the range of integer variables x and n to $\{ -5, \ldots, 5 \} \cup \{ \mathit{unknown} \}$, where *unknown* is a symbolic value that satisfies all constraints. This corresponds to the $?_i$ value proposed at the end of the last chapter and is a conservative approximation. If we run NuSMV on the program model and the negated LTL formula, it finds the state sequence shown in table 8.1. Note that this is not the shortest sequence possible (if `x` was initially -5 and `n` 2, we would have saved one iteration and still obtained a witness for the influence). The influence happens along the path $1 \xrightarrow[a_1 - a_2]{\mathsf{fd}} 5, 5 \xrightarrow[b_1 - b_2]{\mathsf{fd}} 7, 7 \xrightarrow[c_1 - c_2]{\mathsf{fd}} 11$. For the initial values given in the first column (state 1), the states with numbers 2,7,10,13 correspond to the nodes on the influence path.

We have also translated the program into ProMeLa (cf. section A.1.2). If we run SPIN on it, it finds a shorter sequence that starts in $\mathtt{a} = \mathtt{b} = \mathtt{c} = \mathtt{y} = \mathtt{z} = \text{true}$, $\mathtt{n} = 1$ and $\mathtt{x} = \mathit{unknown}$. Then, both $(x_3 > 0)$ and $!(x_3 > 0)$ are satisfied simultaneously (the path produced by SPIN passes through node 5) and the loop can be left after one iteration. Notice that this path satisfies the temporal path

[37] SPIN expects to be given a never claim for which it tries to find a satisfying sequence whereas NuSMV expects to be given a specification for which it tries to find a counter example. Hence, we must give $\overline{\mathsf{IC}(1, 11)}$ to SPIN and $\neg\overline{\mathsf{IC}(1, 11)}$ to NuSMV.

Figure 8.2 Transition graph for the program shown in figure 8.1.



State No. / **Variable**	1	2	3	4	5	6	7	8	9	10	11	12	13	...
a	F	F												
b	F						F							
c	F			F						F				
n	3				2			1			0			
x	5		−5			5			−5					
y	F													
z	F											F		
a_1	$\perp_b$	F												
a_2	$\perp_b$						F							
b_1	$\perp_b$						F							
b_2	$\perp_b$			F						F				
c_1	$\perp_b$			F						F				
c_2	$\perp_b$											F		
n_1	$\perp_i$		3			2			1			0		
n_2	$\perp_i$				3			2			1			
n_3	$\perp_i$				2			1			0			
x_1	$\perp_i$		5			−5			5					
x_2	$\perp_i$		−5			5			−5					
x_3	$\perp_i$			−5			5			−5				
y	$\perp_b$	F												
z	$\perp_b$											F		

Table 8.1: State sequence for the program in figure 8.1 output by NuSMV. Only assignments to variables are shown. If a cell is empty, the variable has the same value as in the previous state.

condition, but is no witness for the influence. Hence, by reducing the state space (and thus increasing the number of possible state sequences), we have made the formula less precise.

If, however, we apply the symbolic model checking approach from section 7.4.2 to the program with $\mathbf{A}_{\mathsf{IMP}}$ as structure for $\mathcal{L}_{\mathsf{IMP}}$, we obtain the following constraint on the initial assignments:

$$\xi(\{\, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{y}, \mathtt{z} \,\}) \subseteq \mathbb{B}^{\perp} \wedge (\xi(\mathtt{n}) > 1 \wedge \xi(\mathtt{x}) < 0 \vee \xi(\mathtt{n}) > 2 \wedge \xi(\mathtt{x} > 0)) \wedge \\ \xi(\{\, a_1, a_2, b_1, b_2, c_1, c_2, n_1, n_2, n_3, x_1, x_2, x_3, y, z \,\}) \subseteq \perp$$

However, we were not able to find a symbolic model checker that outputs such constraints.

8.2 A cheese scale

This example is a fictitious measurement software for a cheese scale (figure 8.3). We introduce a new expression `input` of type `int` in IMP to model I/O. We assume that `input` does not depend in any way on program variables, i. e. `input` does not generate any dependences. Semantically, whenever `input` occurs in an `int` expression, a random value from $\overline{\mathbb{Z}}$ is used for $\mathtt{input}^{\mathbf{A}_{\mathsf{IMP}}}$.[38]

In the model scale, there is a hardware port `p_weigh` for the weight sensor, an input line `p_paperout` to signal whether the scale has run out of paper, and a keyboard port `p_keyb`. Whenever a port value has been processed, we assign to it the next value from the port using `input`. We do not include an extra update for the out-of-paper signal line. This port is assumed to be constant for the analysis and modelled as a Boolean variable. There are two output variables `u_kg` and `disp`. The former contains the mass whose weight is being measured, the latter is filled with 8 characters from the keyboard which can then be displayed or printed. The display is updated whenever the keyboard buffer contains a return character (key code 13).

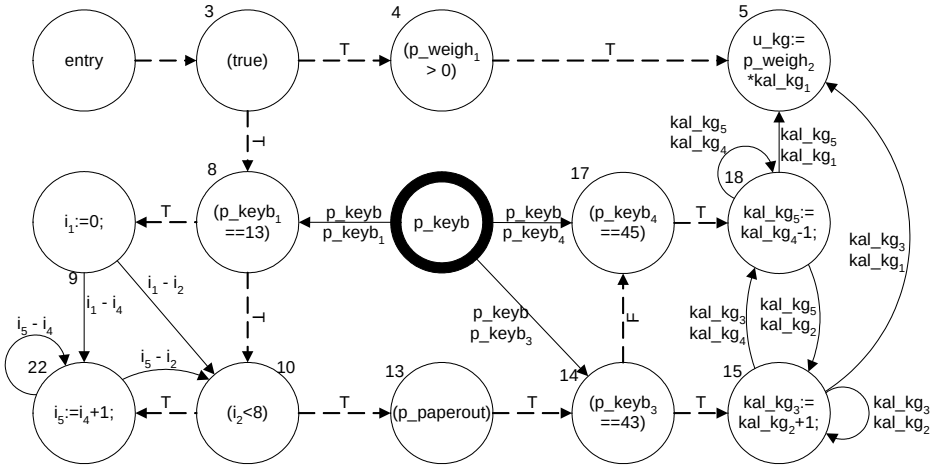
In the PDG for p we unite the nodes for lines 1, 11, and 25 in a single node called `p_keyb`. Similarly, we include only a single node `p_weigh` for lines 2 and 6. The chop for `p_keyb` and line 5 is shown in figure 8.4. We are interested whether keyboard input `p_keyb` can influence the mass amount `u_kg` that is displayed. Hence, we want to compute $\mathsf{IC}(\mathtt{p\_keyb}, 6)$ which is a shorthand for $\mathsf{IC}(8,5) \vee \mathsf{IC}(12,5) \vee \mathsf{IC}(14,5) \vee \mathsf{IC}(17,5)$. For $\mathsf{IC}(\mathtt{p\_keyb}, 5)$, we have to consider the influence paths listed in table 8.2 (p. 114).

[38] It is easy to see that the `input` construct does not affect how influence conditions are generated and simplified.

Figure 8.3 A fictitious measurement software for a cheese scale.

```
 1  p_keyb := input;
 2  p_weigh := input;
 3  while (true) {
 4      if (p_weigh > 0) {
 5          u_kg := p_weigh * kal_kg;
 6          p_weigh := input;
 7      }
 8      if (p_keyb == 13) {
 9          i := 0;
10          while (i < 8) {
11              p_keyb := input;
12              disp[i] := p_keyb;
13              if (p_paperout) {
14                  if (p_keyb == 43) {
15                      kal_kg := kal_kg + 1;
16                  } else {
17                      if (p_keyb == 45) {
18                          kal_kg := kal_kg - 1;
19                      }
20                  }
21              }
22              i := i + 1;
23          }
24      }
25      p_keyb := input;
26  }
```

Figure 8.4 Chop for `p_keyb` and line 5 of the PDG for the cheese scale program in figure 8.3.



π_1	14	15	5							
π_2	14	15	18	5						
π_3	17	18	5							
π_4	17	18	15	5						
π_5	14	17	18	5						
π_6	14	17	18	15	5					
π_7	8	10	11	17	18	5				
π_8	8	10	11	17	18	15	5			
π_9	8	10	11	14	15	5				
π_{10}	8	10	11	14	15	18	5			
π_{11}	8	10	11	14	17	18	5			
π_{12}	8	10	11	14	17	18	15	5		
π_{13}	8	10	13	14	15	5				
π_{14}	8	10	13	14	15	18	5			
π_{15}	8	10	13	14	17	18	5			
π_{16}	8	10	13	14	17	18	15	5		
π_{17}	8	9	10	11	17	18	5			
π_{18}	8	9	10	11	17	18	15	5		
π_{19}	8	9	10	11	14	15	5			
π_{20}	8	9	10	11	14	15	18	5		
π_{21}	8	9	10	11	14	17	18	5		
π_{22}	8	9	10	11	14	17	18	15	5	
π_{23}	8	9	10	13	14	15	5			
π_{24}	8	9	10	13	14	15	18	5		
π_{25}	8	9	10	13	14	17	18	5		
π_{26}	8	9	10	13	14	17	18	15	5	
π_{27}	8	9	22	10	11	17	18	5		
π_{28}	8	9	22	10	11	17	18	15	5	
π_{29}	8	9	22	10	11	14	15	5		
π_{30}	8	9	22	10	11	14	15	18	5	
π_{31}	8	9	22	10	11	14	17	18	5	
π_{32}	8	9	22	10	11	14	17	18	15	5
π_{33}	8	9	22	10	13	14	15	5		
π_{34}	8	9	22	10	13	14	15	18	5	
π_{35}	8	9	22	10	13	14	17	18	5	
π_{36}	8	9	22	10	13	14	17	18	15	5

Table 8.2: List of influence paths in $\Pi^*(\texttt{p\_keyb}, 5)$ for the program in figure 8.3 and its chop from figure 8.4.

We obtain the following influence conditions for π_1, π_2, π_3, and π_4:

$$\begin{aligned}
\Diamond\,\overline{\mathsf{IC}(\pi_1)} \Longleftrightarrow\ & \Diamond((p_keyb_1 == 13) \wedge (i_2 < 8) \wedge (p_paperout) \wedge (p_keyb_3 == 43) \wedge \\
& \Diamond((p_weigh_1 > 0) \wedge (p_weigh_2! = 0) \wedge (kal_kg_3 == kal_kg_1))) \\
\Diamond\,\overline{\mathsf{IC}(\pi_2)} \Longleftrightarrow\ & \Diamond((p_keyb_1 == 13) \wedge (i_2 < 8) \wedge (p_paperout) \wedge (p_keyb_3 == 43) \wedge \\
& \Diamond((p_keyb_1 == 13) \wedge (i_2 < 8) \wedge (p_paperout) \wedge !(p_keyb_3 == 43) \wedge \\
& (p_keyb_4 == 45) \wedge (kal_kg_3 == kal_kg_4) \wedge \Diamond((p_keyb_1 == 13) \wedge \\
& (i_2 < 8) \wedge (p_paperout) \wedge !(p_keyb_3 == 43) \wedge (p_keyb_4 == 45) \wedge \\
& \Diamond((p_weigh_1 > 0) \wedge (p_weigh_2! = 0) \wedge (kal_kg_5 == kal_kg_1))))) \\
\Diamond\,\overline{\mathsf{IC}(\pi_3)} \Longleftrightarrow\ & \Diamond((p_keyb_1 == 13) \wedge (i_2 < 8) \wedge (p_paperout) \wedge !(p_keyb_3 == 43) \wedge \\
& (p_keyb_4 == 45) \wedge \Diamond((p_weigh_1 > 0) \wedge (p_weigh_2! = 0) \wedge \\
& (kal_kg_5 == kal_kg_1))) \\
\Diamond\,\overline{\mathsf{IC}(\pi_4)} \Longleftrightarrow\ & \Diamond((p_keyb_1 == 13) \wedge (i_2 < 8) \wedge (p_paperout) \wedge !(p_keyb_3 == 43) \wedge \\
& (p_keyb_4 == 45) \wedge \Diamond((p_keyb_1 == 13) \wedge (i_2 < 8) \wedge (p_paperout) \wedge \\
& (p_keyb_3 == 43) \wedge (kal_kg_5 == kal_k g_2) \wedge \Diamond((p_keyb_1 == 13) \wedge \\
& (i_2 < 8) \wedge (p_paperout) \wedge (p_keyb_3 == 43) \wedge \Diamond((p_weigh_1 > 0) \wedge \\
& (p_weigh_2! = 0) \wedge (kal_kg_3 == kal_kg_1)))))
\end{aligned}$$

For influence paths π_5 and π_6, we have $\mathsf{IC}(\pi_5) \Longleftrightarrow \mathsf{IC}(\pi_3)$ and $\mathsf{IC}(\pi_6) \Longleftrightarrow \mathsf{IC}(\pi_4)$. For all other influence paths $\pi \in \{\, \pi_7, \ldots, \pi_{36} \,\}$, we see there is always a path $\rho \in \{\, \pi_1, \ldots, \pi_4 \,\}$ such that $\pi = (\pi^*, \rho)$ for some information flow path π^*. Hence, by definition of $\mathsf{IC}(\pi)$, we have that $\mathsf{IC}(\pi) \Longleftrightarrow \mathsf{IC}(\pi^*)[\mathsf{IC}(\rho)/\beth]$. Lemma 21 (p. 77) then gives $\Diamond\,\overline{\mathsf{IC}(\pi)} \rightarrowtail\!\!\!\Longrightarrow \Diamond\,\overline{\mathsf{IC}(\rho)}$. Thus, we do not have to generate any more influence conditions. Note further that $\Diamond\,\overline{\mathsf{IC}(14,15,18)[\Diamond\,\mathsf{IC}(\pi_3)/\beth]} \Longleftrightarrow \Diamond\,\overline{\mathsf{IC}(\pi_2)}$ and $\Diamond\,\overline{\mathsf{IC}(17,18,15)[\Diamond\,\mathsf{IC}(\pi_3)/\beth]} \Longleftrightarrow \Diamond\,\overline{\mathsf{IC}(\pi_1)}$. It follows from the construction of $\mathsf{IC}(14,15,18)$ and $\mathsf{IC}(17,18,15)$, that $\Diamond\,\overline{\mathsf{IC}(\pi_2)} \rightarrowtail\!\!\!\Longrightarrow \Diamond\,\overline{\mathsf{IC}(\pi_3)}$ and $\Diamond\,\overline{\mathsf{IC}(\pi_4)} \rightarrowtail\!\!\!\Longrightarrow \Diamond\,\overline{\mathsf{IC}(\pi_1)}$. Hence, we have $\Diamond\,\overline{\mathsf{IC}(\mathtt{p\_keyb}, 5)} \Longleftrightarrow \Diamond(\overline{\mathsf{IC}(\pi_1)} \vee \overline{\mathsf{IC}(\pi_3)})$, which can be simplified[39] further to:

$$\begin{aligned}
\Diamond\,\overline{\mathsf{IC}(\mathtt{p_keyb}, 5)} \rightarrowtail\!\!\!\Longrightarrow\ & \Diamond((p_keyb_1 == 13) \wedge (i_2 < 8) \wedge (p_paperout) \wedge \\
& ((p_keyb_3 == 43) \vee !(p_keyb_3 == 43) \wedge (p_keyb_4 == 45)) \wedge \\
& \Diamond((p_weigh_1 > 0) \wedge (p_weigh_2! = 0) \wedge \\
& ((kal_kg_3 == kal_kg_1) \vee (kal_kg_5 == kal_kg_1))))
\end{aligned}$$

When we look at the simplified condition more closely, we can already see how keyboard input can influence the weight on the display: First, there must be the key code 13 in the keyboard buffer and the paper out signal must be on. Second, there must be either the key code 43 or 45 in the buffer, too. Then, at some later

[39] We apply the implication $\beth \wedge \daleth \vee \beth' \wedge \daleth' \rightarrowtail (\beth \vee \beth') \wedge (\daleth \vee \daleth')$ to $\beth := (p\_keyb_3 == 43)$, $\beth' := !(p\_keyb_3 == 43) \wedge (p\_keyb_4 == 45)$, $\daleth := \Diamond((p\_weigh_1 > 0) \wedge (p\_weigh_2! = 0) \wedge (kal\_kg_3 == kal\_kg_1))$, and $\daleth' := \Diamond((p\_weigh_1 > 0) \wedge (p\_weigh_2! = 0) \wedge (kal\_kg_5 == kal\_kg_1))$ from $\mathsf{IC}(\pi_1)$ and $\mathsf{IC}(\pi_3)$. All other transformations that we have applied were congruences.

stage, the weight sensor must send a positive value, i. e. something must be placed on the scale. Since this formula is rather simple and a close look at it already shows us how an influence can happen, we do not present what model checking may give as a satisfying sequence. In this case, we already see that an influence can happen even though we do not (yet) know the exact conditions on the input variables for it to happen. In the next example, we extend the measurement software and apply model checking to the influence condition we obtain there.

8.3 A cheese scale with service mode

Now, we alter the measurement software: We remove the calibration path violation (ll. 13–21 in figure 8.4 (p. 113)) from the program and add an additional service mode in which the scale can be calibrated. Figure 8.5 shows the new program. The variable `mode` is set to 5 if the software is in service mode (ll. 37–43). Otherwise, the software is in normal mode (ll. 47–59). As before, in normal mode, the weight in kg is displayed (`u_kg`, l. 48) and when the keyboard input is keycode 13 (RETURN), eight keycodes are read from the keyboard and displayed in `disp` (ll. 51–58). Additionally, the calibration factor `kal_kg` can be incremented and decremented by pressing + (keycode 43) and - (keycode 45) respectively (ll. 37–43) in service mode.

To enter the service mode, the user must press a sequence of keys, stored in the variables `s_key0` to `s_key4` (ll. 2–6). In the example, the keycodes are 65,43,66,45,13 which are ASCII encodings of A,+,B,-,RETURN. Whenever the user enters this keycode sequence, the service mode is activated (ll. 17–32). Note that all but the last keycode in the sequence are treated like in normal mode, too (`p_keyb` is not changed in lines 21-32). To leave the service mode, the user presses keycode 27 (ESC) (ll. 12–15).

Now, by using temporal path conditions, we want to make sure that the user can influence the figures on the weight display `u_kg` only if he has entered the correct keycode sequence. Again, we subsume all statements `p_keyb := input;` in a single node `p_keyb` in the PDG which has outgoing edges to nodes 12, 17, 21, 24, 27, 30, 37, 40, 51, and 55, which are the nodes where `p_keyb` is read. Similarly, we combine all statements `p_weigh := input;` in a single node `p_weigh`.

Figure 8.6 (p. 118) shows the chop for nodes `p_keyb` and 48, which is the only node in which `u_kg` occurs. Nodes for lines 1 to 6 and 51 to 58 are not part of the chop because they do not lie on paths from node `p_keyb` to node `kal_kg`.

As before, there is a large number of different influence paths from `p_keyb` to 48. However, we see that all of them have to pass either through node 37 or node 40. Hence, it is sufficient to consider only the four influence paths listed in table 8.3 (p. 118).

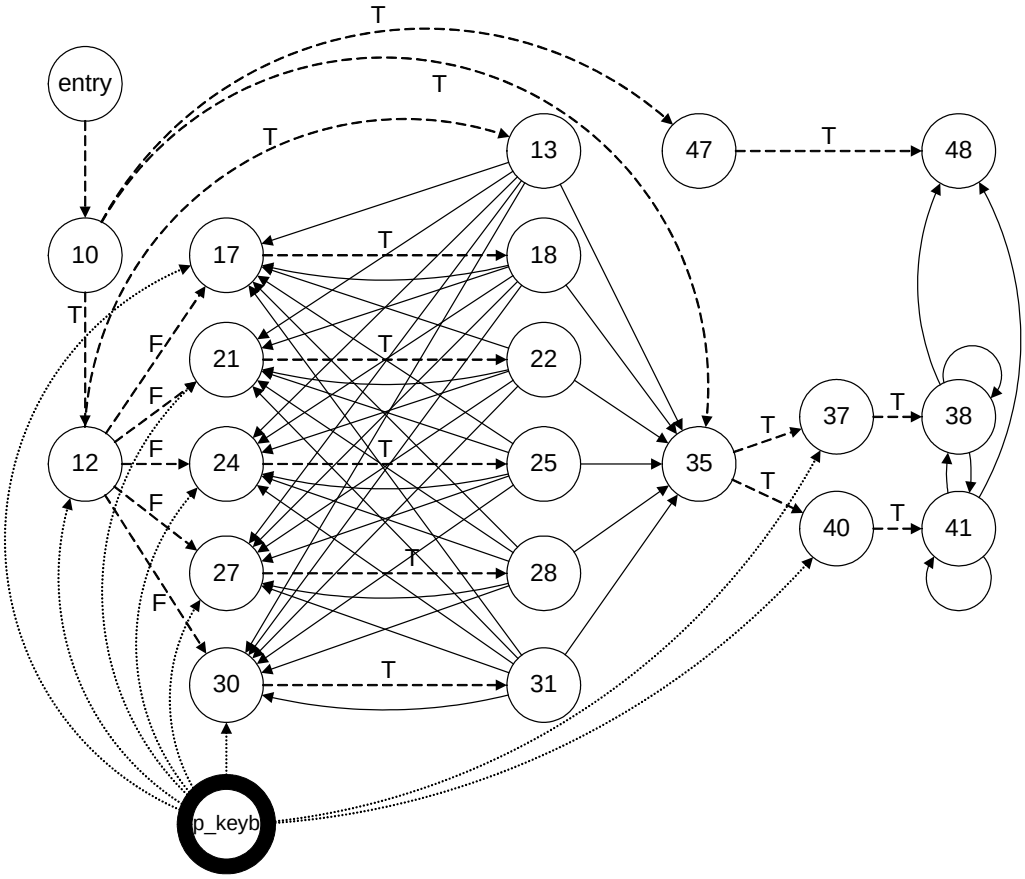
Next, we compute the influence condition $\Diamond\,\mathsf{IC}(\texttt{p\_keyb}, 48)$. By lemma 21 (p. 77), we have

$$\Diamond\,\overline{\mathsf{IC}(\texttt{p\_keyb}, 48)} \iff \bigvee_{i \in \{\,1,\ldots,4\,\}} \Diamond\,\overline{\mathsf{IC}(\pi_i)}.$$

Figure 8.5 A fictitious measurement software for a cheese scale with service mode to set the calibration factor.

```
 1  mode := 0;
 2  s_key0 := 65;
 3  s_key1 := 43;
 4  s_key2 := 66;
 5  s_key3 := 45;
 6  s_key4 := 13;
 7  p_keyb := input;
 8  p_weigh := input;
 9
10  while (true) {
11
12      if (p_keyb == 27) {
13          mode := 0;
14
15      } else {
16
17          if ((mode == 4) && (p_keyb == s_key4)) {
18              mode := 5;
19              p_keyb := input;
20          }
21          if ((mode == 3) && (p_keyb == s_key3)) {
22              mode := 4;
23          }
24          if ((mode == 2) && (p_keyb == s_key2)) {
25              mode := 3;
26          }
27          if ((mode == 1) && (p_keyb == s_key1)) {
28              mode := 2;
29          }
30          if ((mode == 0) && (p_keyb == s_key0)) {
31              mode := 1;
32          }
33      }
34
35      if (mode == 5) {
36
37          if (p_keyb == 43) {
38              kal_kg := kal_kg + 1;
39          }
40          if (p_keyb == 45) {
41              kal_kg := kal_kg - 1;
42          }
43          p_keyb := input;
44
45      }
46
47      if (p_weigh > 0) {
48          u_kg := p_weigh * kal_kg;
49      }
50
51      if (p_keyb == 13) {
52          i := 0;
53          while (i < 8) {
54              p_keyb := input;
55              disp[i] := p_keyb;
56              i := i + 1;
57          }
58      }
59
60      p_keyb := input;
61      p_weigh := input;
62  }
```

Figure 8.6 Chop on nodes `p_keyb` and 48 for the program shown in figure 8.5.



π_1	37	38	48	
π_2	37	38	41	48
π_3	40	41	48	
π_4	40	41	38	48

Table 8.3: Influence paths for figures 8.5 and 8.6 from nodes 37 and 40 to node 48.

The influence conditions for π_1 to π_4 yield:

$$\begin{aligned}
\Diamond\,\overline{\mathsf{IC}(\pi_1)} &\Longleftrightarrow \Diamond((mode_{13} == 5) \wedge (p_keyb_9 == 43) \wedge \\
&\qquad \Diamond((p_weigh_2 > 0) \wedge (kal_kg_2 == kal_kg_5) \wedge (p_weigh_3! = 0))) \\
\Diamond\,\overline{\mathsf{IC}(\pi_2)} &\Longleftrightarrow \Diamond((mode_{13} == 5) \wedge (p_keyb_9 == 43) \wedge \\
&\qquad \Diamond((mode_{13} == 5) \wedge (p_keyb_{10} == 45) \wedge (kal_kg_2 == kal_kg_3) \wedge \\
&\qquad\quad \Diamond((p_weigh_2 > 0) \wedge (kal_kg_4 == kal_kg_5) \wedge (p_weigh_3! = 0)))) \\
\Diamond\,\overline{\mathsf{IC}(\pi_3)} &\Longleftrightarrow \Diamond((mode_{13} == 5) \wedge (p_keyb_{10} == 45) \wedge \\
&\qquad \Diamond((p_weigh_2 > 0) \wedge (kal_kg_4 == kal_kg_5) \wedge (p_weigh_3! = 0))) \\
\Diamond\,\overline{\mathsf{IC}(\pi_4)} &\Longleftrightarrow \Diamond((mode_{13} == 5) \wedge (p_keyb_{10} == 45) \wedge \\
&\qquad \Diamond((mode_{13} == 5) \wedge (p_keyb_9 == 43) \wedge (kal_kg_4 == kal_kg_1) \wedge \\
&\qquad\quad \Diamond((p_weigh_2 > 0) \wedge (kal_kg_2 == kal_kg_5) \wedge (p_weigh_3! = 0))))
\end{aligned}$$

We see that $\Diamond\,\overline{\mathsf{IC}(\pi_2)} \rightarrowtail \Diamond\,\overline{\mathsf{IC}(\pi_3)}$ and $\Diamond\,\overline{\mathsf{IC}(\pi_4)} \rightarrowtail \Diamond\,\overline{\mathsf{IC}(\pi_1)}$, hence we have $\Diamond\,\overline{\mathsf{IC}(\mathtt{p\_keyb}, 48)} \Longleftrightarrow \Diamond\,\overline{\mathsf{IC}(\pi_1)} \vee \Diamond\,\overline{\mathsf{IC}(\pi_3)}$ which we can simplify[40] further to get

$$\begin{aligned}
\Diamond\,\overline{\mathsf{IC}(\mathtt{p_keyb}, 48)} \rightarrowtail \Diamond((mode_{13} == 5) \wedge ((p_keyb_9 == 43) \vee (p_keyb_{10} == 45)) \wedge \\
\Diamond((p_weigh_2 > 0) \wedge (p_weigh_3! = 0) \wedge \\
((kal_kg_2 == kal_kg_5) \vee (kal_kg_4 == kal_kg_5)))) =: \Diamond\,\theta
\end{aligned}$$

When we look at $\Diamond\,\theta$ more closely, we see that it is necessary to be in mode 5 ($mode_{13} == 5$), i. e. service mode, to influence `kal_kg` and that either keycode 43 or 45 needs to be pressed. Then, some weight must be placed on the scale ($p\_weigh_2 > 0$).

However, the formula does not tell us how the service mode is enabled, i. e. whether we can change `kal_kg` only after having pressed the correct sequence of keycodes.[41] Now, we have two options: On the one hand, we can restrict the set of information flow paths $\Pi(\mathtt{p\_keyb}, 48)$ so that we prefix the paths π_1 to π_4 with influence paths that model how the service mode is enabled. On the other hand, we can use model checking to gain further information on how to enable it.

Regarding the former, we note that initially service mode is not enabled (l. 1), hence ($mode_{13} == 5$) is not trivially fulfilled. This atomic formulae comes from the execution condition of nodes 37 and 40, i. e. is the predicate of node 35. Thus, we know that $\Diamond \bigvee_{i=1}^{4} \mathsf{IC}(35, \pi_i) \overset{\mathcal{M}_p}{\Longleftrightarrow} \Diamond\,\theta$. $mode_{13}$ in node 35 can be influenced only by nodes $j \in N := \{\, 13, 18, 22, 25, 28, 31 \,\}$. Hence we also have

$$\Diamond \bigvee_{v \in \{\, 13,18,22,25,28,31 \,\}} \bigvee_{i=1}^{4} \mathsf{IC}(v, 35, \pi) \overset{\mathcal{M}_p}{\Longleftrightarrow} \Diamond\,\theta.$$

[40] Like in footnote 39, we apply the implication $\beth \wedge \daleth \vee \beth' \wedge \daleth' \rightarrowtail (\beth \vee \beth') \wedge (\daleth \vee \daleth')$, this time to $\beth := (mode_{13} == 5) \wedge (p\_keyb_9 == 43)$, $\beth' := (mode_{13} == 5) \wedge (p\_keyb_{10} == 45)$, $\daleth := \Diamond((p\_weigh_2 > 0) \wedge (kal\_kg_2 == kal\_kg_5) \wedge (p\_weigh_3! = 0))$, and $\daleth' := \Diamond((p\_weigh_1 > 0) \wedge (kal\_kg_4 == kal\_kg_5) \wedge (p\_weigh_3! = 0))$ from $\mathsf{IC}(\pi_1)$ and $\mathsf{IC}(\pi_3)$.

[41] Note that with Boolean path conditions we would have obtained a Boolean formula similar to θ.

Note that every subformula of the disjunction on the left-hand side contains a subformula that is congruent to ${}^{j}\Phi^{35}_{mode} \wedge (mode_{13} == 5)$ for some $j \in N$. We see that this subformula is only satisfiable over $\mathcal{M}_p$ for $j = 18$, i. e. we can simplify the disjunction to $\bigvee_{i=1}^{4} \mathsf{IC}(18, 35, \pi)$. If we compute this LTL influence condition, we obtain:

$$\Diamond((mode_3 == 4) \wedge !(p\_keyb_2 == 27) \wedge (p\_keyb_5 == s\_key4_2) \wedge \Diamond((mode_4 == mode_{13}) \wedge \theta)) \overset{\mathcal{M}_p}{\Longleftrightarrow} \Diamond\theta$$

Of course, we can now repeat this approach for $(mode_3 == 4)$: we can prefix the each path $(18, 35, \pi_i)$ with $(22, 17)$. If we iterate this, we obtain the prefix 30,31,27,28,24,25,21,22,17,18,35 and then continue with one of the π_i $(1 \leq i \leq 4)$ from table 8.3 (p. 118). The disjunction over the influence conditions for the prefixed paths simplifies to

$$\begin{aligned}
&\Diamond((mode_{11} == 0) \wedge a \wedge (p_keyb_8 == s_key0_2) \wedge \\
&\Diamond((mode_9 == 1) \wedge a \wedge (p_keyb_7 == s_key1_2) \wedge (mode_{12} == mode_9) \wedge \\
&\Diamond((mode_7 == 1) \wedge a \wedge (p_keyb_6 == s_key2_2) \wedge (mode_{10} == mode_7) \wedge \\
&\Diamond((mode_5 == 1) \wedge a \wedge (p_keyb_5 == s_key3_2) \wedge (mode_8 == mode_5) \wedge \\
&\Diamond((mode_3 == 1) \wedge a \wedge (p_keyb_5 == s_key4_2) \wedge (mode_6 == mode_3) \wedge \\
&\Diamond((mode_{13} == 5) \wedge (mode_4 == mode_{13}) \wedge ((p_keyb == 43) \vee (p_keyb_{10} == 45)) \wedge \\
&\Diamond((p_weigh_2 > 0) \wedge (p_weigh_3 ! = 0) \wedge \\
&\quad ((kal_kg_2 == kal_kg_5) \vee (kal_kg_4 == kal_kg_5)))))))))
\end{aligned}$$

where a stands for the atomic formula $!(p\_keyb_2 == 27)$.

This LTL formula is still a necessary condition for `p_keyb` influencing `u_kg`. We see that in order to influence the weight that is displayed, we do have to enter the correct sequence of keycodes to enter service mode. Note however that this formula does not require that these keycodes are entered consecutively.

We explicitly see this when we apply model checking to the LTL formulae. We now convert the program into NuSMV's and SPIN's input languages and use these model checkers to model-check $\Diamond\theta$, which is equivalent to θ with respect to $\mathcal{M}_p$. Actually we want to know something about all paths that satisfy $\Diamond\theta$. Hence it would be ideal to compute all initial states where such a path starts with symbolic model checking. However, we have not been able to find a tool that can do that for us, so for now, we settle with finding any such path.

In section A.2.1, we give an SMV model for cheese scale with service mode. It is not a direct conversion of the IMP program because this would result in far too many states to be coded manually. We have applied the following measures:

- `input` statements result in the assignment variable being assigned a random value from its domain.
- SSA variants of program variables are only computed for $mode_{13}$, p_weigh_2, p_weigh_3, p_keyb_9, p_keyb_{10}, kal_kg_2, kal_kg_4, and kal_kg_5 as they are the only ones that occur in the LTL formula. Program execution is independent

of all other SSA variables because the current value of a program variable is always available by its variant in V_p.

- We did not include nodes that are not in the chop for `p_keyb` and 48. Equally, we left out all program variables that no longer occur in the program.
- We have reduced the number of possible values for most variables:
  - `p_keyb` is restricted to seven symbolic values: `key13`, `key27`, `key43`, `key45`, `key65`, `key66`, and `keyOther`.
  - `mode` is restricted to $\{ 0, \ldots, 5 \}$.
  - `kal_kg` is restricted to $\{ -5, \ldots, 5 \} \cup \{$ `unknown` $\}$.
  - `p_weigh` is restricted to $\{ 0, 1 \}$.

  All these restrictions are conservative approximations because static analysis can yield that either a variable always remains in their restricted range or, as in the case of `p_keyb`, an additional value `keyOther` is introduced to model all other possible values; similarly for `kal_kg`. `p_weigh` has been restricted to $\{ 0, 1 \}$ because it is only tested for being greater than zero or not, both in the chop of the program and in the LTL formula except in line 48, but line 48 can not influence any other statements.

When we run NuSMV on $\neg \Diamond \theta$ and the SMV model, it finds a path that violates $\neg \Diamond \theta$, i. e. that satisfies $\Diamond \theta$. They keycode sequence in this path is other, 65, 43, 66, 45, 13, 66, other, 45, other, 45, other, other, which means that this path almost directly enters the service mode and then decrements `kal_kg` twice. After having decremented `kal_kg` once, the state sequence puts some weight on the scale. The overall output path including a loop at the end contains 34 states.

To see what SPIN find, we have also written a model for the IMP program in ProMeLa (cf. A.2.2) with similar abstractions. SPIN finds another path, which is far longer. In the ProMeLa model it consists of 2053 steps. The keycode sequence for this path is:
13, 13, 13, 27, 43, 45, **65**, 13, **27**, 43, 45, **65**, 13, **43**, 43, 13, 45, 45, 65, 65, **66**, 13, 43, **45**, **13**, 13, 13, 13, 13, 13, **27**, 27, 13, 27, 13, 43, 43, 45, 45, **65**, 13, **43**, 13, 43, 13, 45, 45, 65, 65, **66**, 13, 43, **45**, **13**, 27,[42] 13, **27**, 13, 27, 13, 43, 43, 45, 45, **65**, 13, **43**, 13, 43, 13, 45, 45, 65, 65, **66**, 13, 43, **45**, **13**, *43*, 13, 13, 13, 13, 13.

Keycodes that change the `mode` variable in bold face. Keycodes that change the `kal_kg` variable are set in italics. Note that the path enters the service mode three times and aborts it with keycode 27 twice without being able to change `kal_kg`. At last, `kal_kg` is incremented by pressing keycode 43, then some weight is placed on the scale. Note that in the ProMeLa model pressing the key with keycode 13 does not result in 8 keycodes being read and transferred in `disp` because lines 51 to 58 are not part of the chop.

It is worth noticing that on this path, the service mode keycode sequence does not need to be pressed consecutively, but the measurement software can be

[42] This keycode 27 does not reset the mode because it lives only between lines 19 and 59.

switched into service mode by pressing almost randomly long enough. Without temporal path conditions and model checking we would have had a hard time to find this bug by static analyses.

Chapter 9

Conclusion

Ten years ago, path conditions were first proposed by Snelting to improve the precision of slices. In subsequent work [KSR99, RS02, Rob04, SRK] this idea has been developed further in form of Boolean path conditions to a stage where they can now be sensibly applied to medium-sized ANSI-C programs.

In this thesis, we explored how the ideas from Boolean path conditions can be transferred and extended to temporal path conditions. The contributions of this thesis can be summarized as follows:

- We present LTL path conditions for information flow paths in the program dependence graph that are satisfiable over the program model if this path can actually happen. We transferred the building blocks of Boolean path conditions (execution conditions, Φ and δ constraints) to the temporal case and came up with new constraint types (intrastatement conditions, loop termination conditions, etc.). The $\mathcal{U}$ operator is used to combine these building blocks into a single LTL formula for the path. Even though these techniques are specifically designed for a small imperative programming language, the underlying concepts can easily be transferred to other programming languages.

- An influence condition is a necessary condition for an influence to happen along any information flow path in the chop for two statements. Although there may be infinitely many, we show how to remain in finite domains by extending the generation rules for path conditions. A solution to an influence condition is a state sequence of the program, which makes the influence explicit. Due to the temporal operators, a temporal path condition already contains a temporal order on the events necessary for the influence happening. Even for small programs influence conditions can become very longish, thus they must inevitably be simplified before they can be presented to programmers and users. LTL equivalences and congruences can help a great deal in this, but there are also major simplification opportunities that exploit the properties of the specific programming language. We show that LTL path conditions are more precise than Boolean path conditions.

- When LTL formulae become more complex, we show how to use both explicit state and symbolic model checking to gain more insight into an LTL influence condition and compute a satisfying state sequence for it. For realistic programs, appropriate design abstraction techniques must be applied when we convert the imperative program to the model checker's modelling language.

- In three application examples, temporal path conditions are used to find a witness for an influence happening. We gave the path conditions derived with pencil and paper to professional model checkers, which have found witness sequences for the influence.

This thesis shows that temporal path conditions are a means to use model checking to make slicing more precise and informative. In contrast to Boolean path conditions, we have not yet found a way to make temporal path conditions scale for larger programs. The main problem here is the exponential increase in the number of influence paths relevant for the influence condition. Unfortunately, the distributivity laws for $\wedge$ and $\vee$ in Boolean logic have no equivalent for the $\mathcal{U}$ operator in LTL, thus we can not yet decompose cycles in the PDG like it is done for Boolean path conditions.

Regarding future directions, we most urgently need to have an implementation for temporal path conditions in order to apply them to more programs. However, it is still unclear how to efficiently simplify LTL formulae algorithmically. Equally, the imperative programming language must be extended by some features (e. g. I/O, data structures, etc.) before realistic application examples are possible. Ultimately, it is desirable to extend temporal path conditions to modern programming concepts like functions, procedures, modules, and objects, but there, the assumption about finite states does no longer hold. Hence, traditional model checking is no longer easily applicable.

Appendix A

Models for IMP programs

This appendix contains the source code of the handcrafted SMV and ProMeLa models we have used in the model checking application examples in chapter 8. The LTL formulae that have been adapted to match the transformational changes are also part of the model files. For further remarks on transformational aspects, see the respective sections.

The never claims for SPIN were created by SPIN with the `-F` option. For the verification runs, we ran the model checkers only with the standard options, i. e. `nusmv` <filename> and `spin -a` <filename>, `gcc -o pan pan.c`, `pan`, `spin -t -p` <filename> where <filename> is the name of the file that contains one of the programs below.

A.1 Models for the example in 8.1

A.1.1 SMV model

```
 1  MODULE main
 2  VAR
 3    state : {entry, l1, l3, l5, l7, l9, l11, error};
 4    a : Bool;
 5    b : Bool;
 6    c : Bool;
 7    n : Counter;
 8    x : Counter;
 9    y : Bool;
10    z : Bool;
11    a1 : Bool1;
12    a2 : Bool1;
13    b1 : Bool1;
14    b2 : Bool1;
15    c1 : Bool1;
16    c2 : Bool1;
17    n1 : Counter;
18    n2 : Counter;
19    n3 : Counter;
20    x1 : Counter;
21    x2 : Counter;
22    x3 : Counter;
23    y1 : Bool1;
24    z1 : Bool1;
25
```

```
26  ASSIGN
27    init(state) := entry;
28    init(a1.initialized) := 0;
29    init(a1.value) := 0;
30    init(a2.initialized) := 0;
31    init(a2.value) := 0;
32    init(b1.initialized) := 0;
33    init(b1.value) := 0;
34    init(b2.initialized) := 0;
35    init(b2.value) := 0;
36    init(c1.initialized) := 0;
37    init(c1.value) := 0;
38    init(c2.initialized) := 0;
39    init(c2.value) := 0;
40    init(n1.initialized) := 0;
41    init(n1.value) := zero;
42    init(n2.initialized) := 0;
43    init(n2.value) := zero;
44    init(n3.initialized) := 0;
45    init(n3.value) := zero;
46    init(x1.initialized) := 0;
47    init(x1.value) := zero;
48    init(x2.initialized) := 0;
49    init(x2.value) := zero;
50    init(x3.initialized) := 0;
51    init(x3.value) := zero;
52    init(y1.initialized) := 0;
53    init(y1.value) := 0;
54    init(z1.initialized) := 0;
55    init(z1.value) := 0;
56
57    init(a.initialized) := 1;
58    init(b.initialized) := 1;
59    init(c.initialized) := 1;
60    init(n.initialized) := 1;
61    init(x.initialized) := 1;
62    init(y.initialized) := 1;
63    init(z.initialized) := 1;
64
65    next(a.initialized) := case
66      state = entry : y.initialized;
67      1             : a.initialized;
68    esac;
69    next(a.value) := case
70      state = entry : y.value;
71      1             : a.value;
72    esac;
73
74    next(b.initialized) := case
75      state = l3 & x.greaterZero : a.initialized;
76      1                          : b.initialized;
77    esac;
78    next(b.value) := case
79      state = l3 & x.greaterZero : a.value;
80      1                          : b.value;
81    esac;
82
83    next(c.initialized) := case
84      state = l3 & x.notGreaterZero : b.initialized;
85      1                             : c.initialized;
86    esac;
87    next(c.value) := case
88      state = l3 & x.notGreaterZero : b.value;
89      1                             : c.value;
90    esac;
```

```
 91
 92    next(n.initialized) := case
 93      state = l5 : n.initialized;
 94      state = l7 : n.initialized;
 95      1          : n.initialized;
 96    esac;
 97    next(n.value) := case
 98      state = l5 : n.decr;
 99      state = l7 : n.decr;
100      1          : n.value;
101    esac;
102
103    next(x.initialized) := case
104      state = l1 & n.greaterZero : x.initialized;
105      state = l9 & n.greaterZero : x.initialized;
106      1                          : x.initialized;
107    esac;
108    next(x.value) := case
109      state = l1 & n.greaterZero : x.inv;
110      state = l9 & n.greaterZero : x.inv;
111      1                          : x.value;
112    esac;
113
114    next(y.initialized) := y.initialized;
115    next(y.value) := y.value;
116
117    next(z.initialized) := case
118      state = l9 & n.notGreaterZero : c.initialized;
119      1                             : z.initialized;
120    esac;
121    next(z.value) := case
122      state = l9 & n.notGreaterZero : c.value;
123      1                             : z.value;
124    esac;
125
126    next(state) := case
127      state = entry : l1;
128      state = l1 & n.greaterZero : l3;
129      state = l1 & n.notGreaterZero : l11;
130      state = l3 & x.greaterZero : l5;
131      state = l3 & x.notGreaterZero : l7;
132      state = l5 : l9;
133      state = l7 : l9;
134      state = l9 & n.greaterZero : l3;
135      state = l9 & n.notGreaterZero : l11;
136      state = l11 : l11;
137      1 : error;
138    esac;
139
140    next(a1.initialized) := case
141      state = entry : y.initialized;
142      1             : a1.initialized;
143    esac;
144    next(a1.value) := case
145      state = entry : y.value;
146      1             : a1.value;
147    esac;
148
149    next(a2.initialized) := case
150      state = l3 & x.greaterZero : a.initialized;
151      1                          : a2.initialized;
152    esac;
153    next(a2.value) := case
154      state = l3 & x.greaterZero : a.value;
155      1                          : a2.value;
```

```
156   esac;
157
158   next(b1.initialized) := case
159     state = l3 & x.greaterZero : a.initialized;
160     1                          : b1.initialized;
161   esac;
162   next(b1.value) := case
163     state = l3 & x.greaterZero : b.value;
164     1                          : b1.value;
165   esac;
166
167   next(b2.initialized) := case
168     state = l3 & x.notGreaterZero : b.initialized;
169     1                             : b2.initialized;
170   esac;
171   next(b2.value) := case
172     state = l3 & x.notGreaterZero : b.value;
173     1                             : b2.value;
174   esac;
175
176   next(c1.initialized) := case
177     state = l3 & x.notGreaterZero : b.initialized;
178     1                             : c1.initialized;
179   esac;
180   next(c1.value) := case
181     state = l3 & x.notGreaterZero : b.value;
182     1                             : c1.value;
183   esac;
184
185   next(c2.initialized) := case
186     state = l9 & n.notGreaterZero : c.initialized;
187     1                             : c2.initialized;
188   esac;
189   next(c2.value) := case
190     state = l9 & n.notGreaterZero : c.value;
191     1                             : c2.value;
192   esac;
193
194   next(n1.initialized) := case
195     state = l1 : n.initialized;
196     state = l9 : n.initialized;
197     1          : n1.initialized;
198   esac;
199   next(n1.value) := case
200     state = l1 : n.value;
201     state = l9 : n.value;
202     1          : n1.value;
203   esac;
204
205   next(n2.initialized) := case
206     state = l5 : n.initialized;
207     state = l7 : n.initialized;
208     1          : n2.initialized;
209   esac;
210   next(n2.value) := case
211     state = l5 : n.value;
212     state = l7 : n.value;
213     1          : n2.value;
214   esac;
215
216   next(n3.initialized) := case
217     state = l5 : n.initialized;
218     state = l7 : n.initialized;
219     1          : n3.initialized;
220   esac;
```

```
221   next(n3.value) := case
222     state = l5 : n.decr;
223     state = l7 : n.decr;
224     1          : n3.value;
225   esac;
226
227   next(x1.initialized) := case
228     state = l1 & n.greaterZero : x.initialized;
229     state = l9 & n.greaterZero : x.initialized;
230     1                          : x1.initialized;
231   esac;
232   next(x1.value) := case
233     state = l1 & n.greaterZero : x.value;
234     state = l9 & n.greaterZero : x.value;
235     1                          : x1.value;
236   esac;
237
238   next(x2.initialized) := case
239     state = l1 & n.greaterZero : x.initialized;
240     state = l9 & n.greaterZero : x.initialized;
241     1                          : x2.initialized;
242   esac;
243   next(x2.value) := case
244     state = l1 & n.greaterZero : x.inv;
245     state = l9 & n.greaterZero : x.inv;
246     1                          : x2.value;
247   esac;
248
249   next(x3.initialized) := case
250     state = l3 : x.initialized;
251     1          : x3.initialized;
252   esac;
253   next(x3.value) := case
254     state = l3 : x.value;
255     1          : x3.value;
256   esac;
257
258   next(y1.initialized) := case
259     state = entry : y.initialized;
260     1             : y1.initialized;
261   esac;
262   next(y1.value) := case
263     state = entry : y.value;
264     1             : y1.value;
265   esac;
266
267   next(z1.initialized) := case
268     state = l9 & n.notGreaterZero : c.initialized;
269     1                             : z1.initialized;
270   esac;
271   next(z1.value) := case
272     state = entry : c.value;
273     1             : z1.value;
274   esac;
275
276 LTLSPEC ! (F (x3.greaterZero & n1.greaterZero & (n1.greaterZero U (n1.greaterZero &
        x3.notGreaterZero & F ( n1.notGreaterZero)))))
277
278
279 MODULE Bool
280 VAR
281   initialized : boolean;
282   value : boolean;
283 DEFINE
284   valid := initialized & value;
```

```
285
286
287  MODULE Bool1
288  VAR
289    initialized : boolean;
290    value : boolean;
291  ASSIGN
292  DEFINE
293    valid := initialized & value;
294
295
296  MODULE Counter
297  VAR
298    initialized : boolean;
299    value : {neg5, neg4, neg3, neg2, neg1, zero, pos1, pos2, pos3, pos4, pos5,
         unknown};
300  ASSIGN
301  DEFINE
302    greaterZero := initialized & (value = pos1 | value = pos2 | value = pos3 | value =
         pos4 | value = pos5 | value = unknown);
303
304    notGreaterZero := initialized & (!greaterZero | value = unknown);
305
306    decr := case
307       !initialized : value;
308       value = neg5 : unknown;
309       value = neg4 : neg5;
310       value = neg3 : neg4;
311       value = neg2 : neg3;
312       value = neg1 : neg2;
313       value = zero : neg1;
314       value = pos1 : zero;
315       value = pos2 : pos1;
316       value = pos3 : pos2;
317       value = pos4 : pos3;
318       value = pos5 : pos4;
319       value = unknown : unknown;
320    esac;
321
322    inv := case
323       !initialized : value;
324       value = neg5 : pos5;
325       value = neg4 : pos4;
326       value = neg3 : pos3;
327       value = neg2 : pos2;
328       value = neg1 : pos1;
329       value = zero : zero;
330       value = pos1 : neg1;
331       value = pos2 : neg2;
332       value = pos3 : neg3;
333       value = pos4 : neg4;
334       value = pos5 : neg5;
335       value = unknown : unknown;
336    esac;
```

A.1.2 ProMeLa model

```
1  mtype = {neg5, neg4, neg3, neg2, neg1, zero, pos1, pos2, pos3, pos4, pos5, unknown};
2
3  bool  a_init = true;
4  bool  a_val;
5  bool  b_init = true;
6  bool  b_val;
7  bool  c_init = true;
8  bool  c_val;
```

```
 9  bool  n_init = true;
10  mtype n_val;
11  bool  x_init = true;
12  mtype x_val;
13  bool  y_init = true;
14  bool  y_val;
15  bool  z_init = true;
16  bool  z_val;
17  bool  a1_init = false;
18  bool  a1_val;
19  bool  a2_init = false;
20  bool  a2_val;
21  bool  b1_init = false;
22  bool  b1_val;
23  bool  b2_init = false;
24  bool  b2_val;
25  bool  c1_init = false;
26  bool  c1_val;
27  bool  c2_init = false;
28  bool  c2_val;
29  bool  n1_init = false;
30  mtype n1_val;
31  bool  n2_init = false;
32  mtype n2_val;
33  bool  n3_init = false;
34  mtype n3_val;
35  bool  x1_init = false;
36  mtype x1_val;
37  bool  x2_init = false;
38  mtype x2_val;
39  bool  x3_init = false;
40  mtype x3_val;
41  bool  y1_init = false;
42  bool  y1_val;
43  bool  z1_init = false;
44  bool  z1_val;
45
46
47  active proctype main() {
48    if
49    :: a_val = true;
50    :: a_val = false;
51    fi;
52    if
53    :: b_val = true;
54    :: b_val = false;
55    fi;
56    if
57    :: c_val = true;
58    :: c_val = false;
59    fi;
60    if
61    :: n_val = neg5;
62    :: n_val = neg4;
63    :: n_val = neg3;
64    :: n_val = neg2;
65    :: n_val = neg1;
66    :: n_val = zero;
67    :: n_val = pos1;
68    :: n_val = pos2;
69    :: n_val = pos3;
70    :: n_val = pos4;
71    :: n_val = pos5;
72    :: n_val = unknown;
73    fi;
```

```
 74    if
 75    :: x_val = neg5;
 76    :: x_val = neg4;
 77    :: x_val = neg3;
 78    :: x_val = neg2;
 79    :: x_val = neg1;
 80    :: x_val = zero;
 81    :: x_val = pos1;
 82    :: x_val = pos2;
 83    :: x_val = pos3;
 84    :: x_val = pos4;
 85    :: x_val = pos5;
 86    :: x_val = unknown;
 87    fi;
 88    if
 89    :: y_val = true;
 90    :: y_val = false;
 91    fi;
 92    if
 93    :: z_val = true;
 94    :: z_val = false;
 95    fi;
 96
 97    d_step{ a_init = true; a_val = y_val; y1_init = true; y1_val = y_val; a1_init =
           true; a1_val = y_val; }
 98    d_step{ n1_init = true; n1_val = n_val; }
 99    do
100    :: n_val == pos5 || n_val == pos4 || n_val == pos3 || n_val == pos2 || n_val ==
           pos1 || n_val == unknown ->
101      d_step{ n1_init = true; n1_val = n_val; }
102      d_step{ x1_init = true; x1_val = x_val; x2_init = true;
103        if
104        :: x_val == neg5 -> x_val = pos5;
105        :: x_val == neg4 -> x_val = pos4;
106        :: x_val == neg3 -> x_val = pos3;
107        :: x_val == neg2 -> x_val = pos2;
108        :: x_val == neg1 -> x_val = pos1;
109        :: x_val == zero -> x_val = zero;
110        :: x_val == pos1 -> x_val = neg1;
111        :: x_val == pos2 -> x_val = neg2;
112        :: x_val == pos3 -> x_val = neg3;
113        :: x_val == pos4 -> x_val = neg4;
114        :: x_val == pos5 -> x_val = neg5;
115        :: x_val == unknown -> x_val = unknown;
116        fi;
117        x2_val = x_val; }
118      d_step{ x3_init = true; x3_val = x_val; }
119      if
120      :: x_val == pos5 || x_val == pos4 || x_val == pos3 || x_val == pos2 || x_val ==
             pos1 || x_val == unknown ->
121         d_step{ a2_init = true; a2_val = a_val; b1_init = true; b1_val = a_val; b_val
                = a_val; }
122      :: x_val == neg5 || x_val == neg4 || x_val == neg3 || x_val == neg2 || x_val ==
             neg1 || x_val == zero || x_val == unknown ->
123         d_step{ b2_init = true; b2_val = b_val; c1_init = true; c1_val = b_val; c_val
                = b_val; }
124      fi;
125      d_step{ n2_init = true; n2_val = n_val; n3_init = true;
126        if
127        :: n_val == neg5 -> n_val = unknown;
128        :: n_val == neg4 -> n_val = neg5;
129        :: n_val == neg3 -> n_val = neg4;
130        :: n_val == neg2 -> n_val = neg3;
131        :: n_val == neg1 -> n_val = neg2;
132        :: n_val == zero -> n_val = neg1;
```

```
133         :: n_val == pos1 -> n_val = zero;
134         :: n_val == pos2 -> n_val = pos1;
135         :: n_val == pos3 -> n_val = pos2;
136         :: n_val == pos4 -> n_val = pos3;
137         :: n_val == pos5 -> n_val = pos4;
138         :: n_val == unknown -> n_val = unknown;
139         fi;
140         n3_val = n_val; }
141       :: n_val == neg5 || n_val == neg4 || n_val == neg3 || n_val == neg2 || n_val ==
              neg1 || n_val == zero || n_val == unknown ->
142        d_step{ n1_init = true; n1_val = n_val; }
143        break;
144     od;
145     c2_init = true; c2_val = c_val; z1_init = true; z1_val = c_val;
146   }
147
148
149   #define x3g0 (x3_init && (x3_val == pos5 || x3_val == pos4 || x3_val == pos3 ||
          x3_val == pos2 || x3_val == pos1 || x3_val == unknown))
150   #define n1g0 (n1_init && (n1_val == pos5 || n1_val == pos4 || n1_val == pos3 ||
          n1_val == pos2 || n1_val == pos1 || n1_val == unknown))
151   #define x3ng0 (x3_init && (x3_val == neg5 || x3_val == neg4 || x3_val == neg3 ||
          x3_val == neg2 || x3_val == neg1 || x3_val == zero || x3_val == unknown))
152   #define n1ng0 (n1_init && (n1_val == neg5 || n1_val == neg4 || n1_val == neg3 ||
          n1_val == neg2 || n1_val == neg1 || n1_val == zero || n1_val == unknown))
153
154
155   never {    /* <>(x3g0 && n1g0 U ( n1g0 && x3ng0 && <> (n1ng0)))
156    */
157   T0_init:
158           if
159           :: ((n1g0) && (n1ng0) && (x3g0) && (x3ng0)) -> goto accept_all
160           :: ((n1g0) && (x3g0) && (x3ng0)) -> goto T0_S10
161           :: ((n1g0) && (x3g0)) -> goto T0_S9
162           :: (1) -> goto T0_init
163           fi;
164   T0_S10:
165           if
166           :: ((n1ng0)) -> goto accept_all
167           :: (1) -> goto T0_S10
168           fi;
169   T0_S9:
170           if
171           :: ((n1g0) && (n1ng0) && (x3ng0)) -> goto accept_all
172           :: ((n1g0) && (x3ng0)) -> goto T0_S10
173           :: ((n1g0)) -> goto T0_S9
174           fi;
175   accept_all:
176           skip
177   }
```

A.2 Models for the example in 8.3

A.2.1 SMV model

```
1   MODULE main
2   VAR
3     state : {entry, l13, l18, l19, l22, l25, l28, l31, l38, l41, l43, l48, l60, l61,
          errorState};
4     p_keyb : {key13, key27, key43, key45, key65, key66, keyOther};
5     p_weigh : 0..1;
6     mode : 0..5;
7     kal_kg : Int;
8     u_kg : Int;
```

```
 9    mode13 : Model;
10    p_weigh2 : Weightl;
11    p_weigh3 : Weightl;
12    p_keyb9 : Keyboardl;
13    p_keyb10 : Keyboardl;
14    kal_kg2 : Intl;
15    kal_kg4 : Intl;
16    kal_kg5 : Intl;
17
18  ASSIGN
19    init(state) := entry;
20    init(mode) := 0;
21    init(p_weigh) := weight;
22    init(p_keyb) := keyboard;
23    init(u_kg.value) := 0;
24    init(u_kg.unknown) := 0;
25
26    next(state) := case
27      state = entry : whileloopstate;
28      state = l13: postfirstifstate;
29      state = l18: l19;
30      state = l19 : postfirstifstate;
31      state = l22 : postfirstifstate;
32      state = l25 : postfirstifstate;
33      state = l28 : postfirstifstate;
34      state = l31 : postfirstifstate;
35      state = l38 : l43;
36      state = l41 : l43;
37      state = l43 : case
38        p_weigh > 0 : l48;
39        1 : l60;
40      esac;
41      state = l48 : l60;
42      state = l60 : l61;
43      state = l61 : whileloopstate;
44      1 : errorState;
45    esac;
46
47    next(p_keyb) := case
48      next(state) = l19 : keyboard;
49      next(state) = l43 : keyboard;
50      next(state) = l60 : keyboard;
51      1 : p_keyb;
52    esac;
53
54    next(p_weigh) := case
55      next(state) = l61 : weight;
56      1 : p_weigh;
57    esac;
58
59    next(mode) := case
60      next(state) = l13 : 0;
61      next(state) = l18 : 5;
62      next(state) = l22 : 4;
63      next(state) = l25 : 3;
64      next(state) = l28 : 2;
65      next(state) = l31 : 1;
66      1 : mode;
67    esac;
68
69    next(kal_kg.value) := case
70      next(state) = l38 : kal_kg.incr;
71      next(state) = l41 : kal_kg.decr;
72      1 : kal_kg.value;
73    esac;
```

```
74    next(kal_kg.unknown) := case
75      next(state) = l38 : kal_kg.incrunknown;
76      next(state) = l41 : kal_kg.decrunknown;
77      1 : kal_kg.unknown;
78    esac;
79
80    next(u_kg.value) := case
81      next(state) = l48 : next(kal_kg.value) * next(p_weigh);
82      1 : u_kg.value;
83    esac;
84    next(u_kg.unknown) := case
85      next(state) = l48 : next(kal_kg.unknown);
86      1 : u_kg.unknown;
87    esac;
88
89    next(mode13.initialized) := mode13.initialized | ((state in { l13, l18, l22, l25,
          l28, l31, l61 }) & (next(state) in { l38, l41, l43, l48, l60 }));
90    next(mode13.value) := case
91      (state in { l13, l18, l22, l25, l28, l31, l61 }) & (next(state) in { l38, l41,
            l43, l48, l60 }) : mode;
92      1 : mode13.value;
93    esac;
94
95    next(p_weigh2.initialized) := p_weigh2.initialized | ((state in { l13, l18, l22,
          l25, l28, l31, l43, l61 }) & (next(state) in { l48, l60 }));
96    next(p_weigh2.value) := case
97      (state in { l13, l18, l22, l25, l28, l31, l43, l61 }) & (next(state) in { l48,
            l60 }) : p_weigh;
98      1 : p_weigh2.value;
99    esac;
100
101   next(p_weigh3.initialized) := p_weigh3.initialized | next(state) = l48;
102   next(p_weigh3.value) := case
103     next(state) = l48 : p_weigh;
104     1 : p_weigh3.value;
105   esac;
106
107   next(p_keyb9.initialized) := p_keyb9.initialized | ((state in { l13, l18, l22, l25,
          l28, l31, l61 }) & (next(state) in { l38, l41, l43 }));
108   next(p_keyb9.value) := case
109     (state in { l13, l18, l22, l25, l28, l31, l61 }) & (next(state) in { l38, l41,
            l43 }) : p_keyb;
110     1 : p_keyb9.value;
111   esac;
112
113   next(p_keyb10.initialized) := p_keyb10.initialized | ((state in { l13, l18, l22,
          l25, l28, l31, l38, l61 }) & (next(state) in { l41, l43 }));
114   next(p_keyb10.value) := case
115     (state in { l13, l18, l22, l25, l28, l31, l38, l61 }) & (next(state) in { l41,
            l43 }) : p_keyb;
116     1 : p_keyb10.value;
117   esac;
118
119   next(kal_kg2.initialized) := kal_kg2.initialized | next(state) = l38;
120   next(kal_kg2.value.value) := case
121     next(state) = l38 : kal_kg.value;
122     1 : kal_kg2.value.value;
123   esac;
124   next(kal_kg2.value.unknown) := case
125     next(state) = l38 : kal_kg.unknown;
126     1 : kal_kg2.value.unknown;
127   esac;
128
129   next(kal_kg4.initialized) := kal_kg4.initialized | next(state) = l41;
130   next(kal_kg4.value.value) := case
```

```
131       next(state) = l41 : kal_kg.value;
132       1 : kal_kg4.value.value;
133     esac;
134     next(kal_kg4.value.unknown) := case
135       next(state) = l41 : kal_kg.unknown;
136       1 : kal_kg4.value.unknown;
137     esac;
138
139     next(kal_kg5.initialized) := kal_kg5.initialized | next(state) = l48;
140     next(kal_kg5.value.value) := case
141       next(state) = l48 : kal_kg.value;
142       1 : kal_kg5.value.value;
143     esac;
144     next(kal_kg5.value.unknown) := case
145       next(state) = l48 : kal_kg.unknown;
146       1 : kal_kg5.value.unknown;
147     esac;
148
149
150   DEFINE
151     keyboard := {key13, key27, key43, key45, key65, key66, keyOther};
152     weight := {0, 1};
153
154     postfirstifstate := case
155       mode = 5 & p_keyb = key43 : l38;
156       mode = 5 & p_keyb = key45 : l41;
157       mode = 5 : l43;
158       p_weigh > 0 : l48;
159       1 : l60;
160     esac;
161
162     whileloopstate := case
163       p_keyb = key27 : l13;
164       mode = 4 & p_keyb = key13 : l18;
165       mode = 3 & p_keyb = key45 : l22;
166       mode = 2 & p_keyb = key66 : l25;
167       mode = 1 & p_keyb = key43 : l28;
168       mode = 0 & p_keyb = key65 : l31;
169       1 : postfirstifstate;
170     esac;
171
172   LTLSPEC !(F(mode13.initialized & mode13.value = 5 & ((p_keyb9.initialized &
          p_keyb9.value = key43) | (p_keyb10.initialized & p_keyb10.value = key45)) &
          F(p_weigh2.initialized & p_weigh2.value > 0 & p_weigh3.initialized &
          p_weigh3.value != 0 & ((kal_kg2.initialized & kal_kg5.initialized &
          kal_kg2.value.unknown = kal_kg5.value.unknown & (kal_kg2.value.value =
          kal_kg5.value.value | kal_kg2.value.unknown)) | (kal_kg4.initialized &
          kal_kg5.initialized & kal_kg4.value.unknown = kal_kg5.value.unknown &
          (kal_kg4.value.value = kal_kg5.value.value | kal_kg4.value.unknown))))))
173
174   MODULE Int
175   VAR
176     unknown : boolean;
177     value : -5..5;
178   DEFINE
179     incr := case
180       unknown : unknown;
181       value = 5 : unknown;
182       1 : value + 1;
183     esac;
184     incrunknown := unknown | value = 5;
185     decr := case
186       unknown : unknown;
187       value = -5 : unknown;
188       1 : value - 1;
```

```
189    esac;
190    decrunknown := unknown | value = -5;
191
192  MODULE Intl
193  VAR
194    initialized : boolean;
195    value : Int;
196  ASSIGN
197    init(initialized) := 0;
198
199  MODULE Model
200  VAR
201    initialized : boolean;
202    value : 0..5;
203  ASSIGN
204    init(initialized) := 0;
205
206  MODULE Weightl
207  VAR
208    initialized : boolean;
209    value : 0..2;
210  ASSIGN
211    init(initialized) := 0;
212
213  MODULE Keyboardl
214  VAR
215    initialized : boolean;
216    value : {key13, key27, key43, key45, key65, key66, keyOther};
217  ASSIGN
218    init(initialized) := 0;
```

A.2.2 ProMeLa model

```
 1  mtype = {key13, key27, key43, key45, key65, key66, keyOther};
 2
 3  mtype p_keyb;
 4  int   p_weigh;
 5  byte  mode;
 6  int   kal_kg;
 7  bool  kal_kg_unknown;
 8  bool  mode13_init = false;
 9  int   mode13_val;
10  bool  p_weigh2_init = false;
11  int   p_weigh2_val;
12  bool  p_weigh3_init = false;
13  int   p_weigh3_val;
14  bool  p_keyb9_init = false;
15  mtype p_keyb9_val;
16  bool  p_keyb10_init = false;
17  mtype p_keyb10_val;
18  bool  kal_kg2_init = false;
19  int   kal_kg2_val;
20  bool  kal_kg2_unknown;
21  bool  kal_kg4_init = false;
22  int   kal_kg4_val;
23  bool  kal_kg4_unknown;
24  bool  kal_kg5_init = false;
25  int   kal_kg5_val;
26  bool  kal_kg5_unknown;
27  int   u_kg;
28  bool  u_kg_unknown;
29
30
31  active proctype scale() {
32    mode = 0;
```

```
33    if
34    :: p_keyb = key13;
35    :: p_keyb = key27;
36    :: p_keyb = key43;
37    :: p_keyb = key45;
38    :: p_keyb = key65;
39    :: p_keyb = key66;
40    :: p_keyb = keyOther;
41    fi;
42
43    do
44    :: if
45       :: (p_keyb == key27) ->
46         mode = 0;
47       :: else ->
48          if
49          :: ((mode == 4) && (p_keyb == key13)) ->
50              mode = 5;
51              if
52              :: p_keyb = key13;
53              :: p_keyb = key27;
54              :: p_keyb = key43;
55              :: p_keyb = key45;
56              :: p_keyb = key65;
57              :: p_keyb = key66;
58              :: p_keyb = keyOther;
59              fi
60          :: else
61          fi;
62          if
63          :: ((mode == 3) && (p_keyb == key45)) -> mode = 4;
64          :: else
65          fi;
66          if
67          :: ((mode == 2) && (p_keyb == key66)) -> mode = 3;
68          :: else
69          fi;
70          if
71          :: ((mode == 1) && (p_keyb == key43)) -> mode = 2;
72          :: else
73          fi;
74          if
75          :: ((mode == 0) && (p_keyb == key65)) -> mode = 1;
76          :: else
77          fi;
78       fi;
79       d_step { mode13_init = true; mode13_val = mode; }
80       if
81       :: (mode == 5) ->
82         d_step { p_keyb9_init = true; p_keyb9_val = p_keyb; }
83         if
84         :: (p_keyb == key43) ->
85           d_step { if
86                    :: kal_kg < 5 && !kal_kg_unknown -> kal_kg++;
87                    :: else -> kal_kg = 0; kal_kg_unknown = true;
88                    fi; kal_kg2_init = true; kal_kg2_val = kal_kg; kal_kg2_unknown =
                          kal_kg_unknown; }
89         :: else
90         fi;
91         d_step { p_keyb10_init = true; p_keyb10_val = p_keyb; }
92         if
93         :: (p_keyb == key45) ->
94           d_step { if
95                    :: kal_kg > -5 && !kal_kg_unknown -> kal_kg--;
96                    :: else -> kal_kg = 0; kal_kg_unknown = true;
```

```
 97                   fi; kal_kg4_init = true; kal_kg4_val = kal_kg; kal_kg4_unknown =
                           kal_kg_unknown; }
 98         :: else
 99         fi;
100         if
101         :: p_keyb = key13;
102         :: p_keyb = key27;
103         :: p_keyb = key43;
104         :: p_keyb = key45;
105         :: p_keyb = key65;
106         :: p_keyb = key66;
107         :: p_keyb = keyOther;
108         fi
109       :: else
110       fi;
111
112       d_step { p_weigh2_init = true; p_weigh2_val = p_weigh; }
113       if
114       :: (p_weigh > 0) ->
115         d_step { p_weigh3_init = true; p_weigh3_val = p_weigh; kal_kg5_init = true;
                kal_kg5_val = kal_kg; kal_kg5_unknown = kal_kg_unknown; }
116         if
117         :: kal_kg_unknown -> u_kg = 0; u_kg_unknown = true;
118         :: else -> u_kg = kal_kg * p_weigh;
119         fi
120       :: else
121       fi;
122
123       if
124       :: p_keyb = key13;
125       :: p_keyb = key27;
126       :: p_keyb = key43;
127       :: p_keyb = key45;
128       :: p_keyb = key65;
129       :: p_keyb = key66;
130       :: p_keyb = keyOther;
131       fi;
132
133       if
134       :: p_weigh = 0;
135       :: p_weigh = 1;
136       fi
137     od
138   }
139
140   #define mode13e5    (mode13_init && mode13_val == 5)
141   #define p_keyb9e43  (p_keyb9_init && p_keyb9_val == key43)
142   #define p_keyb10e45 (p_keyb10_init && p_keyb10_val == key45)
143   #define p_weigh2g0  (p_weigh2_init && p_weigh2_val > 0)
144   #define p_weigh3n0  (p_weigh3_init && p_weigh3_val != 0)
145   #define kal_kg2e5   (kal_kg2_init && kal_kg5_init && kal_kg2_val == kal_kg5_val)
146   #define kal_kg4e5   (kal_kg4_init && kal_kg5_init && kal_kg4_val == kal_kg5_val)
147
148   never {    /* <>(mode13e5 && ((p_keyb9e43) || (p_keyb10e45)) && <>((p_weigh2g0) &&
          (p_weigh3n0) && ((kal_kg2e5) || (kal_kg4e5))))
149    */
150   T0_init:
151           if
152           :: ((((kal_kg2e5) && (mode13e5) && (p_keyb10e45) && (p_weigh2g0) &&
                  (p_weigh3n0)) || ((((kal_kg4e5) && (mode13e5) && (p_keyb10e45) &&
                  (p_weigh2g0) && (p_weigh3n0)) || ((((kal_kg2e5) && (mode13e5) &&
                  (p_keyb9e43) && (p_weigh2g0) && (p_weigh3n0)) || ((kal_kg4e5) &&
                  (mode13e5) && (p_keyb9e43) && (p_weigh2g0) && (p_weigh3n0)))))))) ->
                  goto accept_all
153           :: ((((mode13e5) && (p_keyb10e45)) || ((mode13e5) && (p_keyb9e43)))) -> goto
```

```
                T0_S6
154         :: (1) -> goto T0_init
155         fi;
156 T0_S6:
157         if
158         :: ((((kal_kg2e5) && (p_weigh2g0) && (p_weigh3n0)) || ((kal_kg4e5) &&
                (p_weigh2g0) && (p_weigh3n0)))) -> goto accept_all
159         :: (1) -> goto T0_S6
160         fi;
161 accept_all:
162         skip
163 }
```

Bibliography

[All70] Francis E. Allen. Control Flow Analysis. In *Proceedings of an ACM SIGPLAN Symposium on Compiler Optimization*, volume 5(7) of *SIGPLAN Notices*, pages 1–19, 1970.

[BCM+90] J. R. Burch, E. M. Clarke, Kenneth L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[BLP73] D. E. Bell and L. J. La Padula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corporation, 1973.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 36(5) of *SIGPLAN Notices*, pages 203–213, Snowbird, UT, June 2001.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[Büc60] J. R. Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1960.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A New Symbolic Model Verifier. In *Computer Aided Verification*, pages 495–499, 1999.

[CCO02] Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil. FLAVERS: a Finite State Verification Technique for Software Systems. *IBM Systems Journal*, 41(1):140–165, 2002.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment

Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGMZ95] Edmund M. Clarke, Orna Grumberg, Kenneth L. McMillan, and Xudong Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *Proceedings of the 32nd Conference on Design Automation (DAC 95)*, pages 427–432, San Francisco, CA, 1995. ACM press.

[CGP00] Edmund M. Clarke, Jr, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, London, 2000.

[Com99] Common Criteria Project Sponsoring Organizations. Common Criteria for Informaiton Technology Security Evaluation, 1999. ISO/IEC 15408.

[CVWY92] Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GH99] Angelo Gargantini and Constance Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In *Proceedings of the 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162, Toulouse, France, 1999. Springer-Verlag.

[GM84] J. Goguen and J. Meseguer. Inference Control and Unwinding. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society Press, 1984.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the 15th Workshop on Protocol Specification, Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[HCL+03] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data Flow Testing as Model Checking. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 232–242, Portland, OR, May 2003. IEEE Computer Society.

[Hoa69] Charles A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hol00] Gerard J. Holzmann. Logic Verification of ANSI-C Code with SPIN. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification: Proceedings of the 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147, Stanford, CA, 2000. Springer-Verlag.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23(7) of *SIGPLAN Notices*, pages 35–46, Atlanta, GA, June 1988.

[HRV+03] Mats P. E. Heimdahl, Sanjai Rayadurgam, Willem Visser, Devaraj George, and Jimin Gao. Auto-generating Test Sequences using Model Checkers: A Case Study. In *Third International Workshop on Formal Approaches to Software Testing (FATES)*, volume 2931 of *Lecture Notes in Computer Science*, pages 42–59, Montreal, Canada, October 2003. Springer-Verlag.

[Koz83] Dexter Kozen. Results on the Propositional Mu-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Warsaw, Poland, April 2003.

[Kri03] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2003.

[KS98] Jens Krinke and Gregor Snelting. Validation of Measurement Software as an application of Slicing and Consraint Solving. *Information and Software Technology*, pages 661–675, 1998.

[KSR99] Jens Krinke, Gregor Snelting, and Torsten Robschink. Software-Sicherheitsprüfung mit VALSOFT. *Informatik Forschung und Entwicklung*, 14(2):62–73, June 1999.

[LM69] Edward S. Lowry and C. W. Medlock. Object Code Optimization. *Communications of the ACM*, 12(1):13–22, 1969.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.

[McM92] Kenneth L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, May 1992.

[MP91] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, New York, Berlin, Heidelberg, 1991.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pnu81] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.

[Pro59] R. T. Prosser. Applications of Boolean Matrices to the Analysis of Flow Diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pages 133–138. Spartan Books, 1959.

[RAB+05] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, and John Hatcliff. *A New Foundation for Control-Dependence and Slicing for Modern Program Structures*, volume 3444 of *Lecture Notes in Computer Science*, pages 77–93. Springer-Verlag, Jan 2005.

[Rob04] Torsten Robschink. *Pfadbedingungen in Abhängigkeitsgraphen und ihre Anwendung in der Softwaresicherheitstechnik*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[RS02] Torsten Robschink and Gregor Snelting. Efficient Path Conditions in Dependence Graphs. In *Proceedings of the 24th International Conference of Software Engineering (ICSE 2002)*, pages 478–488. ACM, May 2002.

[Sis83] A. Prasad Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD thesis, Harvard University, 1983.

[Sne96] Gregor Snelting. Combining Slicing and Constraint Solving for Validation of Measurement Software. In *Proceedings of the Third International Static Analysis Symposium*, pages 332–348, Aachen, September 1996.

[Sne05] Gregor Snelting. Quantifier Elimination and Information Flow Control for Software Security. In Andreas Dolzmann, Andreas Seidl, and Thomas Sturm, editors, *Algorithmic Algebra and Logic. Proceedings of the A3L 2005 Conference in Honor of the 60th Birthday of Volker Weispfenning*, pages 237–242, April 2005.

[SRK] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. To appear in ACM Transactions on Software Engineering and Methodology.

[SS98] David Schmidt and Bernhard Steffen. Program Analysis as Model Checking of Abstract Interpretations. In *Proceedings of the 5th International Static Analysis Symposium (SAS'98)*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Springer-Verlag, 1998.

[SW05] Thomas Sturm and Volker Weispfenning. Algebra und Logik, 2005. Lecture notes.

[Tar55] Alfred Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[vwr] VW-Rulez.at – Technik / Klimacodes. URL: http://www.vw-rulez.at/content/category/8/26/47/.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on on Software Engineering*, 10(4):352–357, 1984.

[Wol96] Michael J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, Inc., 1996.

Glossary

$R(A)$	Image of $A \subseteq X$ under the binary relation $R \subseteq X \times Y$, p. 8
$(\underline{A}, \tau_{\mathcal{T}}, \iota_{\mathcal{F}}, \iota_{\mathcal{R}})$	Structure for a language consisting of a universe $\underline{A}$, a type function $\tau_{\mathcal{T}}$ for the universe, and the interpretation functions $\iota_{\mathcal{F}}$ and $\iota_{\mathcal{R}}$ for function and predicate operators, p. 18
$(Q, \Sigma, \Delta, \Upsilon, F)$	The Büchi automaton with states Q over the alphabet Σ with transition relation $\Delta \subseteq Q \times \Sigma \times Q$, initial states $\Upsilon \subseteq Q$ and accepting states $F \subseteq Q$, p. 91
(Q, Υ, Δ, L)	The Kripke structure over $W \subseteq \mathcal{W}$ with states set Q, initial states $\Upsilon \subseteq Q$, transition relation $\Delta \subseteq Q \times Q$ and a labelling function $L : Q \mapsto \mathfrak{P}(W)$, p. 90
$(V, E, \odot\!\!\succ, \prec\!\!\odot)$	Multigraph with nodes V, edges E and mappings $\odot\!\!\succ, \prec\!\!\odot : E \mapsto V$ that assign each edge its source and target node, p. 8
$(\!\|\cdot\|\!)$	The typing function $\mathcal{V} \mapsto \mathcal{T}$ for typed variable identifiers, p. 18
$(\!\|e\|\!)^{\mathbf{A}}$	The set of universe elements of $\mathbf{A}$ of type $(\!\|e\|\!)$ for expression $e \in \mathfrak{E}$, p. 18
$\rightarrowtail$	Implication between LTL formulae, p. 15
$\overset{G}{\rightsquigarrow}$	$v \overset{G}{\rightsquigarrow} w$ in a graph G if there is a path $\pi : v \xrightarrow[G]{}{}^{*} w$, p. 9
$\odot\!\!\succ$	The mapping that assigns each edge in a (multi)graph its source node, p. 8
$\prec\!\!\odot$	The mapping that assigns each edge in a (multi)graph its target node, p. 8
$\overset{M}{\rightarrowtail}$	Implication between LTL formulae over a language $\mathcal{L}$ with respect to $M \subseteq \mathcal{M}^{\mathbf{A}}_{X,W}$, $X \subseteq \mathcal{V}$, $W \subseteq \mathcal{W}$, p. 21
$\xrightarrow[p]{\mathsf{cd}}{}^{*}$	The reflexive and transitive closure of the control dependence relation $\xrightarrow[p]{\mathsf{cd}}$ for an IMP program p, p. 27
$w \xrightarrow[p]{\mathsf{cd}} v$	Node v is control dependent on node w in the CFG CFG_p for the IMP program p, p. 27
$v \xrightarrow[x]{\mathsf{dd}} w$	The def-def dependence from v to w with respect to variable x, p. 30
$v \xrightarrow[x,\pi,u]{\mathsf{dd}} w$	The def-def dependence from node v to node w with respect to variable x along the CFG path π loop-carried by loop node u, p. 30

$v \xrightarrow[x,\pi]{\mathsf{dd}} w$	The def-def dependence from node v to node w with respect to variable x along the CFG path π, p. 30
$v \xrightarrow[x]{\mathsf{fd}} w$	The flow dependence from v to w with respect to variable x, p. 30
$v \xrightarrow[x,\pi,u]{\mathsf{fd}} w$	The flow dependence from node v to node w with respect to variable x along the CFG path π loop-carried by loop node u, p. 30
$v \xrightarrow[x,\pi]{\mathsf{fd}} w$	The flow dependence from node v to node w with respect to variable x along the CFG path π, p. 30
$\overline{\theta}$	Shorthand for $\theta[\mathrm{true}\,/\mathcal{W}(\theta)]$, p. 20
R^{-1}	Inverse relation of the binary relation $R \subseteq X \times Y$, p. 8
$a^{\mathbf{A}}$	The interpretation of the atomic formula $a \in \mathfrak{A}$ in structure $\mathbf{A}$, p. 18
$e^{\mathbf{A}}$	The interpretation of expression $e \in \mathfrak{E}$ in structure $\mathbf{A}$, p. 18
$\leftrightarrow\!\!\!\!\rightarrow$	Equivalence relation on LTL formulae, p. 15
$\overset{M}{\leftrightarrow\!\!\!\!\rightarrow}$	Equivalence relation between LTL formulae over a language $\mathcal{L}$ with respect to $M \subseteq \mathcal{M}^{\mathbf{A}}_{X,W}$, $X \subseteq \mathcal{V}$, $W \subseteq \mathcal{W}$, p. 21
$\theta\,\langle\!\langle \circ, \beth \rangle\!\rangle$	The operands of the innermost binary operator of type $\circ \in \{\,\mathcal{U}, \vee, \wedge, \rightarrow\,\}$ in the LTL formula θ one of which contains the propositional variable $\beth \in \mathcal{W}$, p. 12
$\theta\,\langle\!\langle \circ, \beth \,|\, \kappa \rangle\!\rangle$	The LTL formula we obtain from θ by replacing the smallest subformula of θ of type $\circ \in \{\,\mathcal{U}, \wedge, \vee, \rightarrow\,\}$ that contains the propositional variable $\beth \in \mathcal{W}$ with the LTL formula κ, p. 12
$\preceq_\pi$	$v \preceq_\pi w$ iff node $v \in \mathrm{V}(\pi)$ comes no later than node $w \in \mathrm{V}(\pi)$ in path π, p. 9
$G_1 \subseteq G_2$	G_1 is a subgraph of G_2, p. 9
$\overset{M}{\Longleftrightarrow}$	Congruence relation between LTL formulae over a language $\mathcal{L}$ with respect to $M \subseteq \mathcal{M}^{\mathbf{A}}_{X,W}$, $X \subseteq \mathcal{V}$, $W \subseteq \mathcal{W}$, p. 21
$\prec_\pi$	$v \preceq_\pi w$ iff node $v \in \mathrm{V}(\pi)$ comes before node $w \in \mathrm{V}(\pi)$ in path π, p. 9
$\overset{M}{\Longrightarrow}$	Entailment between LTL formulae over a language $\mathcal{L}$ with respect to $M \subseteq \mathcal{M}^{\mathbf{A}}_{X,W}$, $X \subseteq \mathcal{V}$, $W \subseteq \mathcal{W}$, p. 21
$e[e_1/x_1, \ldots, e_n/x_n]$	Expression e where variable identifier x_i is simultaneously replaced by expression e_i of type $(\!|x_i|\!)$, $1 \leq i \leq n$, p. 18
$\theta[\theta_1/\beth_1, \ldots, \theta_n/\beth_n]$	The LTL formula we obtain from θ by simultaneously replacing all occurrences of $\beth_i$ with θ_i, $1 \leq i \leq n$, p. 11
$\theta[\vartheta/W]$	The LTL formula we obtain from θ by replacing all propositional variables in $W \subseteq \mathcal{W}$ with the LTL formula ϑ, p. 12

$\omega[\mathfrak{R} \leftarrow P]$ The environment for the Kripke structure $\boldsymbol{\mathcal{M}} = (Q, \Upsilon, \Delta, L)$ we obtain from the environment $\omega \in \mathfrak{P}(Q)^{\mathcal{X}}$ by assigning $P \subseteq Q$ to the relational variable $\mathfrak{R} \in \mathcal{X}$, p. 96

$[\![f]\!]^{\omega}_{\boldsymbol{\mathcal{M}}}$ The set of states of the Kripke structure $\boldsymbol{\mathcal{M}}$ in which the μ-calculus formula $f \in \mathsf{MU}_{\boldsymbol{\mathcal{M}}}$ holds in the environment ω, p. 96

$[\![\theta]\!]_{\boldsymbol{\mathcal{M}}}$ The set of states of the Kripke structure $\boldsymbol{\mathcal{M}}$ modelled by the Boolean formula θ, p. 97

$\perp$ The set $\{ \perp_a, \perp_b, \perp_i \}$ of undefined values in the structure $\mathbf{A}_{\mathsf{IMP}}$, p. 25

$\models$ $\Xi \models \theta$ iff Ξ is a model for the LTL formula θ, p. 13

$f|_A$ Restriction of $f : X \mapsto Y$ on $A \subseteq X$, p. 8

$G|_W$ Subgraph of G generated by nodes $W \subseteq \mathrm{V}(G)$, p. 9

$\bigcup\!\!\cdot\, M$ Disjoint union of all sets in M, p. 8

$\bigcup\!\!\cdot_{\lambda \in \Lambda} M_\lambda$ Disjoint union over a family of sets $(M_\lambda)_{\lambda \in \Lambda}$, p. 8

$\bigcup M$ Union of all sets in M, p. 7

$\bigcup_{\lambda \in \Lambda} M_\lambda$ Union of a family of sets $(M_\lambda)_{\lambda \in \Lambda}$, p. 7

$|\pi|$ The length (number of edges) of path π, p. 9

$\mathbb{A}$ The set $\left\{ a : \overline{\mathbb{Z}}^{\perp} \mapsto \overline{\mathbb{Z}}^{\perp} \;\middle|\; a(\perp_i) = \perp_i \right\}$ of values of type `array` in the structure $\mathbf{A}_{\mathsf{IMP}}$, p. 25

$\mathfrak{A}$ The set of atomic formulae for a language $\mathcal{L}$, p. 18

$\boldsymbol{\mathcal{A}}$ A Büchi automaton, p. 91

$\mathbf{A}$ Structure for a language, represented by a quadruple $(\underline{\mathrm{A}}, \tau_{\mathcal{I}}, \iota_{\mathcal{F}}, \iota_{\mathcal{R}})$, p. 18

$\mathbf{A}_{\mathsf{IMP}}$ The structure over $\mathcal{L}_{\mathsf{IMP}}$ to interpret IMP expressions and atomic formulae in LTL formulae over, p. 24

$\mathbf{A}'_{\mathsf{IMP}}$ The structure over $\mathcal{L}_{\mathsf{IMP}}$ to interpret IMP expressions and atomic formulae in Boolean path conditions over, p. 25

$\aleph$ The Hebrew letter "aleph", usually denotes a propositional variable, p. 10

α The labelling function for edges in a control flow graph, p. 26

$\overline{\alpha}$ The labelling function for edges in the control dependence graph for an IMP program, p. 28

$\boldsymbol{\mathcal{A}}_p$ The Büchi automaton for the IMP program p, p. 91

$\mathfrak{A}_X$ The set of all atomic formulae $a \in \mathfrak{A}$ with typed variables from $X \subseteq \mathcal{V}$, i. e. $\mathcal{V}(a) \subseteq X$, p. 20

$\mathbb{B}$ Set of the Boolean truth values, $\{ \mathtt{T}, \mathtt{F} \}$, p. 8

$\mathbb{B}^{\perp}$ The set $\mathbb{B} \cup \{ \perp_i \}$ of values of type `bool` in the structure $\mathbf{A}_{\mathsf{IMP}}$, p. 25

$\beth$ The Hebrew letter "beth", usually denotes a propositional variable, p. 10

$\beth_e$	The propositional variable that we identify with the control dependence edge e, p. 58
${}^{v}\beth_{x}^{w}$	The propositional variable identified with the data dependence $v \xrightarrow[x]{\mathsf{fd}} w$ or $v \xrightarrow[x]{\mathsf{dd}} w$, p. 58
$\mathbf{BPC}(\pi)$	Boolean path condition for the PDG path π, p. 37
$\mathbf{BPC}(s,t)$	Boolean path condition for the two statements s and t, p. 37
C	The set of control dependence edges in the control dependence graph, p. 28
CDG_p	The control dependence graph for the IMP program p, p. 28
CFG_p	The control flow graph for the IMP program p, p. 26
$\mathrm{Cl}\,(\theta)$	The closure of the LTL formula θ, p. 93
D	The set of flow and def-def dependence edges in the data dependence graph, p. 31
$\daleth$	The Hebrew letter "daleth", usually denotes a propositional variable, p. 10
DDG_p	The data dependence graph for the IMP program p, p. 31
$\mathbf{def}(v)$	The set of all IMP variable occurrences that are defined in the IMP statement v, p. 29
$\delta(\rho,\eta)$	Data dependence edge conditions for the subpath ρ of def-def-use edges, η is an arbitrary LTL formula over $\mathcal{L}_{\mathsf{IMP}}$, p. 52
$\delta_G(v \xrightarrow[a]{\mathsf{fd}} w)$	The global array constraint for the flow dependence $v \xrightarrow[a]{\mathsf{fd}} w$ with respect to the array variable a, p. 40
$\delta_l\,(e)$	The loop termination condition along the data dependence edge e, p. 51
$\delta_\pi(\rho)$	The δ constraint for the maximal def-def-use subpath ρ of the information flow path π, p. 39
δ_x^e	The intrastatement condition for variable x in expression $e \in \mathfrak{E}$, p. 54
δ_x^w	The intrastatement condition for the occurrence x of the variable x read in statement w, p. 54
$\mathbf{DOM}\,(w)$	The set of nodes that dominate w in a CFG, p. 27
$v\,\mathbf{DOM}\,w$	v dominates w in a CFG, p. 26
$\mathfrak{E}$	Set of expressions, p. 18
$\mathbf{E}((v,w),\lambda)$	Execution condition for the control dependence edge $(v,w) \in C$ and label $\lambda \in \overline{\alpha}((v,w))$, p. 35

$\mathsf{E}\left(P^{f}_{e_\pi}\right)$	The joint execution condition for all cyclic paths that can be inserted in the influence path π after $f \in \mathrm{E}(\pi) \cup \{\,\epsilon\,\}$ and before $e_\pi \in \mathrm{E}(\pi)$ while maintaining the information flow condition and that are not open array dependence cycles, p. 64
$\mathsf{E}(\pi)$	Execution condition for the control dependence path $\pi : v_e \xrightarrow[\mathrm{CDG}_p]{}{}^{*} v$ in the CDG from the entry node to node v, p. 35
$\mathrm{E}(\pi)$	Set of edges in path π, p. 9
$\mathsf{E}(s)$	The execution condition for node s, p. 48
$\mathsf{E}_{\circlearrowleft}(s)$	The execution condition from loop predicates for node s, p. 48
$\mathsf{E}_{\mathsf{cd}}(s)$	Execution condition from control dependence for the node s in the PDG, p. 35
$\mathsf{E}_{\delta}(e)$	The execution condition along the data dependence edge $e \in D$, p. 51
$\mathrm{el}\,(\theta)$	The set of elementary formulae of the LTL formula θ, p. 98
E_ϱ	The joint execution condition for all nodes and edges in the PDG path ϱ, p. 60
$\mathfrak{E}_X$	The set of all expressions $e \in \mathfrak{E}$ with typed variables from $X \subseteq \mathcal{V}$, i. e. $\mathcal{V}\,(e) \subseteq X$, p. 20
$G_{\boldsymbol{\mathcal{A}}}$	The transition graph for the Büchi automaton $\boldsymbol{\mathcal{A}}$, p. 91
ג	The Hebrew letter "gimel", usually denotes a propositional variable, p. 10
$G_{\boldsymbol{\mathcal{M}}}$	The transition graph for the Kripke structure $\boldsymbol{\mathcal{M}}$, p. 90
$\mathsf{IC}(\pi)$	The influence condition for the influence path π, p. 64
$\mathsf{IC}(s,t)$	The LTL influence condition for statement s influencing statement t, p. 57
$v\,\mathsf{IDOM}\,w$	v immediately dominates w in a CFG, p. 27
IMP	The set of all IMP programs, p. 23
ι	The initial state of the Büchi automaton for an IMP program, p. 91
$\iota_{\mathcal{F}}$	Interpretation function for function operators in a structure, p. 18
$\iota_{\mathcal{R}}$	Interpretation function for predicate operators in a structure, p. 18
$\mathcal{IS}_p$	The set of all initial states for the IMP program p, p. 45
L	The labelling function for flow and def-def dependence edges in the data dependence graph, p. 31
$\mathfrak{L}\,(\boldsymbol{\mathcal{A}})$	The language accepted by the Büchi automaton $\boldsymbol{\mathcal{A}}$, p. 91
$\mathrm{L}^{c}\left(v \xrightarrow[x]{\mathsf{dd}} w\right)$	The set of loop predicate nodes that may carry the def-def dependence from v to w with respect to variable x, p. 31

$\mathrm{L}^c\left(v \xrightarrow[x]{\mathsf{fd}} w\right)$	The set of loop predicate nodes that may carry the flow dependence from v to w with respect to variable x, p. 31
$\mathcal{L}_{\mathsf{IMP}}$	The language for IMP expressions and atomic formulae, p. 24
$\mathrm{L}^t(s)$	The set of loop predicate nodes whose loop must terminate before the statement s can be executed, p. 48
$\mathbf{LTL}$	The set of LTL formulae over propositional variables, p. 10
$\mathbf{LTL}^{\mathcal{L}}$	The set of all LTL formulae over the language $\mathcal{L}$, p. 19
$\mathbf{LTL}^{\mathcal{L}}_X$	The set of LTL formulae over $\mathcal{L}$ with typed variables from $X \subseteq \mathcal{V}$ and no propositional variables, p. 20
$\mathbf{LTL}^{\mathcal{L}}_{X,W}$	The set of LTL formulae over $\mathcal{L}$ with typed variables from $X \subseteq \mathcal{V}$ and propositional variables from $W \subseteq \mathcal{W}$, p. 20
$\mathrm{L}^x\left(v \xrightarrow[x]{\mathsf{dd}} w\right)$	The set of loop predicate nodes that are left by the def-def dependence $v \xrightarrow[x]{\mathsf{dd}} w$, p. 31
$\mathrm{L}^x\left(v \xrightarrow[x]{\mathsf{fd}} w\right)$	The set of loop predicate nodes that are left by the flow dependence $v \xrightarrow[x]{\mathsf{fd}} w$, p. 31
$\boldsymbol{\mathcal{M}}$	A Kripke structure, p. 90
$\mathcal{M}^{\mathbf{A}}_X$	The set of all state sequences in structure $\mathbf{A}$ over X, p. 20
$\mathcal{M}^{\mathbf{A}}_{X,W}$	The set of all extended state sequences in structure $\mathbf{A}$ over X and W, p. 20
$\boldsymbol{\mathcal{M}}_p$	The Kripke structure for the IMP program p, p. 90
$\mathcal{M}_p$	The set of all state sequences for the IMP program p, p. 46
$\boldsymbol{\mathcal{M}}_\theta$	The tableau for the LTL formula θ, p. 98
$\mathbf{MU}_{\boldsymbol{\mathcal{M}}}$	The set of μ-calculus formulae over the Kripke structure $\boldsymbol{\mathcal{M}}$, p. 95
M^W	The set of all extended state sequences in $\mathcal{M}^{\mathbf{A}}_{X,W}$ that extend state sequences from $M \subseteq \mathcal{M}^{\mathbf{A}}_X$, p. 21
$\mathcal{M}_W$	The set of all state sequences over W, p. 13
$\mathbb{N}$	Natural numbers $0, 1, 2, \ldots$, p. 8
$\mathcal{O}_{\mathbf{LTL}}$	The set of LTL operators, p. 10
$\mathcal{O}_{\mathbf{MU}}$	The set of operators for μ-calculus formulae, p. 95
$\mathfrak{P}(X)$	The power set of X, p. 8
$\mathbf{PC}(\pi)$	The temporal path condition for the information flow path π, p. 54
PDG_p	The program dependence graph for the IMP program p, p. 32
$\mathbf{PDOM}(w)$	The set of nodes that postdominate w in a CFG, p. 27
$v\,\mathbf{PDOM}\,w$	v postdominates w in a CFG, p. 26

$p_{f,i}$	State formula for function or predicate operator f that expresses a necessary condition over parameters to f other than the i-th one for f not being constant over $\mathbf{A}_{\mathsf{IMP}}$ except for undefined values in the i-th parameter, p. 53
$\Phi(x_0, x_1, \ldots, x_n)$	Φ constraint generated by the Φ function $x_0 := \Phi(x_1, \ldots, x_n)$, p. 35
${}^{v}\Phi_{x}^{w}$	Φ constraint for the flow (def-def) dependence edge $v \xrightarrow[x]{\mathsf{fd}\triangleright} w$ ($v \xrightarrow[x]{\mathsf{dd}\triangleright} w$), p. 36
π	Path in a graph G, p. 9
$\Pi(\pi)$	The set of all information flow paths that contain the influence path π, p. 64
$\Pi(s,t)$	The set of all information flow paths from s to t, p. 39
$\Pi^*(s,t)$	The set of all influence paths from s to t, p. 64
$\Pi^{**}(s,t)$	The set of all cycle-free information flow paths from s to t in PDG_p, p. 41
$\Pi_{\pi'}$	The set of all information flow paths from $\Pi(\pi')$ whose first and last edge is identical to those of the path π' which is a subpath of an influence path, p. 71
$\pi : v \xrightarrow[G]{}\!\!\!\blacktriangleright^{*} w$	Path in graph G from $v \in \mathrm{V}(G)$ to $w \in \mathrm{V}(G)$, p. 9
$\mathrm{pred}(e)$	Predecessor edge to e in a path π, p. 9
$\mathrm{pred}(v)$	Set of predecessor nodes $u \in V$ for v such that $(u,v) \in E$ in graph $G = (V,E)$, p. 8
$\mathbf{reuse}(v)$	The set of all IMP variable occurrences that are reused in the IMP statement v, p. 29
$\mathrm{sat}(\eta)$	The set of states of the tableau for the LTL formula θ in which the subformula or elementary formula η of θ holds, p. 98
$\mathcal{S}_X^{\mathbf{A}}$	The set of all states in structure $\mathbf{A}$ over X, p. 20
SF	The set of all state formulae over propositional variables, p. 10
$\mathsf{SF}^{\mathcal{L}}$	The set of all state formulae over the language $\mathcal{L}$, p. 19
$\mathsf{SF}_X^{\mathcal{L}}$	The set of all state formulae over $\mathcal{L}$ with typed variables from $X \subseteq \mathcal{V}$ and no propositional variables, p. 20
$\mathsf{SF}_{X,W}^{\mathcal{L}}$	The set of all state formulae over $\mathcal{L}$ with typed variables from $X \subseteq \mathcal{V}$ and propositional variables from $W \subseteq \mathcal{W}$, p. 20
$\mathrm{succ}(e)$	Successor edge to e in a path π, p. 9
$\mathrm{succ}(v)$	Set of successor nodes $u \in V$ to v such that $(v,u) \in E$ in graph $G = (V,E)$, p. 8
$\mathcal{S}_W$	The set of all states over $W \subseteq \mathcal{W}$, $\mathcal{S}_W = \mathbb{B}^W$, p. 13
$\mathcal{T}$	The set of types in a language, p. 18
TG_p	The transition graph for the IMP program p, p. 44

$\tau_{\mathcal{T}}$	Type function for the universe of a structure, p. 18
$\mathbf{use}(e)$	The set of all IMP variable occurrences that are read in the IMP expression e, p. 29
$\mathbf{use}(v)$	The set of all IMP variable occurrences that are read in the IMP statement v, p. 29
$\mathbb{V}$	The set of all variables in IMP, p. 24
$\mathcal{V}$	The countable set of typed variable identifiers, p. 18
$\mathcal{V}(e)$	Set of typed variables in expression e, p. 18
$\mathrm{V}(\pi)$	Set of nodes in path π, p. 9
v_e	The entry node in a control flow graph, p. 26
V_p	The set of program variables (identifiers) in the IMP program p, p. 45
$\mathbb{V}_p$	The set of SSA program variables in the IMP program p, p. 45
$\mathbb{V}_{\pi'}$	The set of all program variables from $\mathbb{V}_p$ accessed on a CFG path that corresponds to an information flow path from $\Pi_{\pi'}$, p. 71
$\mathcal{V}(\theta)$	The set of all typed variables that occur inside atomic formulae in the LTL formula θ, p. 19
v_x	The exit node in a control flow graph, p. 26
$\mathcal{W}$	The countable set of propositional variables for LTL formulae, p. 10
$\mathcal{W}(\theta)$	The set of propositional variables that occur in the LTL formula θ, p. 11
$W^+_{E'}$	Set of nodes that are reachable in the graph G from $W \subseteq \mathrm{V}(G)$ via edges in $E' \subseteq \mathrm{E}(G)$, p. 9
$\mathcal{X}$	The set of relational variables, p. 95
X^*	Set of all finite words or all tuples over X, p. 8
X^+	Set of all nonempty finite words or all nonempty tuples over X, p. 8
X/f	Partition on X induced by $f : X \mapsto Y$. $X/f = \{ f^{-1}(y) \mid y \in f(X) \}$, p. 8
Ξ	Usually used to denote a state sequence, p. 13
ξ	Usually used to denote a state, p. 13
$\xi(a)$	The value of the atomic formula $a \in \mathfrak{A}$ in state ξ, p. 20
$\xi(e)$	The value of expression $e \in \mathfrak{E}$ in state ξ, p. 20
X^ω	Set of all infinite words or infinite tuples over X, p. 8
$\mathbb{Z}$	Set of integers $\ldots, -2, -1, 0, 1, 2, \ldots$, p. 8
$\overline{\mathbb{Z}}$	Set of 32-bit integers $\{ -2^{31}, \ldots, 2^{31} - 1 \}$, p. 8

$\overline{\mathbb{Z}}^{\perp}$	The set $\overline{\mathbb{Z}} \cup \{ \perp_i \}$ of values of type `int` in the structure $\mathbf{A}_{\mathsf{IMP}}$, p. 24
$\mathcal{Z}_{\mathsf{LTL}}$	The alphabet for LTL formulae over propositional variables, p. 10
$\mathcal{Z}_{\mathsf{MU}}$	The alphabet for μ-calculus formulae, p. 95

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Passau, den 8. Mai 2006