HAL
open science

Verification and falsification of programs with loops using predicate abstraction

Daniel Kroening, Georg Weissenbacher

To cite this version:

Daniel Kroening, Georg Weissenbacher. Verification and falsification of programs with loops using predicate abstraction. Formal Aspects of Computing, 2009, 22 (2), pp.105-128. 10.1007/s00165-009-0110-2 . hal-00534924

HAL Id: hal-00534924

https://hal.science/hal-00534924v1

Submitted on 11 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOI 10.1007/s00165-009-0110-2
BCS © 2009
Formal Aspects of Computing (2010) 22: 105–128

Formal Aspects of Computing

Verification and falsification of programs with loops using predicate abstraction

Daniel Kroening[1] and Georg Weissenbacher[1,2]

[1] Computing Laboratory, Oxford University, Wolfson Building, Parks Road, Oxford OX1 3QD, UK. E-mail: georg@weissenbacher.name
[2] Computer Systems Institute, ETH Zurich, Zurich, Switzerland

Abstract. Predicate abstraction is a major abstraction technique for the verification of software. Data is abstracted by means of Boolean variables, which keep track of predicates over the data. In many cases, predicate abstraction suffers from the need for at least one predicate for each iteration of a loop construct in the program. We propose to extract *looping counterexamples* from the abstract model, and to parametrise the simulation instance in the number of loop iterations. We present a novel technique that speeds up the detection of long counterexamples as well as the verification of programs with loops.

1. Introduction

Software Model Checking [CGP99] provides automated support to discover flaws in computer programs. Despite its promise, software model checking is difficult to apply in practice, as the respective verification tools either lack scalability due to the state-space explosion problem, or trade precision for efficiency.

Abstraction techniques map the original, concrete set of states to a smaller set of states resulting in a (conservative) approximation of the system with respect to the property of interest [CC77]. Predicate abstraction is one of the most popular and widely applied methods for systematic state-space reduction of programs [GS97]. This technique is promoted by the success of the SLAM project [BR02b, BCLR04]. SLAM is used to show lightweight properties of Windows device drivers. Predicate abstraction enables SLAM to scale to large instances.

In predicate abstraction, a finite set of predicates keeps track of certain facts about the program variables. This set of predicates determines the precision of the abstraction. The resulting abstract model is an over-approximation of the original program and preserves all feasible paths of the original program. Unfortunately, given an insufficient set of predicates, the abstraction step introduces *spurious counterexamples*, leading to false alarms. In order to overcome this problem, predicate abstraction has been paired with counterexample-guided abstraction refinement (CEGAR) [CGJ+00, Kur95]. In CEGAR, spurious counterexamples are automatically analysed in order to obtain additional predicates that inhibit the undesired behaviour. The current refinement algorithms based on predicate transformers [BR02a] or interpolation [HJMM04] eliminate spurious counterexamples one by one, not taking control flow constructs like loops into account. This approach may result in an enumeration of infeasible paths. In the case of loops, the refinement algorithm often adds one predicate in order to extend the spurious counterexample by one additional iteration in each refinement cycle, resulting in increasingly longer, but still spurious, counterexamples. This process only terminates if there is a number of loop iterations that yields

Correspondence and offprint requests to: G. Weissenbacher, E-mail: georg@weissenbacher.name
Supported by Microsoft Research through its European PhD scholarship programme and by the EU FP7 STREP MOGENTES (project ID ICT-216679). This paper is an extension of [KW06]. The work was mainly carried out at ETH Zurich.

a feasible counterexample. If that is not the case, the refinement algorithm may generate a diverging sequence of predicates and the verification process fails to terminate [JM06].

The information about loop structures is actually available in the abstract model. Model checkers for the abstract model, however, never report paths with loops, as they aim at counterexamples that are as short as possible. We use this additional information to bypass the expensive, repetitive refinement steps that may occur when dealing with loops. We have to distinguish two cases: If there exists a path that traverses the loop and violates the specification, then we would like to compute the appropriate number of iterations that constitutes a feasible counterexample. If, on the other hand, no such counterexample exists, we would like to be able to deduce a set of predicates that eliminates a whole *class* of spurious counterexamples that traverse this loop.

Contribution and outline We proposed an approach to accelerate the detection of counterexamples with loops in [KW06]. After introducing the required preliminaries in Sect. 2, we provide a detailed discussion of this approach in Sect. 3. Based on the technique presented in Sect. 3, we propose a novel refinement algorithm that is (in some cases) able to remove a *family* of spurious counterexamples with a single refinement by constructing an invariant for each loop traversed by these paths (see Sect. 4). In Sect. 5, we illustrate the potential of our algorithm by means of several examples. In Sect. 6, we define the class of programs that our algorithm can prove to be safe. Section 7 provides an experimental evaluation of our technique based on a number of buffer overflow benchmarks.

2. Background

In this section, we introduce the formalism we use to present programs, and we provide a brief introduction to predicate abstraction (based on [Bal05]).

2.1. Programs and assertions

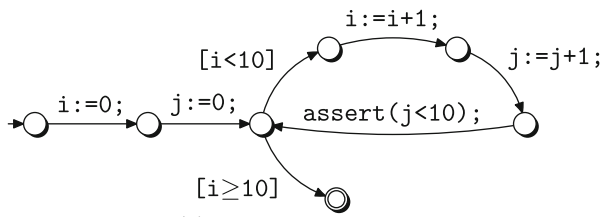
We use control flow automata (CFA) [HJM+02] to represent programs. A CFA comprises a finite number of nodes and a set of edges. Each CFA contains one designated entry node ➔◯, which has no predecessors, and one designated exit node ◎, which has no successors. Each edge connects two nodes and is annotated with an *instruction*. Let $\mathcal{C}$ be the domain of valuations to the program variables. An instruction is either a *test*, i.e., a total map from the concrete domain $\mathcal{C}$ to the Boolean domain $\mathbb{B}$, or an *assignment* mapping $\mathcal{C}$ into $\mathcal{C}$ [CC79]. Furthermore, since we are concerned with the safety of programs, we assume that programs are annotated with assertions. We conclude that a program is *safe* if none of the assertions can be violated. We use $[p]$ to denote tests, $\mathtt{x} := e$ to denote assignments, and $\mathtt{assert}(p)$ to represent assertions, where $\mathtt{x}$ is a program variable, e a quantifier-free expression, and p a quantifier-free Boolean predicate, both e and p being well-formed members of some quantifier-free first-order language $\mathcal{L}$. We consider only expressions and predicates that have a well-defined meaning in the context of the program.

Figure 1a shows an example of a CFA. The CFA determines the paths that may be traversed during the execution of the program. Each path corresponds to a sequence of instructions (see, for instance, the path in Fig. 1b).

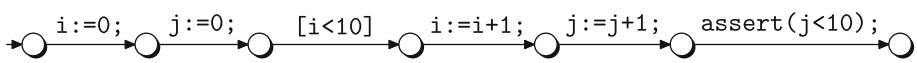
2.2. Semantics of programs

We define the semantics of programs by regarding instructions as binary relations on predicates. Each instruction $\mathtt{instr}$ relates state $s \in \mathcal{C}$ to outcome $s' \in \mathcal{C}$ if s' is a possible result of executing $\mathtt{instr}$ in state s. Thus, an instruction relates a set of *input* states $\mathcal{S} \subseteq \mathcal{C}$ to a set of (potential) successors $\mathcal{S}' \subseteq \mathcal{C}$. Since a failing test has no successor states, the relation corresponding to an instruction may be non-total. Each set of states $\mathcal{S}$ can be characterised using a predicate p which maps a state s to **true** (denoted by $p(s) = \textbf{true}$) iff $s \in \mathcal{S}$. We use the Hoare triple formalism of Floyd–Hoare logic [Flo67, Hoa69] to assign a formal meaning to each instruction.

Definition 1 (Hoare Triple) The Hoare Triple $\{p\}\ \mathtt{instr}\ \{q\}$ means that *for all* states s and s', if $\mathtt{instr}$ relates s to s', then $p(s)$ implies $q(s')$ [Nel89]. We refer to p as the *pre-condition* and to q as the *post-condition* of the Hoare Triple.



(a) A control flow automaton



(b) A path reaching the assertion

Fig. 1. A control flow automaton and a corresponding path

$$\frac{}{\{p[\mathtt{x}/e]\}\ \mathtt{x:=}e\ \{p\}}\ \text{assignment} \qquad \frac{}{\{p \Rightarrow q\}\ [p]\ \{q\}}\ \text{test}$$

$$\frac{\{p\}\ \pi_1\ \{q\},\ \{q\}\ \pi_2\ \{r\}}{\{p\}\ \pi_1;\pi_2\ \{r\}}\ \text{composition} \qquad \frac{p \Rightarrow q,\ \{q\}\ \pi\ \{r\},\ r \Rightarrow s}{\{p\}\ \pi\ \{s\}}\ \text{consequence}$$

Fig. 2. Hoare logic rules for loop-free paths of finite length

Figure 2 shows a set of rules that relate the predicates and the instructions accordingly. A brief explanation of these rules, which we use to reason about the safety of finite paths, is in order (for a more elaborate discussion we refer the reader to Nelson's excellent paper on Hoare's logic and Dijkstra's calculus [Nel89]).

1. *Assignments.* Let the term $p[\mathtt{x}/e]$ denote the predicate p with all occurrences of `x` replaced by the expression e. The assignment rule states that what the pre-condition says about the expression e holds for the variable `x` after the assignment. Given a post-condition p, the respective pre-condition for an assignment can be efficiently computed by means of substitution.
2. *Tests.* Tests, by virtue of being partial statements, require more consideration. A succeeding test $[p]$ discharges the guarding condition p in the pre-condition $p \Rightarrow q$ and establishes the post-condition q. In case of a failing test, however, the pre-condition $p \Rightarrow q$ evaluates to **true** irrespective of the post-condition q: if $p(s)$ is **false**, then $[p]$ does not relate s to any successor state s', and therefore, $q(s')$ holds vacuously (see Definition 1). In the case that $[p]$ does not have any successor states at all, any arbitrary post-condition holds. In particular, $\{\mathbf{true}\}\ [\mathbf{false}]\ \{p\}$ holds for any p (even for **false**).
   Dijkstra ruled out this oddity by introducing the law of the excluded miracle $\vdash \{\mathbf{false}\}\ \mathtt{instr}\ \{\mathbf{false}\}$, stating that no instruction can establish the post-condition **false** [Dij75]. Nelson considers this law too restrictive, since it effectively prevents partial instructions [Nel89].
   In our setting, we do allow partial instructions. Since, however, we are interested in the feasibility of unsafe paths, we use the law of the excluded miracle to check whether a test fails or succeeds. Given a test $[p]$ and the post-condition **false**, we obtain the pre-condition $p \Rightarrow \mathbf{false}$ using the test rule (Fig. 2). By the law of the excluded miracle, this pre-condition has to be **false** if $[p]$ is not supposed to terminate in "no state." It follows from setting ($p \Rightarrow \mathbf{false}$) (or equivalently, $\neg p$) to **false** that the instruction $[p]$ can be treated as a total relation if its pre-condition implies p. In that case, $\{p \wedge q\}\ [p]\ \{q\}$ holds.
3. *Composition.* The composition rule naturally extends the semantics for single instructions to finite execution paths (i.e., sequences of instructions). It propagates the post-condition q of the first instruction to the second instruction, for which q then acts as the pre-condition. Thus, we can compute Hoare triples for finite paths $\pi := \pi_1;\pi_2$ by iteratively applying the composition rule to sub-paths π_1 and π_2.
4. *Consequence.* The consequence rule states that pre-conditions of Hoare triples may be strengthened and post-conditions may be weakened. This rule enables the composition of sub-paths even if the post- and pre-conditions of two adjacent sub-paths do not match exactly. It will prove to be essential when we restrict the language $\mathcal{L}$ used to express the predicates (as explained in Sect. 2.3).

Notably, Fig. 2 does not list a rule for assertions. In our setting, no such rule is required. We are merely interested in assertions that do not hold when reached via a path π. Given a path π; `assert(`p`)`, an execution along that path cannot violate the assertion p if {**true**} π $\{p\}$ holds. Whenever we encounter an assertion `assert(`p`)` that is not violated, we treat it as a succeeding test `[`p`]`. As elaborated in case 2 above, $\vdash \{p \wedge q\}$ `assert(`p`)` $\{q\}$ holds in this setting.

Hoare triples for an instruction can be computed using predicate transformers like the *weakest pre-condition* and the *strongest post-condition*. The weakest pre-condition wp(`instr`, q) is the weakest predicate p (with respect to the implication ordering) that guarantees that q holds after the execution of `instr` [Dij75]. The strongest post-condition sp(`instr`, p) is the strongest fact q that is guaranteed to hold if p holds before the execution of `instr` [Gri87].

Example 1 We demonstrate the use of the rules in Fig. 2 by analysing the path in Fig. 1b. Our intention is to show that (`j` < 10) holds when the assertion is reached. We achieve this by performing a backwards analysis, starting with the predicate (`j` < 10). According to the assignment axiom, we obtain the pre-conditions for the last two assignment instructions in the path by replacing all occurrences of the assigned variables `i` and `j` in the respective post-conditions with `i`+1 and `j`+1, respectively. We derive

$$\{\mathtt{j} + 1 < 10\}\ \mathtt{j} := \mathtt{j} + 1\ \{\mathtt{j} < 10\}. \tag{1}$$

Since `i` does not occur in the predicate `j`+1 < 10, we obtain

$$\{\mathtt{j} + 1 < 10\}\ \mathtt{i} := \mathtt{i} + 1\ \{\mathtt{j} + 1 < 10\}. \tag{2}$$

By applying the composition rule to (1) and (2) we obtain

$$\{\mathtt{j} + 1 < 10\}\ \mathtt{i} := \mathtt{i} + 1;\ \ \mathtt{j} := \mathtt{j} + 1\ \{\mathtt{j} < 10\}. \tag{3}$$

According to the test axiom,

$$\{(\mathtt{i} < 10) \Rightarrow (\mathtt{j} + 1 < 10)\}\ [\mathtt{i} < 10]\ \{\mathtt{j} + 1 < 10\} \tag{4}$$

holds. As in the previous steps, deriving the pre-condition from the post-condition is a syntactic transformation. The path in Fig. 1b does not reach the assertion if the test `[i` < 10`]` fails. We restrict our analysis to the case in which the test succeeds, and strengthen (`i` $< 10) \Rightarrow$ (`j` $+ 1 < 10$) to (`i` $< 10) \wedge$ (`j` $+ 1 < 10$). Applying the assignment axiom to `j:=0`, we obtain (after simplifying the pre-condition)

$$\{\mathtt{i} < 10\}\ \mathtt{j} := 0\ \{(\mathtt{i} < 10) \wedge (\mathtt{j} + 1 < 10)\}. \tag{5}$$

Similarly, we use the assignment axiom to derive

$$\{\textbf{true}\}\ \mathtt{i} := 0\ \{\mathtt{i} < 10\}. \tag{6}$$

Combining (3) to (6) using the composition rule, we conclude

$$\{\textbf{true}\}\ \mathtt{i} := 0;\ \mathtt{j} := 0;\ [\mathtt{i} < 10];\ \mathtt{i} := \mathtt{i} + 1;\ \mathtt{j} := \mathtt{j} + 1\ \{\mathtt{j} < 10\},$$

i.e., the path cannot violate the assertion (`j` < 10).

A program is safe if and only if it contains no path that violates an assertion. Since control flow automata may contain cycles (i.e., loops), a program may comprise infinitely many paths. The question whether a program is safe or not is undecidable in general.

2.3. Predicate abstraction and refinement

Predicate abstraction [GS97] is a technique that restricts the pre- and post-conditions that are tracked at each location of a path to Boolean combinations of a finite set P of predetermined predicates (we defer the discussion of how we obtain this set of predicates P to the end of Sect. 2.3). There are only finitely many logically non-equivalent propositional combinations of the predicates P. This restriction limits the expressiveness of the language $\mathcal{L}$ used to represent the predicates of the Hoare triples. For instance, there is no Boolean combination of the predicates (`i` $= 0$) and (`j` $= 0$) that allows us to express (`j` < 10). Under this restriction, it may be impossible to prove the safety of a path, even if the path does not violate an assertion.

Example 2 We attempt to prove the safety of the path in Fig. 1b using only Boolean combinations of the predicates $(\mathtt{i} = 0)$ and $(\mathtt{j} = 0)$ to keep track of the pre- and post-conditions at each location of the path. We already noticed that $(\mathtt{j} < 10)$ cannot be expressed in terms of $(\mathtt{i} = 0)$ and $(\mathtt{j} = 0)$. Unless we can establish that $(\mathtt{j} = 0)$ holds at the end of the path, we cannot guarantee the safety of the path. A brief informal inspection of the path shows us that this attempt must fail.

In the remaining part of the example, we show how Hoare logic can be used to reason over a restricted set of predicates P. As previously mentioned, the consequence rule plays an important role in this setting. At any point in the path, we try to find the strongest predicate that holds.

Since we do not make any assumptions about the initial state of the program, **true** is the strongest assertion that holds at the beginning of the path. By the assignment axiom

$$\{\mathbf{true}\}\ \mathtt{i} := 0\ \{\mathtt{i} = 0\}. \tag{7}$$

By simply considering all possible (logically non-equivalent) propositional combinations of the predicates $(\mathtt{i} = 0)$ and $(\mathtt{j} = 0)$, we can see that it is impossible to derive a post-condition stronger than $(\mathtt{i} = 0)$. Using this result as a pre-condition for the next instruction, we obtain

$$\{\mathtt{i} = 0\}\ \mathtt{j} := 0\ \{(\mathtt{j} = 0) \wedge (\mathtt{i} = 0)\} \tag{8}$$

by a similar argument.

The condition of the test $[\mathtt{i} < 10]$ following the instruction $\mathtt{j} := 0$ introduces a predicate that cannot be expressed as a Boolean combination of $(\mathtt{i} = 0)$ and $(\mathtt{j} = 0)$. According to the test rule

$$\{(\mathtt{i} < 10) \Rightarrow (\mathtt{i} = 0) \wedge (\mathtt{j} = 0)\}\ [\mathtt{i} < 10]\ \{(\mathtt{i} = 0) \wedge (\mathtt{j} = 0)\} \tag{9}$$

holds. The pre-condition of (9) contains the predicate $\mathtt{i} < 10$, which is not an element of P. The consequence rule, however, allows us to strengthen the pre-condition, since it is implied by $(\mathtt{i} = 0) \wedge (\mathtt{j} = 0)$:

$$\{(\mathtt{i} = 0) \wedge (\mathtt{j} = 0)\}\ [\mathtt{i} < 10]\ \{(\mathtt{i} = 0) \wedge (\mathtt{j} = 0)\} \tag{10}$$

In general, strengthening requires first-order logic reasoning and an exhaustive examination of the propositional combinations of P in order to find the best approximation of a pre-condition not expressible in terms of a Boolean formula over the atoms P.

Using the assignment axiom, we obtain

$$\{(\mathtt{i} = 0) \wedge (\mathtt{j} = 0)\}\ \mathtt{i} := \mathtt{i} + 1\ \{(\mathtt{i} = 1) \wedge (\mathtt{j} = 0)\}. \tag{11}$$

Again, the predicate $(\mathtt{i} = 1)$ is not in P, nor can it be expressed as a propositional formula over the predicates in P. The post-condition $(\mathtt{i} = 1) \wedge (\mathtt{j} = 0)$ implies $\neg(\mathtt{i} = 0) \wedge (\mathtt{j} = 0)$, though, and therefore

$$\{(\mathtt{i} = 0) \wedge (\mathtt{j} = 0)\}\ \mathtt{i} := \mathtt{i} + 1\ \{\neg(\mathtt{i} = 0) \wedge (\mathtt{j} = 0)\} \tag{12}$$

holds according to the consequence rule. Analogously, we derive

$$\{\neg(\mathtt{i} = 0) \wedge (\mathtt{j} = 0)\}\ \mathtt{j} := \mathtt{j} + 1\ \{\neg(\mathtt{i} = 0) \wedge \neg(\mathtt{j} = 0)\} \tag{13}$$

using the assignment axiom and the consequence rule. For each of the instructions considered so far, the pre- and post-conditions we derived are Boolean combinations of the given set of predicates. Using the composition rule, we combine (7), (8), (10), (12), and (13):

$$\{\mathbf{true}\}\ \mathtt{i} := 0;\ \ \mathtt{j} := 0;\ \ [\mathtt{i} < 10];\ \ \mathtt{i} := \mathtt{i} + 1;\ \ \mathtt{j} := \mathtt{j} + 1\ \{\neg(\mathtt{i} = 0) \wedge \neg(\mathtt{j} = 0)\}. \tag{14}$$

The choice of the propositional combinations of the predicates in P to express the pre- and post-conditions in this example may seem ad hoc. A more systematic and formal treatment of this issue follows after the example (see, in particular, Definition 2).

As mentioned above, the post-condition of (14) is not strong enough to imply the safety of the path. The sequence of instructions in the Hoare triple (14), however, does *not* result in a violation of the assertion, i.e., the path is safe. A safe path that cannot be proved safe using the given set of predicates P is called a *spurious counterexample*.

Each pre- or post-condition corresponds to a set of valuations of the predicates in P. Given a set $P = \{p_1, \ldots, p_n\}$ of n predicates, we introduce n Boolean variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$, each of which corresponds to one of the predicates. The states induced by these Boolean variables define a finite abstract domain [GS97, CC77]. Each abstract state $s \in \mathbb{B}^n$ is a valuation of the variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$ and corresponds to a condition $\mathcal{C}_P(s)$:

$$\mathcal{C}_P(s) := \bigwedge_{p_i \in P} \begin{cases} p_i & \text{if } \mathtt{x}_i = \textbf{true} \text{ in state } s \\ \neg p_i & \text{if } \mathtt{x}_i = \textbf{false} \text{ in state } s \end{cases}$$

Each predicate q that is a Boolean combination of the predicates in P maps to the set of abstract states $\{s \mid \mathcal{C}_P(s) \Rightarrow q\}$. The post-condition of (14), for instance, constrains the negations of $(\mathtt{i} = 0)$ and $(\mathtt{j} = 0)$, and therefore maps to $\{(x_1 = \textbf{false}, x_2 = \textbf{false})\}$. The pre-condition $\{\textbf{true}\}$ allows these predicates (and therefore the corresponding Boolean variables) to take arbitrary Boolean values, and therefore maps to $\{(x_1 = \textbf{false}, x_2 = \textbf{false}), (x_1 = \textbf{false}, x_2 = \textbf{true}), (x_1 = \textbf{true}, x_2 = \textbf{false}), (x_1 = \textbf{true}, x_2 = \textbf{true})\}$.

In the context of the original program, the predicate q characterises the set of (not necessarily reachable) concrete states in which q evaluates to **true**. The post-condition $\neg((\mathtt{i} = 0) \wedge (\mathtt{j} = 0))$, for instance, denotes the set of states $\{(\mathtt{i}, \mathtt{j}) \mid \mathtt{i} \neq 0 \vee \mathtt{j} \neq 0\}$ in which $\mathtt{i}$ or $\mathtt{j}$ differs from zero. Thus, the set of concrete states that correspond to an abstract state s is defined by the predicate $\mathcal{C}_P(s)$.

Given the abstract domain induced by the Boolean variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$, *existential abstraction* [CGL92] is a technique to compute a corresponding abstract transition relation $R_{\text{instr}} \subseteq \mathbb{B}^n \times \mathbb{B}^n$.

Definition 2 (Existential abstraction) The existential abstraction of an instruction `instr` with respect to a set P of n predicates is a total relation $R_{\text{instr}} \subseteq \mathbb{B}^n \times \mathbb{B}^n$ defined as

$$R_{\text{instr}}(s_1, s_2) := \begin{cases} \textbf{false} & \text{if } \{\mathcal{C}_P(s_1)\}\ \mathtt{instr}\ \{\neg\mathcal{C}_P(s_2)\} \text{ holds,} \\ \textbf{true} & \text{otherwise.} \end{cases}$$

Intuitively, whenever $\mathcal{C}_P(s_1)$ contains a concrete state from which a concrete state in $\mathcal{C}_P(s_2)$ can be reached by executing `instr`, then the abstract transition relation R_{instr} enables the transition between the corresponding abstract states s_1 and s_2.

By means of existential abstraction, we obtain an abstract finite state transition system that reflects all Hoare triples $\{p\}\ \pi\ \{q\}$ that cannot be ruled out using only Boolean combinations of the predicates P. Note that if $\{p\}\ \mathtt{instr}\ \{\neg q\}$ holds, then the concrete states represented by q cannot be reached from the set of concrete states that correspond to p via `instr`. Given the original CFA, we replace each instruction `instr` with a corresponding abstract transition relation $R_{\text{instr}} \subseteq \mathbb{B}^n \times \mathbb{B}^n$ over the state space induced by the variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$.

This approach preserves all paths that violate safety properties: No path that violates an assertion can be proved safe. Safe paths, however, may be reported unsafe, as shown in Example 2. Existential abstraction is computationally expensive, since all combinations of the states s_1 and s_2 have to be considered, i.e., in the worst case, computing the exact relation R_{instr} requires an exhaustive examination of all pairs of logically non-equivalent propositional combinations of the predicates P, as indicated in Example 2.

In practice, more efficient (but less accurate) abstraction techniques like Cartesian abstraction [BPR01] or symbolic techniques [KS06] are applied. In the former approach, the relation R_{instr} is computed for each predicate of P separately: Given a predicate $q \in P$, the respective pre-condition p for q with respect to `instr` is constructed using the rules in Fig. 2 (such that $\{p\}\ \mathtt{instr}\ \{q\}$ holds). If p is not expressible as a Boolean combination of the predicates P, a heuristic is used to compute a propositional formula p' over P such that $\{p'\}\ \mathtt{instr}\ \{q\}$ holds. The same technique is used to obtain an approximation p'' of the pre-condition of $\neg q$ with respect to `instr`. The

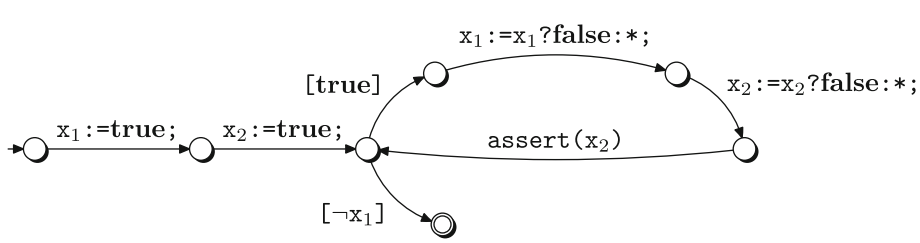


Fig. 3. Abstract transition system for the program in Fig. 1a. The Boolean variables x_1 and x_2 correspond to the predicate $(\mathtt{i} = 0)$ and $(\mathtt{j} = 0)$, respectively

abstract transition relation $R_{\text{instr}}(s_1, s_2)$ is constructed such that the variable x that corresponds to p is **true** in s_2 if $\mathcal{C}_P(s_1)$ implies p', **false** if $\mathcal{C}_P(s_1)$ implies p''. Otherwise, the value of x in s_2 is arbitrary.

The abstraction step preserves the control flow structure of the program. Figure 3 shows the finite state transition system that corresponds to the program in Fig. 1a. The variables $\mathtt{x}_1$ and $\mathtt{x}_2$ correspond to the predicates $(\mathtt{i} = 0)$ and $(\mathtt{j} = 0)$, respectively. We represent the abstract transition relation using a C-like syntax. The assignment $\mathtt{x}_1 := \mathtt{x}_1 ?\textbf{false} : *$ should be read as follows: If $\mathtt{x}_1$ is **true** before the assignment, then it must be false afterwards. Otherwise, if $\mathtt{x}_1$ is **false**, it is assigned *non-deterministically* (indicated by $*$).

The former case derives from the fact that

$$\{\neg(\mathtt{i} + 1 = 0)\}\ \mathtt{i} := \mathtt{i} + 1\ \{\neg(\mathtt{i} = 0)\} \quad \text{and therefore} \quad \{(\mathtt{i} = 0)\}\ \mathtt{i} := \mathtt{i} + 1\ \{\neg(\mathtt{i} = 0)\}$$

holds. On the other hand, we derive

$$\{(\mathtt{i} + 1 = 0)\}\ \mathtt{i} := \mathtt{i} + 1\ \{(\mathtt{i} = 0)\}$$

using the assignment rule, but the pre-condition $(\mathtt{i} + 1 = 0)$ is not implied by either $(\mathtt{i} = 0)$ or its negation. The resulting loss of information is represented by the non-deterministic assignment of $\mathtt{x}_1$.

The assumption $[\mathtt{i} < 10]$ is approximated by $[\textbf{true}]$, since neither the pre-condition $(\mathtt{i} = 0)$ nor $(\mathtt{i} \neq 0)$ rules out the transition. The condition of the assertion $(\mathtt{j} < 10)$ is replaced by the weakest combination of predicates that guarantees that it holds, which is $(\mathtt{j} = 0)$ in our case.

Based on R_{instr}, we define the reachability relation for a path π (where $\pi = \mathtt{instr}_1;\ \ldots;\ \mathtt{instr}_{n-1}$):

$$R_\pi(s_1, s_n) := \bigwedge_{i=1}^{n-1} R_{\text{instr}_i}(s_i, s_{i+1}) \tag{15}$$

This relation denotes whether the abstract state s_n is reachable from s_1 via the path π. To prove the safety of a program, we have to show that the abstract transition system allows *no* path π for which all of the following conditions hold:

- π starts at ➔○,
- the instruction following the last instruction of π is $\mathtt{assert}(r)$, and
- there exists an initial state s_1 such that $R_\pi(s_1, s_n)$ holds and $\mathcal{C}_P(s_n)$ does not imply r.

There are several efficient model checking tools [BR00, EHRS00, CKS05, McM92] that are able to exhaustively examine finite state transition systems for paths that reach a given unsafe state or location. If there are paths reaching the unsafe state in question (i.e., an abstract state that corresponds to a potential violation of an assertion), these tools report the *shortest* of these paths as a *counterexample*. As explained in Example 2, this counterexample may be spurious, since the abstraction *over-approximates* the set of feasible execution traces of the original program. An alternative view is that the finite state transition system represents an under-approximation of the facts that can be proved about the original program (i.e., there may be paths that are not feasible in the original program, but cannot be ruled out in the abstraction).

The spurious counterexample π can be eliminated from the abstract transition system by means of counterexample-guided abstraction refinement [CGJ+00]. Consider the Hoare logic proof in Example 1, which establishes the safety of the spurious counterexample π of Example 2. Adding the pre- and post-conditions that occur in this proof to P is sufficient to eliminate π from the abstract transition system [BR02a]. For instance, if we are allowed to use the predicate $(\mathtt{j} < 10)$, we can strengthen the Hoare triple (13) in Example 2 to

$$\{\neg(\mathtt{i} = 0) \wedge (\mathtt{j} = 0)\}\ \mathtt{j} := \mathtt{j} + 1\ \{\neg(\mathtt{i} = 0) \wedge \neg(\mathtt{j} = 0) \wedge (\mathtt{j} < 10)\},$$

thus establishing the safety of the spurious counterexample. The corresponding refined abstract transition system is shown in Fig. 5, where $\mathtt{x}_3$ corresponds to the predicate $(\mathtt{j} < 10)$. Note that if we treated j as an unbounded integer variable, the abstraction of `j:=j+1` in Fig. 5 could be strengthened by considering the value of $\mathtt{x}_3$. If we take arithmetic overflow on bit-vector variables into account, though, it is as accurate as possible.

We summarise the verification technique presented in this chapter in Fig. 4. Starting with an empty set of predicates $P = \emptyset$, the algorithm uses predicate abstraction to compute a coarse abstraction (ABSTRACT). The resulting finite-state transition system is analysed by a model checking tool (MODELCHECK). If the abstraction is safe, we can conclude that the original program is safe, too. Otherwise, the model checking tool provides an abstract counterexample. The corresponding path is mapped back into the original program, where its feasibility is analysed by means of Hoare logic (SIMULATE). Genuine counterexamples are reported. If the counterexample

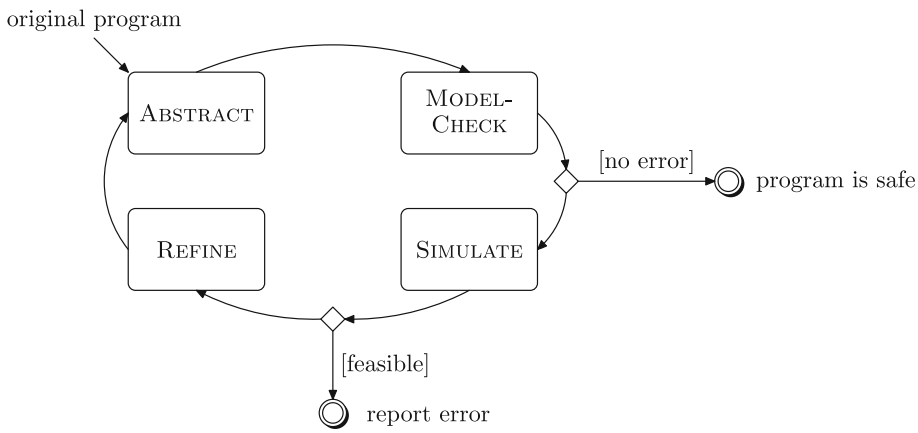


Fig. 4. Counterexample-guided Abstraction Refinement (CEGAR)

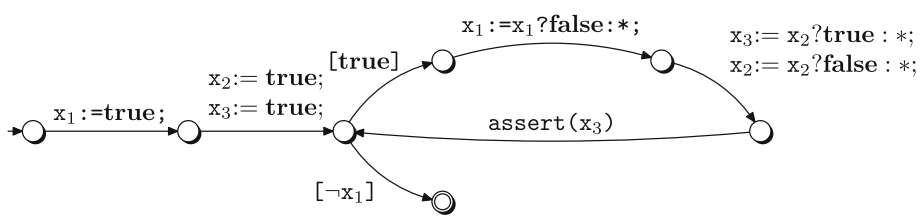


Fig. 5. A refined version of the transition system in Fig. 3. The variable x_3 corresponds to the predicate ($\mathtt{j} < 10$)

turns out to be spurious, the predicates encountered in the simulation step are added to P (Refine), and the abstraction-refinement cycle is repeated.

3. Counterexamples with loops

The technique presented in Sect. 2 does not work very well if the assertion that is violated can only be reached via a path with a large number of loop iterations. We demonstrate and discuss this problem in Sect. 3.1. In Sects. 3.2 and 3.3 we describe a heuristic to accelerate the detection of counterexamples that contain many loop iterations [KW06].

3.1. How predicate abstraction handles loops

In order to detect a counterexample that contains several loop iterations, predicate abstraction may require at least one predicate for each iteration of the loop.[1] Consider the abstract transition system in Fig. 5, which rules out the spurious counterexample of Example 2. Unfortunately, this refined transition system does not exclude the path that iterates the instructions of the loop in the control flow automaton *twice*. Figure 6 shows the corresponding counterexample.

Again, adding a refinement predicate (namely $\mathtt{j} + 1 < 10$) to P eliminates the spurious counterexample that iterates the loop twice. This predicate, however, fails to eliminate the spurious counterexample that executes the loop three times. In order to find the only unsafe path, which iterates the loop ten times, it is necessary to add all predicates from ($\mathtt{j} + 1 < 10$) up to ($\mathtt{j} + 9 < 10$). To achieve this, at least ten refinement steps are required.

[1] Technically, $\log_2(n)$ predicates are sufficient to enforce n iterations through a loop in the abstract program. However, we are not aware of any predicate abstraction-based tool that generates a binary encoding of the loop counter.

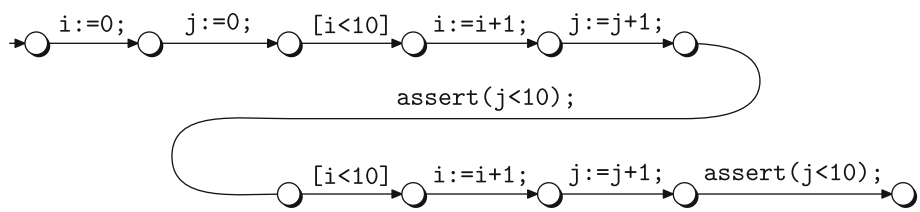


Fig. 6. A concrete path with two loop iterations

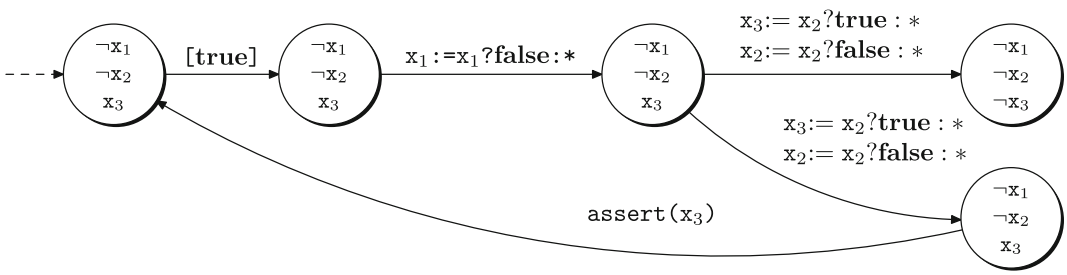


Fig. 7. Detecting potential loops in abstract paths

3.2. Detecting loops in abstract counterexamples

The abstract transition system in Fig. 5 contains not only a single counterexample, but a *family* of similar counterexamples:

Example 3 Consider the path π in Fig. 6. We examine the abstract states of the corresponding abstract path, which we obtain by mapping π to a path of the abstract transition system in Fig. 5. Starting with an arbitrary state, we reach an abstract state $x_1 = \textbf{true}$, $x_2 = \textbf{true}$, and $x_3 = \textbf{true}$ after two transitions. The first iteration of the loop changes this state to $x_1 = \textbf{false}$, $x_2 = \textbf{false}$, and $x_3 = \textbf{true}$ (the first state in Fig. 7). Once we reach the transition $x_3 := x_2?\textbf{true} : *;\ x_2 := x_2?\textbf{false} : *$, the non-deterministic transition function allows us to make a choice: Either we change x_3 to **false**, which results in a violation of the assertion in the subsequent transition, or we do not change x_3 and iterate the loop once more (see Fig. 7). Alternatively, the program may terminate without violating the assertion. The transition system in Fig. 5 allows to iterate the loop arbitrarily often before the assertion is finally violated.

This suggests that there is a potential loop in the original program (as indicated by the repetition signs ‖: and :‖ in Fig. 8). The model checking tool, though, reports only a finite path π, but does not provide information on potential loops that are traversed by this path.

The missing information can be added using a post-processing step: The algorithm presented in Fig. 9 searches for loops in abstract counterexamples. A loop has to contain a back-edge that allows us to jump back to an earlier location in the path π. We construct a propositional formula (in step ②) that enables us to efficiently search for back-edges R_{instr} in the abstract transition system. Intuitively, step ② corresponds to a model checking run that checks for each location i in π whether there is a path that leads back to i visiting only locations occurring in π. An example for such a back-edge is the transition labelled `assert(x3)` in Fig. 7 (as explained in Example 3).

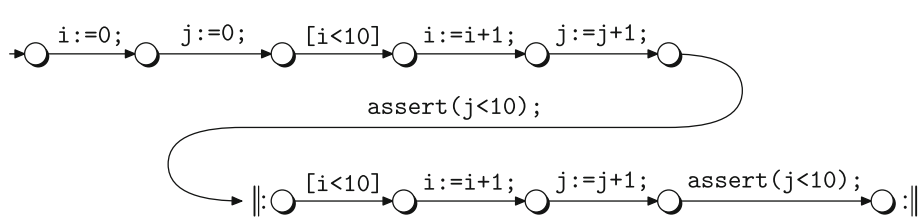


Fig. 8. The path from Fig. 6 annotated with loop information

FINDLOOPS(π) where $\pi = \texttt{instr}_1; \ldots; \texttt{instr}_{n-1};$

① The model checking tool determines a loop-free abstract counterexample $s_1, \ldots, s_n$ using symbolic simulation:

$$\mathcal{S}(s_1, \ldots, s_n) := \bigwedge_{k=1}^{n-1} R_{\text{instr}_k}(s_k, s_{k+1})$$

② For all pairs (i, j), where $j \in \{2, \ldots, n\}$ and $i < j$, check whether there is an instruction `instr` that follows $\texttt{instr}_{j-1}$ in the CFA such that

$$\exists s_1, \ldots, s_n. \mathcal{S}(s_1, \ldots, s_n) \wedge$$
$$\exists s'_i, \ldots, s'_j. R_{\text{instr}_{i-1}}(s_{i-1}, s'_i) \wedge \bigwedge_{k=i}^{j-1} R_{\text{instr}_k}(s'_k, s'_{k+1}) \wedge R_{\text{instr}}(s'_j, s'_i)$$

holds.

③ For each (i, j) for which the formula in step ② holds, add following annotation to the path π:

$$\|: \texttt{instr}_i, \ldots, \texttt{instr}_j :\|$$

Fig. 9. Algorithm to detect loops in abstract counterexamples

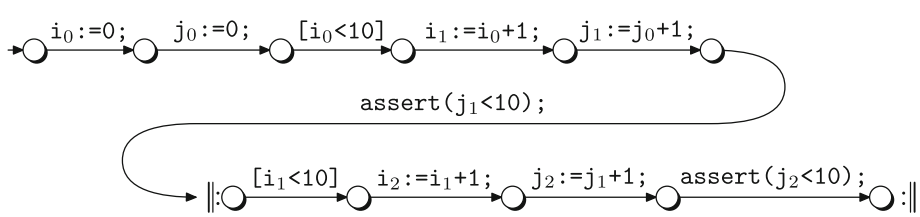


Fig. 10. The path from Fig. 8 converted to Single Static Assignment form

This idea is formalised in Fig. 9. The conjunction of abstract transition relations in step ① and ② can be encoded as a propositional formula, for which satisfiability (SAT) is usually efficiently decidable [ES04]. The formula $\mathcal{S}(s_1, \ldots, s_n)$ is a symbolic representation of the abstract counterexample π. The algorithm tries to find back-edges in the *abstract* transition system for each sub-path $\texttt{instr}_i; \ldots, \texttt{instr}_j$ of π, resulting in a quadratic number of SAT-instances. Our experiments show that the overhead for detecting loops is negligible compared to the time the model checking tool spends searching for counterexamples. Note that the algorithm is also able to detect nested loop structures.

3.3. Checking the safety of counterexamples with loops

The existence one or more loops in the abstract counterexample does not imply that there is a corresponding path that violates the assertion at the end of the counterexample π. The question whether a counterexample with loops is safe is undecidable in general. It is, however, possible to obtain a loop-free instance of the annotated counterexample by *unrolling* the loops a certain number of times. Using forward symbolic simulation (which is equivalent to computing the strongest post-condition of the path), we can then determine whether this instance violates the assertion or not (as outlined in Example 1). Consider, for instance, the counterexample π_9 obtained by unrolling the loop in Fig. 6 nine times. Let π be the prefix of π_9 that does not contain the last assertion of π_9. Then, we can show that $\{\textbf{true}\}\ \pi\ \{\texttt{j} \geq 10\}$ holds, i.e., the path π_9 is not safe.

In general, we do not know how many times we need to unroll the loops to obtain an unsafe path (it might even be that there is no such path). Therefore, we use a heuristic to find promising candidates.

The first step is to convert the counterexample π into static single assignment form (SSA) [CFR+91]. The SSA form is a representation in which each variable of a program is assigned exactly once. For this purpose, we replace

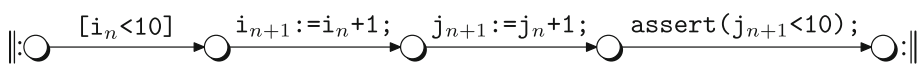


Fig. 11. A pattern derived by unrolling the loop in Fig. 10

$$F_\pi(n) \equiv$$
$$(\mathtt{i}_0 = 0) \wedge (\mathtt{j}_0 = 0) \wedge (\mathtt{i}_0 < 10) \wedge (\mathtt{i}_1 = \mathtt{i}_0 + 1) \wedge (\mathtt{j}_1 = \mathtt{j}_0 + 1) \wedge (\mathtt{j}_1 < 10) \wedge$$
$$(\mathtt{i}_1 < 10) \wedge (\mathtt{i}_2 = \mathtt{i}_1 + n) \wedge (\mathtt{j}_2 = \mathtt{j}_1 + n) \wedge (\mathtt{j}_2 \geq 10)$$

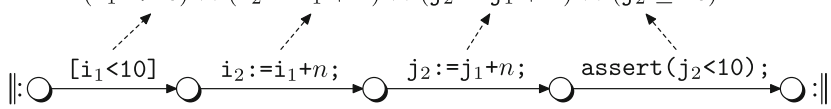


Fig. 12. Constraints derived from the parametrised loop

each existing variable occurrence by an indexed version of this variable (see Fig. 10, for example). Whenever we reach an assignment, the version number is increased by one.[2]

As we unroll the loop, a pattern emerges: In each iteration, the variables $\mathtt{i}_n$ and $\mathtt{j}_n$ depend on $\mathtt{i}_{n-1}$ and $\mathtt{j}_{n-1}$, respectively (see Fig. 11). Since the counterexample is in SSA form, it is easy to identify variants of the loop by means of a simple syntactic analysis. We obtain a *recurrence equation* for each variable that is changed in the loop:

$$\mathtt{i}_0 = 0, \quad \mathtt{i}_1 = \mathtt{i}_0 + 1 \qquad \mathtt{i}_n = \mathtt{i}_{n-1} + 1 \tag{16}$$
$$\mathtt{j}_0 = 0, \quad \mathtt{j}_1 = \mathtt{j}_0 + 1 \qquad \mathtt{j}_n = \mathtt{j}_{n-1} + 1 \tag{17}$$

We proceed by computing the *closed form* of these recurrence equations. Computing the closed form of an arbitrary recurrence equation is a non-trivial problem. In fact, in some cases such a closed form may not even exist [vEBG04]. In many real-world programs, however, the recurrence equations that occur in a loop are relatively simple. In our implementation, we consider only recurrence equations of the form

$$\mathtt{i}_0 = \alpha, \quad \mathtt{i}_n = \mathtt{i}_{n-1} + \beta + \gamma \cdot n$$

(where $n > 0$ and α, β, and γ are numeric constants or loop-invariant symbolic expressions and $\mathtt{i}$ is the variant). According to [GKP89], the corresponding closed form is

$$\mathtt{i}_n = \alpha + \beta n + \gamma \frac{n \cdot (n+1)}{2}.$$

It follows that the closed form of the recurrence equations (16) and (17) is $\mathtt{i}_n = n$ and $\mathtt{j}_n = n$, respectively. This translates to $\mathtt{i}_{n+1} = \mathtt{i}_1 + n$ and $\mathtt{j}_{n+1} = \mathtt{j}_1 + n$ in our example (see Figs. 10 and 11). The variable n corresponds to the number of loop iterations. If we replace the right-hand sides of the assignments in the loop in Fig. 10 by their corresponding closed forms, we obtain a counterexample that is *parametrised* with the number of loop iterations n. The instructions of the parametrised counterexample constrain the indexed variables that occur in the path as well as the variable n. Figure 12 shows the parametrised loop and the formula $F_\pi(n)$ derived from the counterexample. Note that the condition contributed by the last assertion is negated. Using a constraint solver, we try to compute the smallest value of n that occurs in a satisfying assignment of the formula $F_\pi(n)$.

The smallest value of n that is part of a satisfying assignment to the formula in Fig. 12 is 9. This value is an educated guess for the number of iterations necessary to violate the assertion. We unwind the loop of the counterexample according to the pattern in Fig. 11 such that the last assignment is $\mathtt{j}_{10} := 10$ and the last assertion is $\mathtt{assert}(\mathtt{j}_{10} < 10)$. As explained above, there exists a Hoare logic proof that this path is not safe.

Figure 13 shows the algorithm we use to compute candidates for the number of iterations of the loops in a counterexample. Note that the algorithm is able to handle more than one loop, and even nested loops. If the algorithm encounters a recurrence equation for which it fails to compute its closed form,[3] then the corresponding

[2] The result of this process is similar to what we obtain by computing the strongest post-condition and eliminating the existential quantifiers using Skolemisation.

[3] While it would certainly be feasible to support a larger class of recurrence equations (see for instance [vEBG04]), it turns out that our approach is sufficient to cover the most common cases like linear counters. We do not need to support cases in which the loop counter increases exponentially: These cases can be handled efficiently by traditional unwinding if the bounded range of the program variables is taken into account.

GuessIterations(π), where π is annotated with loops

① Transform π into SSA.

② For each loop in π (as reported by FindLoops):

❶ Generate a recurrence equation for each variable that is updated in the loop.

❷ Introduce a fresh variable n for the loop and try to calculate the closed forms for the recurrence equations. If unable to compute a closed form, leave the corresponding variable unconstrained (i.e., assign it non-deterministically).

$$\left.\begin{array}{lcl} \mathtt{i}_0 & = & \alpha \\ \mathtt{i}_n & = & \mathtt{i}_{n-1} + \beta + \gamma \cdot n \end{array}\right\} \longrightarrow \mathtt{i}_n = \alpha + \beta \cdot n + \gamma \frac{n \cdot (n+1)}{2}$$

$$\text{otherwise} \longrightarrow \mathtt{i}_n = *$$

❸ Substitute the right-hand sides of the assignments in π with the corresponding closed forms.

③ Generate the constraints for the parametrised path. Search for a satisfying assignment for the corresponding formula that minimises the parameters $\{n_1, n_2, \ldots\}$ introduced in step ②. If such an assignment exists, return the parameter values. Otherwise, return "unsatisfiable".

Fig. 13. Computing the number of iterations for a counterexample with loops

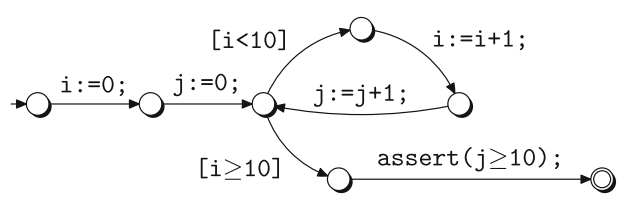


Fig. 14. A safe program

variable is assigned non-deterministically, i.e., the assignment does not contribute a constraint to the formula. Even though this may result in wrong guesses for the number of iterations, the soundness of the approach is guaranteed by checking the safety of the unwound counterexample by means of forward symbolic simulation.

In general, the parametrised formula is not necessarily satisfiable. If the formula $F_\pi(n)$ is unsatisfiable, then *all* counterexamples that can be obtained by unwinding the loop of π are spurious. On the other hand, the satisfiability of the formula $F_\pi(n)$ does not imply that unwinding the loop gives us a counterexample that violates the assertion. In Sect. 4, we discuss how to refine the abstract transition system if our heuristic fails.

4. Refinement in the presence of loops

The heuristic presented in Sect. 3.3 may fail to determine the number of iterations necessary to obtain an unsafe path. We distinguish two causes of failure: Either there is no valuation to the parameters $\{n_1, n_2, \ldots\}$ that satisfies the constraints of the path, or the heuristic suggests values for the parameters that result in an unwound counterexample that does not violate the assertion. We discuss the former cause in Sect. 4.1, and the latter cause in Sect. 4.2.

4.1. Refinement using closed recurrence equations

Figure 14 shows a safe program. The traditional predicate abstraction approach described in Sect. 2 requires ten refinement steps to introduce all predicates necessary to show the safety of the program. After two abstraction refinement cycles, the FindLoops algorithm (Fig. 9) detects a loop in the abstract transition relation. The suffix of the corresponding annotated concrete path is shown in Fig. 15.

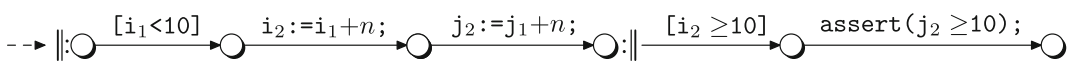


Fig. 15. A loop detected in the abstraction of the program in Fig. 14

$$\frac{\{p \wedge q\}\ \pi_1\ \{p\}}{\{p\}\ \|\!:\ [q]\,;\ \pi_1\ :\!\|\ [\neg q]\ \{\neg q \wedge p\}}\ \text{loop}$$

Fig. 16. Hoare logic rule for loops

GUESSITERATIONS in Fig. 13 fails to compute a valuation for the parameter n, since the formula

$$\ldots \wedge (\mathtt{i}_1 < 10) \wedge (\mathtt{i}_2 = n) \wedge (\mathtt{j}_2 = n) \wedge (\mathtt{i}_2 \geq 10) \wedge (\mathtt{j}_2 < 10) \tag{18}$$

is unsatisfiable. The formula derived from the parametrised path may have more satisfying assignments than the formula corresponding to the loop-free path. This stems from the potentially introduced non-determinism and from the fact that it encodes an arbitrary number of iterations of the loop. Therefore, its unsatisfiability implies the infeasibility of the original counterexample. Therefore, it is of course possible to fall back to the traditional refinement approach and use a proof of safety (generated using the rules in Fig. 2, as demonstrated in Example 1) for the counterexample *without loops* to refine the abstract transition relation. This approach, however, does not exploit the knowledge we have about the loop in the abstract program. Instead, we take advantage of this information by using a Hoare logic rule that allows us to reason about loops: Fig. 16 shows the rule for `while`-loops. It states that, given p is an invariant of the loop body π_1, the execution the entire loop also maintains this invariant. This rule is not as easy to apply as the rules in Fig. 2: There is no general technique to automatically infer a loop invariant p that is strong enough to show the safety of a path with loops.

In our example, we already have a sufficiently strong loop invariant at hand: The conjunction of the closed forms of the recurrence equations $(\mathtt{i} = 1 + n)$ and $(\mathtt{j} = 1 + n)$ implies that $(\mathtt{i} = \mathtt{j})$ is an invariant of the loop in Fig. 15. This predicate is sufficient to show that $(\mathtt{i} \geq 10) \wedge (\mathtt{j} < 10)$ can not hold. Therefore, we can apply the rule in Fig. 16 to show the safety of the program and of the path in Fig. 15. By applying the loop rule to the loop in Fig. 14, we infer

$$\{(\mathtt{i} = \mathtt{j})\}\ \|\!:\ [\mathtt{i} < 10];\ \mathtt{i} := \mathtt{i} + 1;\ \mathtt{j} := \mathtt{j} + 1\ :\!\|\ [\mathtt{i} \geq 10]\ \{(\mathtt{i} \geq 10) \wedge (\mathtt{i} = \mathtt{j})\}. \tag{19}$$

The prefix $\mathtt{i} := 0;\ \ \mathtt{j} := 0$ establishes the pre-condition. Since the post-condition of (19) implies $(\mathtt{j} \geq 10)$, the assertion cannot be violated.

Obviously, the closed forms of the recurrence equations are always loop invariants for their corresponding parametrised counterexamples. Adding these loop invariants to the set of predicates P can result in a significantly smaller number of refinement iterations. Note that these predicates refer to the parameters $\{n_1, n_2, \ldots\}$ introduced by GUESSITERATIONS (see Fig. 13). Instead of trying to eliminate these parameters from the predicates, we instrument the original program with corresponding induction variables (see Fig. 17). For each loop, we introduce an *induction variable* n_i, which is initialised before the loop is entered, and increased at the end of the loop body. This modification has no impact on the safety of the current counterexample or the program. The advantage of this approach is that it is not necessary to modify the abstraction algorithm (see Sect. 2): In order to make sure that the information about the loop invariant is preserved, we add *three* versions of the recurrence predicate p to P, namely $p[n/0]$, p, and $p[n/n+1]$. Then, the resulting abstraction is strong enough to show that p is preserved when the loop is traversed along the path of the spurious counterexample:

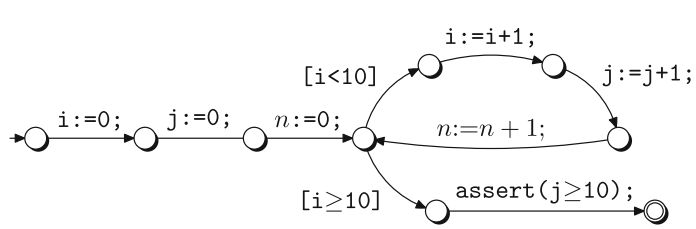


Fig. 17. The program of Fig. 14 augmented with an induction variable n

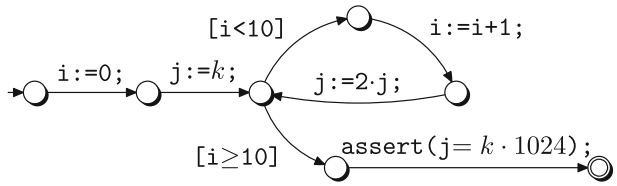


Fig. 18. A safe program

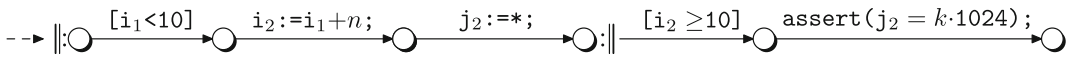


Fig. 19. A parametrised path with a loop for the program in Fig. 18

1. First, $\{p[n/0]\}$ n:= 0 $\{p\}$ establishes p upon entrance to the loop.
2. Let stmt be the statement that modifies the induction variable. Then $\{p\}$ stmt $\{p[n/n+1]\}$ holds.
3. Finally, by the assignment rule, $\{p[n/n+1]\}$ n:=$n+1$ $\{p\}$ holds upon exit from the loop.

Intuitively, we avoid the universal quantification over n by an induction over the parameter n. In our example in Fig. 17, the predicates $(\mathtt{i} = 0)$, $(\mathtt{i} = n)$, $(\mathtt{i} = n+1)$, $(\mathtt{j} = 0)$, $(\mathtt{j} = n)$, and $(\mathtt{j} = n+1)$ are sufficient to prove the loop invariant $(\mathtt{i} = n) \wedge (\mathtt{j} = n)$. In combination with the predicates $(\mathtt{i} \geq 10)$ and $(\mathtt{j} \geq 10)$, this invariant is strong enough to show the safety of all unwindings of the counterexample in Fig. 15.

The loop invariant given by the conjunction of the recurrence predicates is not always strong enough to eliminate the spurious counterexample. The reason is that the approach presented above ignores all statements in the loop body except the ones that contribute a recurrence equation. We observe that the original counterexample is also infeasible, since the parametrised path is more general than the original path. Therefore, adding the predicates that we can extract from the non-parametrised path is sufficient to eliminate the original counterexample. In addition, we add the recurrence predicates, hoping that they eliminate other potential counterexamples that contain more than one loop iteration from the model.

Predicate abstraction is able to establish disjunctive invariants that can be expressed in terms of the predicates [BMMR01]. Therefore, our technique is also able to establish the safety of programs with disjunctive loop invariants. In Sect. 5, we will give an example for such a program.

4.2. Refinement using unwound spurious counterexamples

The scheme we use to solve recurrence equations matches only the cases specified in step ②.❷ of Fig. 13. It fails to solve recurrences as simple as $\mathtt{j}_n = 2{\cdot}\mathtt{j}_{n-1}$. Therefore, the approach described in Sect. 3.3 yields the parametrised path in Fig. 19 for the program in Fig. 18. The resulting formula does not constrain the variable $\mathtt{j}_2$:

$$\ldots \wedge (\mathtt{i}_1 < 10) \wedge (\mathtt{i}_2 = n) \wedge (\mathtt{i}_2 \geq 10) \wedge (\mathtt{j}_2 \neq \mathtt{k} \cdot 1024) \tag{20}$$

GuessIterations determines that 10 is the smallest value for n such that Formula (20) is satisfiable. The corresponding unwound path, however, is safe. Even though the predicate $(\mathtt{i} = n)$ is a loop invariant, it is not strong enough to show the safety of the program.

In that case, we fall back on the traditional refinement approach. To eliminate all spurious counterexamples represented by the path with loops, it is necessary to add the refinement predicates from the proof of safety for the unwound counterexample. In our example, this approach yields the predicates $(\mathtt{j} = 2 \cdot \mathtt{k})$, $(\mathtt{j} = 4 \cdot \mathtt{k})$, $(\mathtt{j} = 8 \cdot \mathtt{k})$, $\ldots$, $(\mathtt{j} = 1024 \cdot \mathtt{k})$ and $(\mathtt{i} < 10)$, $(\mathtt{i}+1 < 10)$, $\ldots$, $(\mathtt{i}+9 < 10)$, which are sufficient to show the safety of the path.

An obvious disadvantage of this approach is that it generates a large number of predicates. The traditional refinement technique, however, yields the same set of predicates, but needs at least ten refinement steps, while our technique shows the safety of the program in only three abstraction refinement cycles. In Sect. 7, we present benchmarks for which this eager refinement approach performs better than traditional iterative refinement.

Integrating our technique into CEGAR. Figure 20 shows how our loop detection algorithm and the improved refinement technique are integrated into the traditional abstraction refinement cycle (see Fig. 4 in Sect. 2). The analyses presented in Sect. 3 (Figs. 9 and 13) are introduced between the model checking step and the simulation

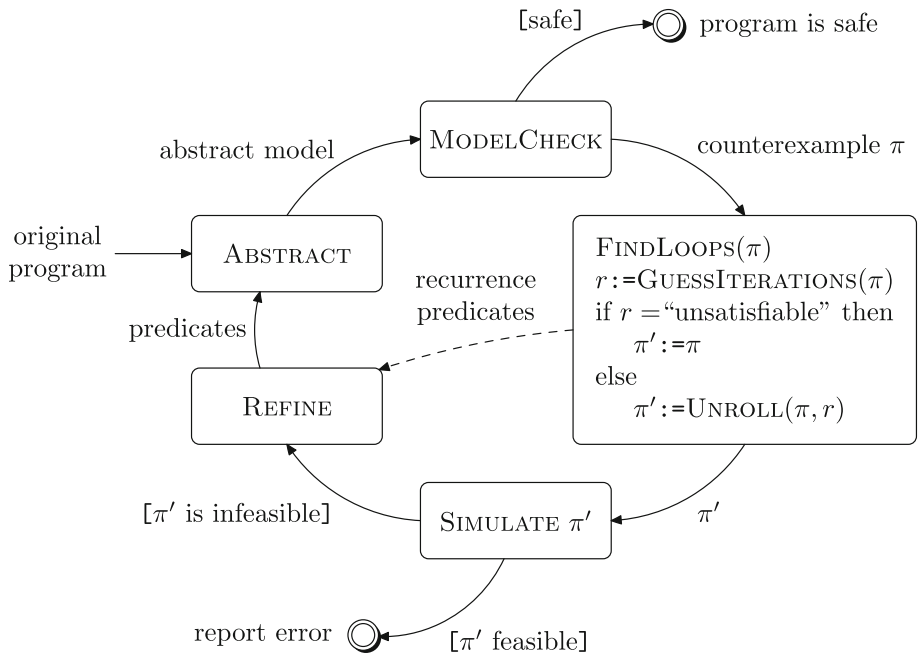


Fig. 20. Integrating loop detection into the abstraction refinement cycle

phase. Depending on the result r of the GuessIterations heuristic, the potential loops in the counterexample π may be unrolled accordingly (indicated by Unroll(π, r)), yielding a loop-free counterexample π'. As explained in Sect. 4, the recurrence predicates are added to the set of predicates P in the refinement step (indicated by the dashed arrow in Fig. 20). Notably, our approach does not require any major modifications of the original steps of the CEGAR algorithm.

5. Examples

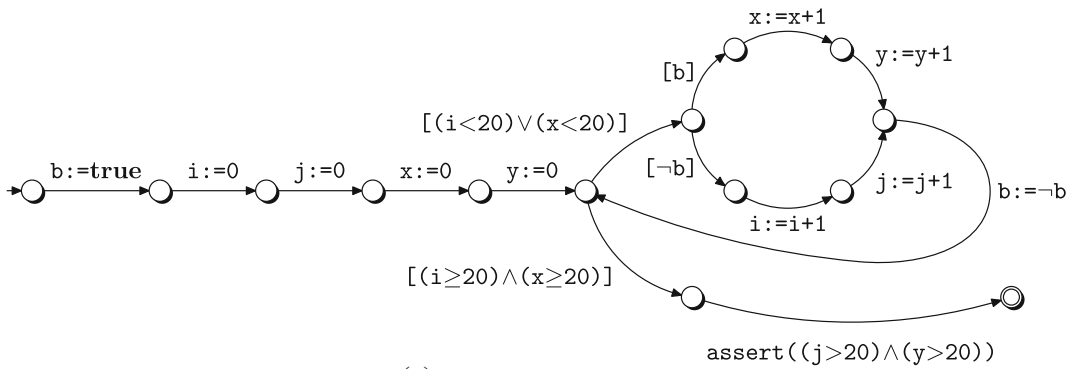
Figure 21 shows three programs which are slightly more sophisticated than the examples discussed so far. We discuss how these programs are verified using the approach presented in the previous sections.

Alternating branches. The program in Fig. 21a is not safe: The assertion can be violated by iterating the loop 40 times. Note that the branches of the conditional statement in the loop body are alternating. Initially, our approach detects a potential loop that repeatedly executes the same branch (for instance, the branch in which `i` and `j` are increased). The corresponding parametrised counterexample is infeasible, since `x` is initialised to 0 in the prefix of the path and never modified when traversing the loop. The predicates ($\mathtt{i} < 20$), ($\mathtt{x} < 20$), and $\neg\mathtt{b}$, which are determined using the traditional refinement approach, eliminate this spurious counterexample and force the execution of both branches of the conditional statement in the correct order. The model checker is now forced to unwind the loop twice and reports a corresponding abstract counterexample. Again, this counterexample contains a potential loop, namely

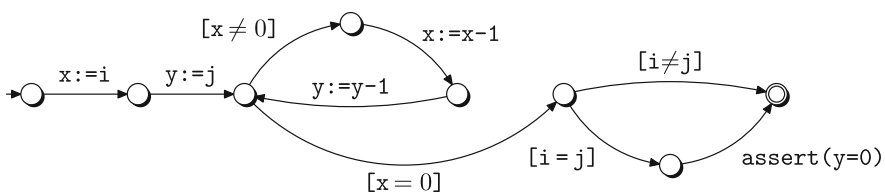
$$\ldots \| : [(\mathtt{i} < 20) \vee (\mathtt{x} < 20)];\ [\mathtt{b}];\ \mathtt{x} := \mathtt{x} + 1;\ \mathtt{y} := \mathtt{y} + 1;\ \mathtt{b} := \neg\mathtt{b};$$
$$[(\mathtt{i} < 20) \vee (\mathtt{x} < 20)];\ [\neg\mathtt{b}];\ \mathtt{i} := \mathtt{i} + 1;\ \mathtt{j} := \mathtt{j} + 1;\ \mathtt{b} := \neg\mathtt{b} : \| \ \ldots .$$

The body of this loop is an unwinding of the cycle in the control flow graph in Fig. 21a. GuessIterations (see Fig. 13) yields 20 as a promising candidate for the number of iterations of this loop. Finally, the forward simulation of the corresponding unwound counterexample confirms that it is indeed a feasible path that violates the assertion.

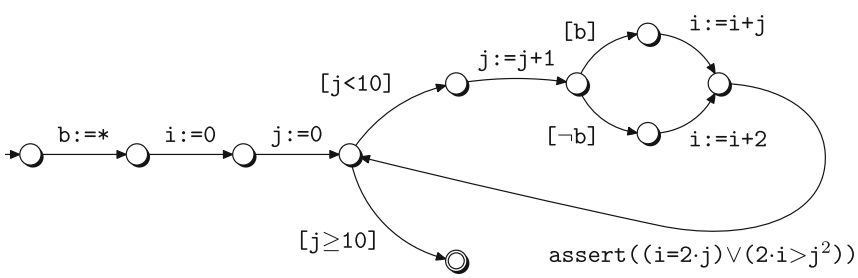
In the setting described above, adding the recurrence predicates $\mathtt{x} = n$, $\mathtt{y} = n$, $\mathtt{i} = n$, and $\mathtt{j} = n$ fails to provide any benefit. If, however, we change the assertion in Fig. 21a to $\mathtt{assert}((\mathtt{j} \geq 20) \wedge (\mathtt{y} \geq 20))$, the resulting loop invariant $(\mathtt{x} = \mathtt{y}) \wedge (\mathtt{i} = \mathtt{j})$ is sufficient to show the safety of the modified program.



(a) Loop body with alternating branches



(b) A program presented by Jhala and McMillan in [JM06]



(c) A loop with a disjunctive invariant

Fig. 21. A panopticon of programs that can be handled using the approach presented in this paper

Diverging sequence of predicates. Figure 21b shows a program presented by Jhala and McMillan [JM06]. For this example, the traditional refinement heuristic[4] yields a diverging sequence of predicates insufficient to represent the loop invariant $(\mathtt{i} = \mathtt{j}) \Rightarrow (\mathtt{x} = \mathtt{y})$. Our approach is capable of detecting the loop and inferring the recurrence predicates $\mathtt{x} = \mathtt{i} - n$ and $\mathtt{y} = \mathtt{j} - n$. In combination with the conditions $(\mathtt{i} = \mathtt{j})$ and $(\mathtt{x} = 0)$, the resulting loop invariant $(\mathtt{x} = \mathtt{i} - n) \wedge (\mathtt{y} = \mathtt{j} - n)$ is sufficient to establish the safety of the program.

Disjunctive and non-linear invariants. The program in Fig. 21c has two interesting aspects: The loop invariant is non-linear; moreover, it is disjunctive. Similar to Fig. 21a, the loop contains a conditional statement with two branches. In the program in Fig. 21c, however, only one of the two branches can be executed. The non-deterministic assignment $\mathtt{b} := *$ determines which branch is selected. Depending on this choice, either the assertion $(\mathtt{i} = 2{\cdot}\mathtt{j})$ or the assertion $(2{\cdot}\mathtt{i} > \mathtt{j}^2)$ holds. Therefore, the invariant of the whole loop is the disjunction $(\mathtt{i} = 2{\cdot}\mathtt{j}) \vee (2{\cdot}\mathtt{i} > \mathtt{j}^2)$ Our approach detects the two parts of the invariant separately and leaves the task to merge this information into a disjunction to the predicate abstraction framework. Our algorithm detects two alternative loops, namely

$$\|\!:\ [\mathtt{j} < 10];\ \mathtt{j} := \mathtt{j} + 1;\ [\neg\mathtt{b}];\ \mathtt{i} := \mathtt{i} + 2;\ \mathtt{assert}((\mathtt{i} = 2 \cdot \mathtt{j}) \vee (2 \cdot \mathtt{i} > \mathtt{j}^2))\ :\!\| \quad \text{and}$$

$$\|\!:\ [\mathtt{j} < 10];\ \mathtt{j} := \mathtt{j} + 1;\ [\mathtt{b}];\ \mathtt{i} := \mathtt{i} + \mathtt{j};\ \mathtt{assert}((\mathtt{i} = 2 \cdot \mathtt{j}) \vee (2 \cdot \mathtt{i} > \mathtt{j}^2))\ :\!\|.$$

[4] Jhala and McMillan refer to what we call the traditional refinement heuristic as "typical predicate heuristic".

The recurrence predicates ($\mathtt{j} = n$) and ($\mathtt{i} = 2 \cdot n$) are sufficient to show the safety of the first loop. The latter case is more complicated, since the two assignments are interdependent. We resolve this dependency by processing the recurrence equations in topological order. First, we determine the closed form $\mathtt{j} = n$ for the assignment `j:=j+1`. Using this closed form, we eliminate `j` from the assignment `i:=i+j` and obtain the instruction $\mathtt{i}{:=}\mathtt{i}{+}\,n$. For the corresponding recurrence equation, $\mathtt{i}_n{:=}\mathtt{i}_{n-1}{+}\,n$, we compute the closed form

$$\mathtt{i}_n = \mathtt{i}_0 + \frac{n \cdot (n+1)}{2}$$

(as explained in Fig. 13). Since $\mathtt{i}_0$ is 0, the resulting recurrence predicate is $\mathtt{i} = \frac{n \cdot (n+1)}{2}$. This predicate, in combination with $\mathtt{j} = n$, implies $2 \cdot \mathtt{i} > \mathtt{j}^2$. Therefore, the recurrence predicates generated by the algorithm proposed in Sect. 4.1 are (in theory) sufficient to prove the safety of the program. In practice, unfortunately, it turns out that most predicate-abstraction-based model checking tools do not support non-linear arithmetic operations at all. Even though our verification tool SatAbs [CKSY05] is able to handle non-linear arithmetic by converting the operations to propositional formulas, the SAT instances generated during the verification of the program in Fig. 21 c turn out to be too complex for the underlying SAT solver.

This negative result, however, must not be mistaken as evidence that our approach does not scale in the presence of non-linear arithmetic in general. For the following counterexample, for instance, our implementation is able to determine 20 as the number of iterations for which the assertion is violated. This takes only two refinement cycles and less than one second:

$$\mathtt{i} = 0;\ \mathtt{j} = 0;\ \|\!:\ [*];\ \mathtt{j} := \mathtt{j} + 1;\ \mathtt{i} := \mathtt{i} + \mathtt{j};\ \mathtt{assert}(\mathtt{i} < 210)\ :\!\|$$

(Here, $[*]$ denotes a condition that non-deterministically evaluates to either **true** or **false** in each iteration of the loop.)

6. Conditions for completeness

The examples in the previous section raise the question of whether the class of programs for which our approach is complete (i.e., able to prove safety) can be defined rigorously. Even though the traditional CEGAR approach discussed in Sect. 2 may succeed to establish an invariant that is sufficiently strong to show the safety (this is the case for the program in Fig. 14, for instance), it fails to do so in general: As discussed in Sect. 5, the traditional refinement algorithm yields a sequence of diverging predicates for the program in Fig. 21b. Our algorithm is able to find loop invariants for a larger class of programs than the traditional refinement approach and may avoid divergence.

In order to define the class of programs for which our approach is complete, we start with a characterisation of the paths for which our algorithm finds sufficiently strong loop invariants. Given a path π with a loop, where π_0, π_1, and π_2 are loop-free sub-paths, and

$$\pi := \pi_0;\ \|\!:\ [p];\ \pi_1\ :\!\|\,[\neg p];\ \pi_2;\ \mathtt{assert}(q), \tag{21}$$

a predicate r is a sufficiently strong loop invariant if all of the following conditions hold:

- r is an invariant of the loop, i.e., $\{p \wedge r\}\ \pi_1\ \{r\}$ holds,
- there is a predicate q_0 such that $\{\mathbf{true}\}\ \pi_0\ \{q_0\}$ holds and $q_0 \Rightarrow r$, i.e., the prefix π_0 establishes the invariant r,
- there is a predicate p_2 such that $\{p_2\}\ \pi_2\ \{q\}$ holds and $(\neg p \wedge r) \Rightarrow p_2$, i.e., the negated loop condition combined with the invariant r implies the pre-condition of π_2 with respect to the post-condition q.

These conditions follow immediately from the rule in Fig. 16. Under the implication order, the loop invariant r is bounded from below by the strongest post-condition of π_0 (denoted by $sp(\pi_0, \mathbf{true})$), and bounded from above by the weakest pre-condition for π_2 terminating with q true (denoted by $wp(\pi_2, q)$).[5] Note that these bounds are not necessarily loop invariants. Our algorithm is able to prove the path π safe if it manages to compute a loop invariant that lies within these bounds, i.e., the predicates P, which determine the abstract domain, must contain an exact representation of such an invariant.

[5] Therefore, the set of invariants is a complete lattice. We refer the reader to [Cou00] for a detailed discussion of this issue.

Our algorithm is "more complete" than the traditional CEGAR approach in the sense that it is able to compute invariants not detected by a refinement approach that does not take the information about loops into account. The class of loop invariants our heuristic is able to infer is restricted, though. Currently, we support only invariants of the form

$$\mathtt{x} = \alpha + \beta \cdot n + \gamma \frac{n \cdot (n+1)}{2} \tag{22}$$

(where x and n are variables and α, β, and γ are expressions constant throughout the loop). Furthermore, an invariant of this kind can only be constructed if the sub-paths π_0 and π_1 of the path π (as defined in (21)) match the following pattern:

$$\begin{array}{l} \pi_0 = \ldots;\ \mathtt{x} := \alpha_0;\ \ \ldots;\ \mathtt{y} := \alpha_1;\ \ \ldots \\ \pi_1 = \ldots;\ \mathtt{x} := \mathtt{x} + \beta_0;\ \ \ldots;\ \mathtt{y} := \mathtt{y} + \beta_1 + \mathtt{x};\ \ \ldots \end{array} \tag{23}$$

As before, α_0, α_1, β_0, and β_1 are expressions not modified in the loop body, and x and y are arbitrary scalar variables. The ellipses indicate arbitrary instructions that do not modify x and y, and the instructions $\mathtt{y} := \alpha_1$ and $\mathtt{y} := \mathtt{y} + \beta_1 + \mathtt{x}$ are optional. Our implementation simplifies arithmetic expressions in order to increase the number of matches. Furthermore, if the pattern matches more than one set of instructions, the algorithm constructs one invariant for each matching combination of assignments. Since all these invariants hold by construction for the detected loop, this does not lead to a combinatorial explosion.

A syntactic definition more restrictive than the pattern in (23) would be too strong: The invariant that is required depends on the assertion that is checked, and a slice of the path may be sufficient to show the safety of the path. Furthermore, we are not restricted to purely arithmetic invariants: If the instructions indicated by the ellipses contain array accesses, pointer arithmetic, or non-linear operations, the resulting loop invariant may use a combination of these theories. If necessary, the set of invariant templates can be increased by using more sophisticated algorithms for solving recurrence equations (see, for instance [vEBG04]).

We conclude that our algorithm is complete for programs for which all paths with loops are of the form (23), and the conditions listed above hold for the resulting invariants. If the sub-paths π_0 and π_2 (see (21)) also contain loops, then the conditions have to be strengthened: The upper and lower bounds for the invariant are not determined by the strongest and weakest pre- and post-conditions of the paths, respectively, but by the strongest and weakest predicates that the abstraction refinement algorithm can infer at the loop entry and loop exit locations.

The completeness of the traditional abstraction refinement algorithm is analysed and compared to a iterative fixed point algorithm with oracle-guided widening in [BPR02]. The results presented there also apply to our algorithm, since the invariants our algorithm detects are a superset of the invariants detected by the traditional refinement technique.

7. Experimental results

We evaluate our approach using a set of programs that contain known buffer overflows. For this purpose, we implemented the technique described in the previous sections into our predicate abstraction-based verification tool SatAbs [CKSY04, CKSY05].[6] SatAbs uses a SAT-solver [ES04] as decision procedure. SatAbs translates arithmetic operations into propositional formulas representing the corresponding hardware implementation, i.e., the bounded size and the bit-vector semantics of integers are modelled accurately. Thus, we rely on efficient satisfiability checking algorithms to solve the constraint-satisfaction problem in step ③ in Fig. 13. While the SAT-solver may be a potential bottle-neck of the verification process (as indicated at the end of Sect. 5), modern SAT-solvers tend to cope extremely well with problems that do not contain complicated arithmetic expressions.

Table 1 shows a comparison of the loop detection algorithm and a version of SatAbs in which the loop detection feature is disabled. We measure the effect of the loop detection algorithm on the number of iterations, the number of predicates generated, and the total runtime of the tool. The Aeon benchmark, also presented in [KW06] and listed in the upper section of Table 1, demonstrates the potential of our approach: Aeon 0.02a is a mail transfer agent that contains a buffer overflow, which can be triggered by means of an overly long environment variable `HOME`. The content of this variable is obtained using the `getenv` POSIX API function and is copied (by means of `strcpy`) to a buffer of fixed size without checking the length of the string.

[6] Available at http://www.cprover.org/satabs/

Table 1. Experimental results

Application	Bulletin	W/o Loop Detection		With Loop Detection		Speedup
		Iterations	Predicates	Iterations	Predicates	
Aeon 0.02a	CVS-2005-1019	>500	>500	2	3	–/15 s
Aeon 0.02a	CVS-2005-1019 (buffer size 5)	10	54	2	3	6.6
Aeon 0.02a	CVS-2005-1019 (buffer size 10)	23	119	2	3	431.4
OpenSER	CVE-2006-6749	20	99	2	3	171.7
OpenSER	CVE-2006-6749 patched	28	128	20	139	3.9
bind	CVE-2001-0011	7	33	6	28	0.5
bind	CVE-2001-0011 patched	7	45	8	54	0.6
sendmail	CVE-2003-0681	5	33	2	9	3.3
sendmail	CVE-1999-0047	3	8	1	1	1.7
MADWiFi	CVE-2006-6332	5	33	3	27	1.8
apache	CVE-2004-0940	10	55	11	60	1.0

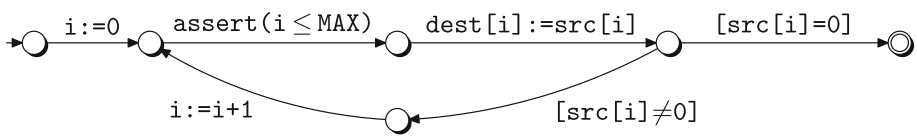


Fig. 22. A model of the `strcpy` function

We replaced the `getenv` function by a model that returns a string of non-deterministic size and content. Figure 22 shows the implementation of `strcpy`, augmented with an assertion[7] that fails if `i` exceeds the upper bound of the string `dest`. In the original program, the size of the `dest` buffer is 512 bytes. Our implementation detects the relevant loop immediately and reports the buffer overflow within 15 s, half of which are spent simulating the unwound counterexample. Without loop detection, SATABS is unable to provide an answer within a reasonable amount of time. Therefore, we reduced the size of the buffer to 5 and 10 bytes, respectively. As expected, the number of iterations necessary to detect the buffer overflow increases linearly with the size of the buffer. As we report in [KW06], the runtime of traditional predicate abstraction tools grows exponentially with the number of iterations of the loop, while our loop detection algorithm is not sensitive to the size of the buffer. For a buffer size of 10, our loop detection algorithm is already 400 times faster than the version of SATABS not supporting the loop detection feature. A similar comparison (also based on Aeon) of our approach to SLAM and BLAST can be found in [KW06], showing that our heuristic can prevent the exponential increase of the runtime both SLAM and BLAST exhibit in the presence of loops.

We obtained the remaining entries in Table 1 by running our algorithm on problems selected from a buffer overflow benchmark presented by Ku et al. [KHCL07]. These test-cases are simplified versions of a variety of buffer overflow vulnerabilities in open source programs like `OpenSER`, `bind`, and `apache`. We used the unmodified, publicly available benchmark to generate the entries in Table 1. The bounds of the loops in these programs are very small, since the benchmark was designed to evaluate software model checking tools that are based on traditional predicate abstraction. If we increase the bounds to a realistic size, the runtime of the traditional algorithm increases exponentially, while the runtime of the loop detection algorithm is mainly determined by the simulation of the unwound counterexample and remains almost unchanged.

The benchmark comprises unsafe programs and their corresponding patched (and therefore safe) counterparts. Our experiments show that our approach works particularly well for unsafe programs, like the `OpenSER` benchmark [KHCL07]. For the corresponding patched version of `OpenSER`, the speedup is less impressive but still measurable: Once the loop is detected, our refinement algorithm adds the relevant predicates in a single iteration. In the case of the `bind` benchmark [KHCL07], our algorithm has no positive impact on the performance. Even though it detects the loop, the number of iterations is too small to result in a significant improvement in runtime. The reason for the performance penalty is that our model checker for abstract programs, in which we implemented the detection algorithm for abstract loops (see Fig. 9), is not as fast and optimised as the SMV model checker [McM92], which SATABS uses by default. Our algorithm fails to detect the loop in the patched

[7] Our tool SATABS automatically generates assertions for array bounds, division by zero, and pointer validity. For the Aeon program, which has approximately 800 lines of code, SATABS generates 576 such assertions.

version of `bind` and in the `apache` benchmark [KHCL07]. In both cases, this has no significant impact on the number of predicates and iterations.

Our experimental results confirm that the technique typically yields a significant performance improvement if our heuristic manages to detect the loop. If, on the other hand, our algorithm fails to detect the crucial loop, the performance impact is negligible.

8. Related work

The approach we present extends the loop detection algorithm in [KW06]. The algorithm presented in [KW06] is covered in Sect. 3. The refinement technique based on adding recurrence predicates (see Sect. 4) is an improvement of the traditional refinement algorithm [BR02a] applied in [KW06].

Beyer et al. [BHMR07b] propose to combine counterexample-guided abstraction refinement (CEGAR) [CGJ+00] and invariant synthesis to prove the absence of counterexamples. Similar to our approach, the algorithm aims at computing invariants of loops in counterexamples. The resulting *path invariants* contain universal quantifiers. Unlike the recurrence predicates generated by our technique, universally quantified predicates are not readily integrated into existing predicate abstraction-based software verification tools (e.g., SLAM [BR02b, BCLR04], BLAST [HJMS02], MAGIC [CCG+04], F- SOFT [IYG+05], and SATABS [CKSY05]) and require special treatment in the abstraction phase. Similar to our recurrence equation-based approach, the algorithm used to generate path invariants is only complete for a certain class of invariant templates (specified in the language of linear arithmetic with uninterpreted function symbols) [BHMR07a]. The class of invariants covered by our approach is discussed in Sect. 6.

DAIKON [EPG+07] is a tool that dynamically detects potential ("likely") invariants of a program. It relies on executing the program using a suite of test cases (e.g., regression tests). DAIKON traces the values of variables at appropriate points in the program (e.g., procedure entries and exits) by means of instrumenting the code. Using the resulting data, DAIKON evaluates a large number of potential invariants (applying pre-defined patterns including linear relations over two or three variables, intervals, ordering of sequences, etc.) and reports the invariants that it observes to hold for the recorded test runs. This approach suffers from the same problems as testing, since it depends heavily on the test suite. A single additional test run may invalidate the reported invariants. Thus, the results are not guaranteed to be invariants of the program, though the precision is claimed to be high [EPG+07]. The invariants generated by our approach are guaranteed to be invariants of the analysed paths and not just of a concrete run of that path. Furthermore, DAIKON generates a large number of potential invariants, which makes it unsuitable in our setting, since the scalability of predicate abstraction decreases rapidly with an increasing number of predicates. Our technique generates only invariants that are promising candidates to eliminate a spurious counterexample.

The DAIKON tool is highly extensible and allows the user to add new types of invariants. Our implementation would certainly benefit from such a flexibility. We intend to investigate the feasibility of extending our approach to more complex data-types such as those provided by the C++ template library (see Sect. 9).

Jain et al. propose to strengthen the transition relation of the original program using statically computed linear invariants of the form $\pm x \pm y \leq c$ [JIG+06]. They observe that predicate abstraction generates a more precise abstraction if the original transition relation is strengthened. Since using all generated invariants may not be beneficial, they use a heuristic to filter out invariants not deemed important. In contrast, we compute invariants on demand, and our technique detects a different class of invariants, including nonlinear ones.

Leino and Logozzo suggest to strengthen loop invariants on demand, as the need for stronger invariants arises during the verification process [LL05]. The accuracy of the (numeric) domain used by the abstract interpreter is increased if the theorem prover fails to show the safety of a path. The technique combines invariants generated by means of abstract interpretation with automatic theorem proving. In contrast, our technique and the path invariants approach are based on predicate abstraction and model checking. Furthermore, neither path invariants, nor Leino and Logozzo's or Jain's approach aim at accelerating the detection of counterexamples.

The incompleteness of traditional predicate-abstraction based CEGAR implementations (e.g., [BR02b, BCLR04]) is a well known problem [Cou00]: If the program is safe, the abstraction refinement algorithm is incomplete unless the refinement step introduces a predicate that represents a (sufficiently strong) invariant of the program. Jhala and McMillan [JM06], to whom we owe the example in Fig. 21b, address this issue by avoiding the generation of a diverging sequence of refinement predicates by restricting the search space of the interpolation-based [HJMM04] refinement algorithm. Unlike our recurrence-based technique and the path invariant approach, their refinement algorithm considers only loop-free counterexamples.

The existence of a counterexample with loops can also be shown by means of induction on the loop bound [WGI07]. This approach has the advantage that it is not necessary to unwind the counterexample. The approach, as presented in [WGI07], is restricted to non-nested loops with a single induction variable and a loop condition that is monotonic with respect to the loop bound. Moreover, it is not suitable for showing the absence of counterexamples.

Path Slicing is an approach that shortens counterexamples by dropping the statements that have no impact on the reachability of the program location in question [JM05]. The statements and branches that can be bypassed are eliminated by backward slicing: For each program location, the set of relevant variables whose valuations at that point determine whether or not the error location is reachable is computed. The feasibility of a path slice implies the feasibility of the original counterexample, but assumes termination of the omitted code sequences. Path slicing eliminates loops during the symbolic simulation if and *only* if they do *not* contribute to the reachability of the error location. Therefore, path slicing is orthogonal to our approach, since it prevents expensive unrolling of loops that are not related to the error.

Ball et al. propose a technique based on identifying a sequence of must-transitions through loops in an abstract transition system generated by predicate abstraction [BKS07]. In order for this approach to succeed, the concrete transition system must adhere to a set of restrictions, for instance, the abstract state a at the loop entry must represent a finite set of concrete states, and each concrete state represented by a must not have more than one successor in a. This technique aims at proving the termination of loops in order to leap loops in the abstraction refinement process without the need for further refinement. In contrast, our approach does not impose any restrictions on the concrete transition system. Furthermore, our goal is not proving loop termination, but to find a single counterexample that traverses the loop and violates an assertion.

Linear programs have been proposed by Armando as an alternative, more fine-grained formalism for abstractions of sequential programs [ACM04]. Due to the higher expressiveness of linear programs (in comparison to Boolean programs), this approach yields a smaller number of spurious execution traces. However, the abstraction algorithm is restricted to a pointer-free subset of the C programming language that employs linear arithmetic and arrays [ABM06, ABC+07].

Rybalchenko and Podelski present a complete method for detecting linear ranking functions of non-nested program loops [PR04]. The inferred ranking function poses an upper bound for the iterations of the loop. This bound is not necessarily tight. Combined with abstraction-refinement, this approach enables proofs of program termination [CPR05]. A proof of termination is insufficient to show the feasibility of counterexamples with loops, since the violation of the property usually depends on the number of iterations. Therefore, we utilise a method that provides the exact number of loop iterations necessary to reach the error state.

Acceleration is a technique that aims at computing the repeated iteration of a sequence of transitions of a symbolic transition system in one step [BFLP03, FL02]. It targets finite linear systems and counter automata. The technique accelerates the computation of the reachable states of the system, but does not specifically target the detection of counterexamples. Acceleration is also called exact widening [CC77]. Our heuristic GuessIterations(π) (see Fig. 13) may also be interpreted as a widening and acceleration step on the transition function defined by the body of the detected loop.

9. Conclusion

This paper presents a novel program verification approach that makes two contributions: First, it enables predicate abstraction to find bugs that emerge as a result of a high number of iterations of loops. We propose an algorithm to detect loops in abstract models and explain how the traditional simulation and refinement algorithms can be extended to cope with loops. Our algorithm is based on replacing the assignment statements in the loop body into a closed-form representation that is parametrised by the number of loop iterations. We show how this closed-form representation can be used to accelerate the detection of counterexamples. Second, we show that the closed form representations, used as refinement predicates, can also be useful to prove the absence of counterexamples. Our implementation outperforms the traditional abstraction-refinement approach based on predicate abstraction on many typical buffer overflow examples.

Future work and outlook. Our idea is neither restricted to the simple invariants discussed in Sect. 6 nor to predicate abstraction. We consider to extend our approach to invariants over more complicated data-types such as those provided by the C++ template library. We intend to base this work on the verification technique presented in [BGK07], which is based on an abstract model of the template library that preserves relevant facts such as

the size of C++ containers (e.g., lists). Furthermore, our technique can (with slight modifications) be applied in any setting in which an abstraction is subsequently refined in a counterexample-guided manner. We have plans to integrate our approach into an interpolation-based model checking tool [McM06].

The idea of extracting invariants from paths is very promising and has recently been successfully applied in a number of different ways (e.g., see [KW06, BHMR07b, EPG+07], discussed in Sect. 8). It is particularly powerful in combination with a refinement-based static analysis technique, allowing it to derive non-trivial disjunctive invariants. We expect to see extensions of our idea that enable the verification of a larger, more general class of programs.

Acknowledgments

The second author would like to thank Prof. Peter Lucas for awakening his lasting interest in formal verification by teaching Hoare logic in the introductory course on computer science at the Graz University of Technology. Both authors thank Sir Tony Hoare, Vijay D'Silva and Mitra Purandare, for their helpful and inspiring comments. Thomas Wahl deserves a special note of thanks for his thorough reading and his insightful comments on this paper. We are particularly grateful to Joseph Ruskiewicz for the critical comments on an earlier version of this paper and for sharing his extensive knowledge about Hoare logic.

References

[ABC+07] Armando A, Benerecetti M, Carotenuto D, Mantovani J, Spica P (2007) The EUREKA tool for software model checking. In: Automated software engineering (ASE), pp 541–542. ACM Press, New York

[ABM06] Armando A, Benerecetti M, Mantovani J (2006) Model checking linear programs with arrays. In: Software model checking (SoftMC). Electronic notes in theoretical computer science, vol 144. Elsevier, Amsterdam, pp 79–94

[ACM04] Armando A, Castellini C, Mantovani J (2004) Software model checking using linear constraints. In: International conference on formal engineering methods (IFCEM). Lecture notes in computer science, vol 3308. Springer, Berlin, pp 209–223

[Bal05] Ball T (2005) Engineering theories of software intensive systems. NATO Science Series II: mathematics, physics and chemistry, vol 195. Formalizing counterexample-driven refinement with weakest preconditions. Springer, Berlin, pp 121–139

[BCLR04] Ball T, Cook B, Levin V, Rajamani SK (2004) SLAM and Static driver verifier: technology transfer of formal methods inside Microsoft. In: Integrated formal verification (IFM). Lecture Notes in Computer Science, vol 2999. Springer, Berlin

[BFLP03] Bardin S, Finkel A, Leroux J, Petrucci L (2003) FAST: Fast acceleration of symbolic transition systems. In: Computer aided verification (CAV). Lecture notes in computer science, vol 2752. Springer, Berlin, pp 118–121

[BGK07] Blanc N, Groce A, Kroening D (2007) Verifying C++ with STL containers via predicate abstraction. In: Automated software engineering (ASE). IEEE, USA, pp 521–524

[BHMR07a] Beyer D, Henzinger TA, Majumdar R, Rybalchenko A (2007) Invariant synthesis for combined theories. In: Verification, model checking and abstract interpretation (VMCAI). Lecture notes in computer science, vol 4349. Springer, Berlin, pp 378–394

[BHMR07b] Beyer D, Henzinger TA, Majumdar R, Rybalchenko A (2007) Path invariants. In: Programming language design and implementation (PLDI). ACM Press, New York, pp 300–309

[BKS07] Ball T, Kupferman O, Sagiv M (2007) Leaping loops in the presence of abstraction. In: Computer aided verification (CAV). Lecture notes in computer science, vol 4590. Springer, Berlin, pp 491–503

[BMMR01] Ball T, Majumdar R, Millstein T, Rajamani SK (2001) Automatic predicate abstraction of C programs. In: Programming language design and implementation (PLDI). ACM Press, New York, pp 203–213

[BPR01] Ball T, Podelski A, Rajamani SK (2001) Boolean and Cartesian abstraction for model checking C programs. In: Tools and algorithms for the construction and analysis of systems (TACAS). Lecture notes in computer science, vol 2031. Springer, Berlin, pp 268–283

[BPR02] Ball T, Podelski A, Rajamani SK (2002) Relative completeness of abstraction refinement for software model checking. In: Tools and algorithms for the construction and analysis of systems (TACAS). Lecture notes in computer science, vol 2280. Springer, Berlin, pp 158–172

[BR00] Ball T, Rajamani SK (2000) Bebop: a symbolic model checker for Boolean programs. In: Model checking and software verification (SPIN), Lecture notes in computer science, vol 1885. Springer, Berlin, pp 113–130

[BR02a] Ball T, Rajamani S (2002) Generating abstract explanations of spurious counterexamples in C Programs. Technical Report MSR-TR-2002-09, Microsoft Research, Redmond

[BR02b] Ball T, Rajamani SK (2002) The SLAM project: debugging system software via static analysis. In: Principles of programming languages (POPL). ACM Press, New York, pp 1–3

[CC77] Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of programming languages (POPL). ACM Press, New York, pp 238–252

[CC79] Cousot P, Cousot R (1979) Systematic design of program analysis frameworks. In: Principles of programming languages (POPL). ACM Press, New York, pp 269–282

[CCG+04] Chaki S, Clarke EM, Groce A, Jha S, Veith H (2004) Modular verification of software components in C. IEEE Trans Softw Eng 30(6):388–402

[CFR+91] Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK (1991) Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst 13(4):451–490

[CGJ+00] Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Computer aided verification (CAV). Lecture notes in computer science, vol 1855. Springer, Berlin, pp 154–169

[CGL92] Clarke E, Grumberg O, Long DE (1992) Model checking and abstraction. In: Principles of programming languages (POPL). ACM Press, New York, pp 343–354

[CGP99] Clarke E, Grumberg O, Peled D (1999) Model checking. MIT Press, Cambridge

[CKS05] Cook B, Kroening D, Sharygina N (2005) Symbolic model checking for asynchronous Boolean programs. In: Model checking and software verification (SPIN). Lecture notes in computer science, vol 3639. Springer, Berlin, pp 75–90

[CKSY04] Clarke E, Kroening D, Sharygina N, Yorav K (2004) Predicate abstraction of ANSI-C programs using SAT. Formal Methods Syst Des (FMSD) 25:105–127

[CKSY05] Clarke EM, Kroening D, Sharygina N, Yorav K (2005) SATABS: SAT-based predicate abstraction for ANSI-C. In: Tools and algorithms for the construction and analysis of systems (TACAS). Lecture notes in computer science, vol 3440. Springer, Berlin, pp 570–574

[Cou00] Cousot P (2000) Partial completeness of abstract fixpoint checking. In: International symposium on abstraction, reformulation, and approximation (SARA). Lecture notes in computer science, vol 1864. Springer, Berlin, pp 1–25.

[CPR05] Cook B, Podelski A, Rybalchenko A (2005) Abstraction-refinement for termination. In: Static analysis symposium (SAS). Lecture notes in computer science, vol 3672. Springer, Berlin, pp 87–101

[Dij75] Dijkstra EW (1975) Guarded commands, nondeterminacy and formal derivation of programs. Commun ACM 18(8):453–457

[EHRS00] Esparza J, Hansel D, Rossmanith P, Schwoon S (2000) Efficient algorithms for model checking pushdown systems. In: Computer aided verification (CAV). Lecture notes in computer science, vol 1855. Springer, Berlin, pp 232–247

[EPG+07] Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, Xiao C (2007) The Daikon system for dynamic detection of likely invariants. Sci Comput Program 69(1–3):35–45

[ES04] Eén N, Sörensson N (2004) An extensible SAT-solver. In: Theory and applications of satisfiability testing (SAT), vol 2919. Springer, Berlin, pp 502–518

[FL02] Finkel A, Leroux J (2002) How to compose Presburger-accelerations: applications to broadcast protocols. In: Foundations of software technology and theoretical computer science (FST TCS). Lecture notes in computer science. Springer, Berlin, pp 145–156

[Flo67] Floyd RW (1967) Assigning meanings to programs. In: Symposium on applied mathematics. Mathematical aspects of computer science, vol 19. American Mathematical Society, Providence, pp 19–32

[GKP89] Graham RL, Knuth DE, Patashnik O (1989) Concrete mathematics: a foundation for computer science. Addison-Wesley Longman Publishing Co., Inc., Reading

[Gri87] Gries D (1987) The science of programming. Springer, Berlin

[GS97] Graf S, Saïdi H (1997) Construction of abstract state graphs with PVS. In: Computer aided verification (CAV). Lecture notes in computer science, vol 1254. Springer, Berlin, pp 72–83

[HJM+02] Henzinger TA, Jhala R, Majumdar R, Necula GC, Sutre G, Weimer W (2002) Temporal-safety proofs for systems code. In: Computer aided verification (CAV). Lecture notes in computer science, vol 2404. Springer, Berlin, pp 526–538

[HJMM04] Henzinger TA, Jhala R, Majumdar R, McMillan KL (2004) Abstractions from proofs. In: Principles of programming languages (POPL). ACM Press, New York, pp 232–244

[HJMS02] Henzinger TA, Jhala R, Majumdar R, Sutre G (2002) Lazy abstraction. In: Principles of programming languages (POPL). ACM Press, New York, pp 58–70

[Hoa69] Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580

[IYG+05] Ivančić F, Yang Z, Ganai MK, Gupta A, Shlyakhter I, Ashar P (2005) F- SOFT: Software verification platform. In: Computer aided verification (CAV). Lecture notes in computer science, vol 3576. Springer, Berlin, pp 301–306

[JIG+06] Jain H, Ivancic F, Gupta A, Shlyakhter I, Wang C (2006) Using statically computed invariants inside the predicate abstraction and refinement loop. In: Computer aided verification (CAV). Lecture notes in computer science, vol 4144. Springer, Berlin, pp 137–151

[JM05] Jhala R, Majumdar R (2005) Path slicing. In: Programming language design and implementation (PLDI). ACM Press, New York, pp 38–47

[JM06] Jhala R, McMillan KL (2006) A practical and complete approach to predicate refinement. In: Tools and algorithms for the construction and analysis of systems (TACAS). Lecture notes in computer science, vol 3920. Springer, Berlin, pp 459–473

[KHCL07] Ku K, Hart TE, Chechik M, Lie D (2007) A buffer overflow benchmark for software model checkers. In: Automated software engineering (ASE). ACM Press, New York, pp 389–392

[KS06] Kroening D, Sharygina N (2006) Approximating predicate images for bit-vector logic. In: Proceedings of TACAS 2006. Lecture notes in computer science, vol 3920. Springer, Berlin, pp 242–256

[Kur95] Kurshan R (1995) Computer-aided verification of coordinating processes. Princeton University Press, Princeton

[KW06] Kroening D, Weissenbacher G (2006) Counterexamples with loops for predicate abstraction. In: Computer aided verification (CAV). Lecture notes in computer science, vol 4144. Springer, Berlin, pp 152–165

[LL05] Leino KRM, Logozzo F (2005) Loop invariants on demand. In: Programming languages and systems (APLAS). Lecture notes in computer science, vol 3780. Springer, Berlin, pp 119–134

[McM92] McMillan KL (1992) The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University

[McM06] McMillan KL (2006) Lazy abstraction with interpolants. In: Computer aided verification (CAV). Lecture notes in computer science, vol 4144. Springer, Berlin, pp 123–136

[Nel89] Nelson G (1989) A generalization of Dijkstra's calculus. ACM Trans Program Lang Syst (TOPLAS) 11(4):517–561

[PR04] Podelski A, Rybalchenko A (2004) A complete method for the synthesis of linear ranking functions. In: Verification, model checking and abstract interpretation (VMCAI). Lecture notes in computer science, vol 2937. Springer, Berlin, pp 239–25

[vEBG04] van Engelen RA, Birch J, Gallivan KA (2004) Array data dependence testing with the chains of recurrences algebra. In: Innovative architecture for future generation high-performance processors and systems (IWIA). IEEE, USA, pp 70–81

[WGI07] Wang C, Gupta A, Ivančić F (2007) Induction in CEGAR for detecting counterexamples. In: Formal methods in computer-aided design (FMCAD). IEEE, USA, pp 77–84

Received 10 February 2008

Accepted in revised form 14 March 2009 by C.B. Jones and J.C.P. Woodcock

Published online 7 April 2009