PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 41, NO. 1, PP. 91-101 (1998)

COMBINING ABDUCTION AND MODEL CHECKING TECHNIQUES FOR REPAIR OF CONCURRENT PROGRAMS (EXTENDED ABSTRACT)

Francesco BUCCAFURRI*, Thomas EITER**, Georg GOTTLOB*** and Nicola LEONE***

* DIEMA, Universitá di Reggio Calabria
I-89100 Reggio Calabria, Italy
E-mail: bucca@si.deis.unical.it
** Institut für Informatik
Universität Gießen
Arndtstraße 2
D-35392 Gießen, Germany
E-mail: eiter@informatik.uni-giessen.de
*** Institut für Informationssysteme
Technische Universität Wien
Paniglgasse 16
A-1040 Wien, Austria
E-mail: (leone|gottlob)@dbai.tuwien.ac.at

Received: Oct. 20, 1997

Abstract

We investigate the combination of AI techniques with model checking, which is a successful approach to verification of particular concurrent systems. We present the system repair problem and as an application the problem of repairing (i.e. correcting) concurrent programs. Moreover, we describe optimization techniques for reducing the search space of a repair, which use the concept of counterexample.

Keywords: automated verification, model checking, abduction, diagnosis and repair.

1. Introduction

Model checking (CLARK and EMERSON, 1981; CLARKE et al., 1986) is an automatic technique for verifying finite-state concurrent systems such as circuit designs and communication protocols. Specifications are expressed in a propositional temporal logic, and the system is modeled as a state transition graph. Model checking has several important advantages over other verification methods such as mechanical theorem provers or proof checkers. The most relevant is that it is highly automatic. Moreover, recent advances in model checking allow to verify tremendously large systems (BURCH et al., 1992). Major companies including Intel, Motorola, Fujitsu and ATT started using symbolic model checkers to verify circuits and protocols.

The research reported here is motivated by the questions whether AI can contribute to model checking, and whether vice versa the AI community can gain something from that area. This paper gives a first answer and studies the combination of abductive reasoning, cf. (POOLE, 1988; CONSOLE et al., 1991; EITER and GOTTLOB, 1995), and model checking. It does not present a final theory; the current approach works under restrictions and must be extended. However, it is a first step towards integrating AI into model checking.

Process P_A

```
1:  flag1A := true;
2:  turn1B := false;
3:  if flag1B & turn1B then
4:     goto 3;
5:  x := x and y;
6:  flag1A := false;
7:  if turn1B then
8:  begin  flag2A := true;
9:         turn2B := true;
10:        if flag2B & turn2B then
11:           goto 10;
12:        y := false;
13:        flag2A := false;
    end
14: goto 1;
```

Process P_B

```
1:  flag1B := true;
2:  turn1B := false;
3:  if flag1A & ¬ turn1B then
4:     goto 3;
5:  x := x and y;
6:  flag2B := true;
7:  turn2B := false;
8:  if flag2A & ¬ turn2B then
9:     goto 8;
10: y := not y;
11: x := x or y;
12: flag2B := false;
13: flag1B:= false;
14: goto 1;
```

Fig. 1. A concurrent program $\mathcal{P}$

To give the flavour of the approach, consider the concurrent program $\mathcal{P}$ in *Fig. 1.* It has processes P_A and P_B, which share two Boolean variables x and y. To ensure mutual exclusion of the assignments to x and y, some control variables, *flags* and *turns*, are used following Peterson's scheme (PETERSON, 1981), in which each critical section (5 and 12 in P_A and –nested 5-11 and 10 in P_B) is executed under an *entry* and *exit protocol.*

The system specification prescribes that P satisfies *mutual exclusion* for assignments to x and y, respectively, and *absence of starvation.* E.g. P_A must not execute instruction 5, if P_B executes instruction 5 or 11 at the same time. Absence of starvation requires that a request of a process for a resource must eventually be satisfied. Clearly, this makes sense only if the underlying scheduler is *fair*; absence of starvation cannot be ensured if the scheduler always dispatches the same process.

Inspection shows that $\mathcal{P}$ is not correct, even under fair schedules; instruction 2 of P_A should be $turn1B := true$; however, detecting the error is

not immediate. Model checking allows to check the correctness of $\mathcal{P}$ fully automatically. The system specification, mutual exclusion and absence of starvation is expressed in the temporal logic *ACTL* (GRUMBERG and LONG, 1994); fair schedules are specified through *fairness constraints* (CLARKE et al., 1994). Then, an automatic procedure verifies whether $\mathcal{P}$ fulfills the specification.

However, if a model checker finds $\mathcal{P}$ incorrect, usually it cannot single out the error, and is far from fixing it. By using abductive reasoning, our method tackles this problem: it tries to locate the bug and proposes a repair of the program. The method works under a single error hypothesis, in which possible errors may occur in the left or right side of an assignment; exchanging two successive assignments and multiple errors should also be manageable.

Like abduction, program repair is computationally expensive. Even if we plausibly restrict in *Fig. 1* to the assignments of control variables, we must consider 12 assignments and 6 control variables. Thus, up to 72 attempts of repair may be done, each of which requires a call of the model checker.

Towards more efficient program repair, we have designed an optimization technique which exploits a suitable formalization of a *counterexample* from (CLARKE et al., 1994). It guarantees to make here only up to 15 attempts.
More details can be found in the extended paper (BUCCAFURRI et al., 1997).

2. Preliminaries on ACTL

ACTL (GRUMBERG and LONG, 1994) is a major fragment of Computational Tree Logic (CLARKE et al., 1986), as it allows *abstraction* and *compositional reasoning* (CLARKE and GRUMBERG, 1992; GRUMBERG and LONG, 1994). There, *state* and *path formulas* are propositions on a point in time and a computation path, respectively, using linear-time (LT) operators $\mathbf{X}$ ("next time"), $\mathbf{U}$ ("Until"), $\mathbf{V}$ ("unless") and the path quantifier $\mathbf{A}$ ("for all paths").

DEFINITION 1 For a set AP of atomic propositions, *ACTL* is the smallest class of state formulas on AP, where

- any atom $a \in AP$ is a state (s)-formula;
- if f, g are s-formulas, then $f \vee g$, $f \wedge g$ are s-formulas, as well as $\neg f$ if f lacks LT-operators;
- if f, g are s-formulas, then $\mathbf{X}f$, $f\mathbf{U}g$ and $f\mathbf{V}g$ are path (p)-formulas;
- if f is a p-formula, then $\mathbf{A}(f)$ is an s-formula.

Let $\mathbf{F}f = true\mathbf{U}f$ ("sometimes" f) and $\mathbf{G}f = false\mathbf{V}f$ ("always" f).

The semantics of $ACTL$ is given in terms of *Kripke structures.*

DEFINITION 2 A *Kripke structure* is a 5-tuple $M = (AP, S, S_0, R, L)$, where

- AP is a finite set of atomic propositions,
- S is a finite set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $R \subseteq S \times S$ is a transition relation,
- $L : S \to 2^{AP}$ assigns $s \in S$ the set of atomic propositions true at s.

A *path* π in M is an infinite sequence $[s_0, s_1, \cdots s_i \cdots]$ such that $(s_i, s_{i+1}) \in R$, $i \geq 0$. We use the notation $\pi(i) = s_i$ and $\pi^j = [\pi(j), \pi(j+1), \cdots]$.

DEFINITION 3 $M, s \models f$ denotes satisfaction of an s-formula f in a state s of M, and $M, \pi \models g$ satisfaction of a p-formula g along a path π in M, where

1. $M, s \models p$, if $p \in L(s)$ where $p \in AP(M)$;
2. $M, s \models \neg f_1$, if $M, s \not\models f_1$;
3. $M, s \models f_1 \vee f_2$, if $M, s \models f_1$ or $M, s \models f_2$;
4. $M, s \models f_1 \wedge f_2$, if $M, s \models f_1$ and $M, s \models f_2$;
5. $M, s \models \mathbf{A}(g_1)$, if $M, \pi \models g_1$ for all paths π with $\pi(0) = s$;
6. $M, \pi \models f$, if $M, \pi(0) \models f$;
7. $M, \pi \models \mathbf{X} f_1$, if $M, \pi^1 \models f_1$;
8. $M, \pi \models f_1 \mathbf{U} f_2$, if $k \geq 0$ exists such that $M, \pi^k \models f_2$ and $M, \pi^j \models f_1$, for all $0 \leq j < k$;
9. $M, \pi \models f_1 \mathbf{V} f_2$, if for every $k \geq 0$, $M, \pi^j \not\models f_1$ for all $0 \leq j < k$ implies $M, \pi^k \models f_2$;

$M \models f$ denotes $M, s_0 \models f$, for every $s_0 \in S_0$.

Fair computations are modeled by *fairness constraints* (BURCH et al., 1992).

DEFINITION 4 A *FC-Kripke structure* is an expanded Kripke structure $M = (M', F)$, where F is a finite set of formulas f (*fairness constraints*).

A path π in M is *fair*, if for every $f \in F$ and $i \geq 0$ there exists $j \geq i$ such that $M, \pi(j) \models f$. Satisfaction as $\models$ but where $\mathbf{A}(g_1)$ is w.r.t. all fair paths is denoted by $\models_f$ (simply $\models$, if unambiguous).

Intuitively, path π is fair if each $f \in F$ holds infinitely often along π. Typical fairness constraints on a concurrent program $\mathcal{P} = P_1, \ldots, P_n$ are $a_i \equiv$ "Process P_i is active;" fair paths amount to schedules of the (infinite) processes where no P_i is paused forever (which is not expressible in $ACTL$).

3. Abductive Reasoning on *ACTL* Specifications

We employ abductive reasoning for the problem of modifying a FC-Kripke structure M in order to satisfy an *ACTL* formula. Recall that abductive reasoning is, roughly speaking, an inverse of modus ponens. From a rule $\phi \supset \psi$ and ψ, abduction concludes ϕ as an explanation of ψ.

Given a FC-Kripke structure M (the "system") and a formula f, we define the *System Repair Problem* as the abduction problem whose solution consists of a set of modifications to the transition relation R (additions or deletions of tuples in R), such that f is true in the modified system.

DEFINITION 5 Let $R \subseteq S \times S$. Every pair $\delta = \langle (s_1, s_2), \oplus \rangle$, where $s_1 \in S$, $s_2 \in S$, and $\oplus \in \{-, +\}$, is a *simple modification.* The *application of* δ *on* R, denoted $\delta(R)$, is $R \cup \{(s_1, s_2)\}$, if $\oplus = +$, and $R \setminus \{(s_1, s_2)\}$ else.

DEFINITION 6 A *modification for* M is a set Γ of simple modifications for R s.t. $\Gamma^+ \cap \Gamma^- = \emptyset$, where $\Gamma^+ = \{(s_1, s_2) \mid \langle s_1, s_2, + \rangle \in \Gamma\}$ and $\Gamma^- = \{(s_1, s_2) \mid \langle s_1, s_2, - \rangle \in \Gamma\}$. $mod(M)$ denotes the set of all modifications for M. The *result* $\Gamma(M)$ *of* Γ is the FC-Kripke structure $(AP, S_0, S, R^\Gamma, L, F)$, where $R^\Gamma = \bigcap_{\delta \in \Gamma^-} \delta(R) \cup (\bigcup_{\delta \in \Gamma^+} \delta(R) \setminus R)$.

Intuitively, a modification Γ of system M is a set of nonconflicting simple modifications, and $\Gamma(M)$ is the result of their simultaneous application.

DEFINITION 7 A *system repair problem (SRP)* is a triple $Q = \langle M, f, \mathcal{Y} \rangle$ of an FC-Kripke structure M, formula f, and computable Boolean function $\mathcal{Y}$ on $mod(M)$. Any modification Γ s.t. $\mathcal{Y}(\Gamma) = \mathbf{true}$ is called admissible. A *solution for* Q is an admissible modification Γ for M s.t. $\Gamma(M) \models f$.

This definition describes the abduction problem in a general framework. The admissibility function $\mathcal{Y}$ is domain-dependent; e.g., in the case of concurrent programs, $\mathcal{Y}$ is derived from possible changes to the code.

A solution of an SRP is an abductive conclusion of how to modify the system for satisfying f. Note that this process is, as well-known, intimately related to theory change and counterfactual reasoning.

4. Repair of Concurrent Programs

A concurrent program is a collection $\mathcal{P} = P_1, \ldots, P_n$ of processes running in parallel. We assume a system with shared memory, i.e. all variables $\mathbf{x} = x_1, x_2, \ldots, x_l$ are accessible to all processes, and that every x_j is Boolean; by coding, this allows expressing all finite domains. Note that communication protocols usually use such domains (flags, finite counters, etc).

In Pnueli's model (PNUELI, 1981), each process P_i is represented as a labeled digraph $G(P_i) = \langle N_i, E_i \rangle$, where

- $N_i = \{1, 2, \ldots, m_i\}$ is the set of **break points** (*BP*s) of P_i, i.e. the points before the code and between successive statements (cf. *Fig. 1*).
- E_i is a set of labeled arcs $a = b_j \rightarrow b_k$. Intuitively, a represents transition from b_j to b_k in P_i. The label $l(a) = (c(\mathbf{x}), stmnt)$ is a pair of a Boolean condition $c(\mathbf{x})$ and an either empty or assignment statement *stmnt*. Intuitively, the transition $b_j \rightarrow b_k$ can be followed if $c(\mathbf{x})$ is true, and then *stmnt* is executed.

Given a set F of fairness constraints, associate with $\mathcal{P}$ an FC-Kripke structure $M_F(\mathcal{P})$ such that

- $S = \times_i N_i \times_j D_j \times \{1, \ldots, n\}$ is the Cartesian product of all sets N_i of break points, domains D_j of the x_j, and the set of process numbers. A state s intuitively corresponds to a configuration of the system, where the last component tells the process executed in the previous step;
- S_0 holds only states s whose last components have the value 1.
- R is obtained from the graphs $G(P_i)$ by following one arc in some $G(P_i)$ from one state to another.
- AP contains $\mathbf{x}$, variables $b_i^k \equiv$ "P_i is at break point k," and $e_i \equiv$ "P_i was executing".
- L is straightforward from that.

Suppose a formula f is a formal specification for $\mathcal{P}$ and we have fairness constraints F for $\mathcal{P}$. Then, $\mathcal{P}$ fulfills f iff $M_F(\mathcal{P}) \models f$. If $\mathcal{P}$ does not fulfill f, we are interested in a change to the code of $\mathcal{P}$ such that the modified program $\mathcal{P}'$ fulfills f. This amounts to a (mostly nontrivial) SRP.

EXAMPLE 1 (ctd) For $\mathcal{P}$ in *Fig. 1*, define

$$\begin{aligned} f \;=\; & \textstyle\bigwedge_{i=1,2}^{V=A,B} \mathbf{AG}(flagiV \rightarrow \mathbf{AF}\neg flagiV) \\ & \wedge \mathbf{AG}(\neg(b_1^{12} \wedge b_2^{9})) \wedge \mathbf{AG}(\neg(b_1^{4} \wedge (b_2^{4} \vee b_2^{10}))) \,. \end{aligned}$$

f says that in every computation, P_V must eventually exit the critical section i after entering it, and that both processes may not be simultaneously in a critical section. For $F = \{e_A, e_B\}$ (fair scheduling), $M_F(\mathcal{P}) \not\models f$.

We tackle program repair under simplifying assumptions. We focus here on the case that a single statement is wrong, and allow that the solution of an instance will be an *assignment correction* γ in $\mathcal{P}$, which is either

- replacing the right-hand side of an assignment $x_j := expr$, by a constant (**true** or **false**), or
- changing the variable of its left-hand side.

DEFINITION 8 A *correction for* $\mathcal{P}$ is a triple $\alpha = \langle k, b, \gamma \rangle$, where $b \in N_k$ is a break point in P_k and γ specifies an assignment correction for the statement b in P_k. Let $\mathcal{P}^\alpha$ be the program obtained from $\mathcal{P}$ if α is implemented.

Implementing α merely changes the label of b in $G(P_k)$; thus, $M_F(\mathcal{P}^\alpha)$ and $M_F(\mathcal{P})$ coincide except for R. Hence, α induces a system modification Γ as in Def. 6 such that $R^\alpha = R^\Gamma$. Not every Γ is induced by some α; $\mathcal{Y}_\mathcal{P}$ selects those Γ which are. Thus, program repair reduces to a SRP.

DEFINITION 9 The tuple $D = \langle \mathcal{P}, F, f \rangle$ defines a *program repair problem (PRP) for* $\mathcal{P}$ w.r.t. f under fairness constraints F. A *solution for* D is a solution Γ for the SRP $Q = \langle M_F(\mathcal{P}), f, \mathcal{Y}_\mathcal{P} \rangle$. A *repair* for D is a correction α for $\mathcal{P}$ such that D has a solution Γ induced by α.

Repairs are characterized as follows.

PROPOSITION 1 *A correction α is a repair for $\mathcal{P}$ w.r.t. f iff $M_F(\mathcal{P}^\alpha) \models f$.*

5. Computation of Repairs and Counterexamples

The naive algorithm for computing a solution for a PRP simply tries every possible correction α on each assignment in $\mathcal{P}$, and checks $M_F(\mathcal{P}^\alpha) \models f$. However, this is clearly not efficient.

Our approach restricts the search space by exploiting counterexamples. Informally, a counterexample for a PRP $\langle \mathcal{P}, F, f \rangle$ is a portion of the possible computation branches witnessing that f fails. Given a counterexample, our technique identifies corrections α under which it is invariant, i.e. still applies if α is implemented. Such α's are useless and can be discarded. This way, the space of candidate repairs may be drastically reduced.

Counterexamples were introduced in (CLARKE et al., 1994); the respective procedure returns as counterexample a single path in M. We need a richer concept as we must consider multiple, possibly nested paths. Thus we introduce *multi-sequences* and *multi-paths*.

DEFINITION 10 Every state $s \in S$ is a finite multi-sequence in S; if $\Pi_0, \Pi_1, \ldots$ are countably infinite many multi-sequences, then $\Pi = [\Pi_0, \Pi_1, \ldots]$ is a multi-sequence. $\Pi(i)$ denotes the i-th element of Π, and $or(\Pi) = s$ if $\Pi = s$ is finite and $or(\Pi) = or(\Pi(0))$ otherwise.

Informally, an infinite multi-sequence Π is a kind of infinitely branching tree whose leaves (where $or(\Pi)$ is the leftmost) are states. It enables representation of nested infinite paths; we call this a *multi-path*. There is a main path from which other paths branch off.

More formally, let the *main sequence* of Π, denoted $\mu(\Pi)$, be $\mu(\Pi) = s$, if $\Pi = s$ is finite, and $\mu(\Pi) = [or(\Pi(0)), or(\Pi(1)), \cdots]$ otherwise.

DEFINITION 11 A multi-sequence Π is a *multi-path* in M, if either Π is finite or $\mu(\Pi)$ is a path in M and for every $i \geq 0$, $\Pi(i)$ is a multi-path in M. Π is *fair*, if Π is finite or $\mu(\Pi)$ and every $\Pi(i)$ is fair.

E.g., consider $\Pi = [[[\, s_0, s_{1_1}, s_{1_2}, \cdots], s_{2_1}, s_{2_2} \cdots], s_{3_1}, s_{3_2}, \cdots]$. It has the main path $\mu(\Pi) = [s_0, s_{3_1}, s_{3_2}, \cdots]$. At the state s_0, the paths $\pi_1 = [s_0, s_{1_1}, s_{1_2}, \cdots]$, $\pi_2 = [s_0, s_{2_1}, s_{2_2}, \cdots]$ branch off.

Informally, a counterexample for f is a particular multi-path Π, originating at an initial state such that f is not true along Π. A counterexample for f without LT-operators is simply an initial state s_0 such that $M, s_0 \not\models f$. Counterexamples for more complex f are defined inductively, using the concept of local (l-) counterexample. For the definition, we need the concept of merge of two multi-paths.

DEFINITION 12 Let Π_1 and Π_2 be two multi-paths such that $or(\Pi_1) = or(\Pi_2)$. The *merge* of Π_1 and Π_2, denoted $\Pi_1 * \Pi_2$, is the multi-path

$$\Pi_1 * \Pi_2 = \begin{cases} \Pi_1, & \text{if } \Pi_2 = s \text{ is finite;} \\ [\Pi_1, \Pi_2(1), \Pi_2(2), \ldots], & \text{if } \Pi_2 \text{ is infinite, } \Pi_2(0) = s; \\ [\Pi_1 * \Pi_2(0), \Pi_2(1), \Pi_2(2), \ldots], & \text{otherwise.} \end{cases}$$

E.g., merging $\Pi = [[s_0, s_{1_1}, s_{1_2}, \cdots], s_{2_1}, s_{2_3}, \cdots]$ and $\Pi' = [s_0, s_{3_1}, s_{3_2}, \cdots]$ yields $\Pi * \Pi' = [[[s_0, s_{1_1}, s_{1_2}, \cdots], s_{2_1}, s_{2_3}, \cdots], s_{3_1}, s_{3_2}, \cdots]$, while $\Pi' * \Pi = [[[s_0, s_{3_1}, s_{3_2}, \cdots], s_{1_1}, s_{1_2}, \cdots], s_{2_1}, s_{2_2}, \cdots]$, which essentially represents the same branching of three paths.

DEFINITION 13 Let M be a FC-Kripke structure and f be a formula on $AP(M)$. A multi-path Π in M is a *local (l-) counterexample for* f, s.t., if

1. f has no LT-operators: $\Pi = s$ and $M, s \not\models f$;
2. $f = \mathbf{A}(f_1 \mathbf{U} f_2)$: Π is an infinite fair multi-path and either
   (2.1) there exists $k \geq 0$ such that $\Pi(k)$ is an l-counterexample for $f_1 \vee f_2$, $\Pi(i)$ is an l-counterexample for f_2, for each $0 \leq i \leq k$, and $\Pi(j)$ is a state, for $j > k$; or
   (2.2) $\Pi(i)$ is an l-counterexample for f_2, for each $i \geq 0$;
3. $f = \mathbf{A}(f_1 \mathbf{V} f_2)$: Π is an infinite fair multi-path and there exists a k such that every $\Pi(j)$, $0 \leq j < k$, is an l-counterexample for f_1, $\Pi(k)$ is an l-counterexample for f_2, and every $\Pi(\ell)$ is a state, for $\ell > k$;
4. $f = \mathbf{A}\mathbf{X} f_1$: Π is an infinite fair multi-path, $\Pi(1)$ is an l-counterexample for f_1, and $\Pi(i)$ is a state, for each $i \neq 1$;
5. $f = f_1 \vee f_2$, where f has LT-operators: $\Pi = \Pi_1 * \Pi_2$, where Π_i, $i = 1, 2$, is an l-counterexample for f_i;
6. $f = f_1 \wedge f_2$, where f has LT-operators: Π is an l-counterexample for either f_1 or f_2.

Counterexamples are particular l-counterexamples.

DEFINITION 14 Let M be a FC-Kripke structure and f be a formula on AP. An l-counterexample Π for f in M such that $or(\Pi)$ is an initial state of M is called a *counterexample for* f *in* M.

An example is considered below. The next theorem states that the concept of counterexample captures failure of a formula.

THEOREM 5.1 *Let M be a FC-Kripke structure, f a formula f. Then, $M, s \not\models f$ iff there exists a counterexample Π for f in M.*

Note that counterexamples, while mostly infinite objects, can be finitely represented; a construction by a modification of the procedure in (CLARKE et al., 1994) seems feasible.

The space of candidate repairs to a PRP $\langle \mathcal{P}, F, f \rangle$ can be reduced as follows: Assume Π is a counterexample for f in $M_F(\mathcal{P})$ and α a correction for $\mathcal{P}$. If Π' is a multi-path in $M_F(\mathcal{P}^\alpha)$ equivalent (in a formally precise sense) to Π, then Π' is a counterexample for f in $\mathcal{P}^\alpha$; hence, α is not a repair for the PRP. Based on this principle, we exploit two properties of counterexamples and repairs for optimization.

Execution : A multi-path Π *executes* a correction $\alpha = \langle k, m, \gamma \rangle$, if transition $(\pi(i), \pi(i+1))$ exists in some paths π of Π in which process P_k proceeds from break point m (i.e. executes the assignment after m).

THEOREM 5.2 *We have: Let $\alpha = \langle k, m, \gamma \rangle$ be a repair for the PRP $Q = \langle \mathcal{P}, F, f \rangle$. Then, every counterexample Π for f in $M_F(\mathcal{P})$ executes α.*

Exploitation : A multi-path Π *exploits* a correction $\alpha = \langle k, m, \gamma \rangle$, if it is infinite and one of the infinite paths π in Π has a transition $(\pi(i), \pi(i+1))$ in which x_i or x_j is evaluated, where $x_i := expr_i$ (resp. $x_j := expr_j$) is the assignment after m in P_k (resp. in P_k corrected by α).

THEOREM 5.3 *Let $V(\alpha) = \{x_i, x_j\}$. Let $\alpha = \langle k, m, \gamma \rangle$ be a repair for the PRP $Q = \langle \mathcal{P}, F, f \rangle$ such that no variable in $V(\alpha)$ occurs in f or F. Then, every counterexample Π for f in $M_F(\mathcal{P})$ exploits α.*

Any correction α which is not executed or not exploited by a counterexample Π can be immediately excluded as a repair. Given Π and α, this can be checked efficiently. This test can lead to drastic savings.

EXAMPLE 5.4 (cont'd) We have $M_F(\mathcal{P}) \not\models f$: Consider the path π (we show each state s_i, left to right, the break points for P_A and P_B, the flag and turn variables that are true, and the process lastly executed):

$\pi(0)$	$= s_0$	$= 1$	1		P_A
$\pi(1)$	$= s_1$	$= 2$	1	flag1A	P_A
$\pi(2)$	$= s_2$	$= 2$	2	flag1A, flag1B	P_B
$\pi(3)$	$= s_3$	$= 3$	2	flag1A, flag1B	P_A
$\pi(4)$	$= s_4$	$= 3$	3	flag1A, flag1B	P_B
	...				

From s_4, where $flag1B$ is true, a path π' branches off:

$$
\begin{array}{lllll}
\pi'(0) & = s_4 & & & \\
\pi'(1) & = s_5 & = 5\ 3 & \text{flag1A, flag1B} & P_A \\
\pi'(2) & = s_6 & = 5\ 4 & \text{flag1A, flag1B} & P_B \\
\pi'(3) & = s_7 & = 5\ 3 & \text{flag1A, flag1B} & P_B \\
\pi'(4) & = s_8 & = 6\ 3 & \text{flag1A, flag1B} & P_A \\
\pi'(5) & = s_9 & = 6\ 4 & \text{flag1A, flag1B} & P_B \\
\pi'(6) & = s_{10} & = 6\ 3 & \text{flag1A, flag1B} & P_B \\
\pi'(7) & = s_{11} & = 7\ 3 & \text{flag1B} & P_A \\
\pi'(8) & = s_{12} & = 14\ 3 & \text{flag1B} & P_A \\
\pi'(9) & = s_{13} & = 1\ 3 & \text{flag1B} & P_A \\
\pi'(10) & = s_{14} & = 2\ 3 & \text{flag1A, flag1B} & P_A \\
\pi'(11) & = s_{15} & = 3\ 3 & \text{flag1A, flag1B} & P_A \\
\pi'(12) & = s_5 & = 5\ 3 & \text{flag1A, flag1B} & P_A \\
\pi'(13) & = s_6 & = 5\ 4 & \text{flag1A, flag1B} & P_B \\
\pi'(14) & = s_7 & = 5\ 3 & \text{flag1A, flag1B} & P_B \\
\pi'(15) & = s_8 & & & \\
 & \cdots & & & \\
\pi'(i) & = s_{15} & & & \\
 & \cdots & & &
\end{array}
$$

where $flag1B$ is always true. Consider now the multi-path $\Pi = [s_0, s_1, s_2, s_3, [s_4, s_5, \cdots, s_{14}, s_{15}, s_5, s_6, \cdots, s_{14}, \cdots], s_5, s_6, s_7, \cdots]$. It is a counterexample for the formula $\mathbf{AG}(flag1B \rightarrow \mathbf{AF}\neg flag1B)$, and consequently also a counterexample for f.

The naive repair technique considers in P_A the assignments after break point $i \in \{1, 2, 6, 8, 9, 13\}$ and in P_B after break point $j \in \{1, 2, 6, 7, 12, 13\}$.

For simplicity, let us only consider repairs changing right sides of assignments as described. Then, our optimization technique allows us to restrict attention in P_A to $i \in \{1, 2, 6\}$ and in P_B to $j \in \{1, 2\}$. Indeed, the variables referenced along Π are $flag1A$, $flag1B$, $turn1B$. Thus, only 5 out of 12 candidate repairs remain. (In case of general repairs, 15 out of 72 remain.)

The only repair for $\mathcal{P}$ is $\alpha = \langle A, 2, \gamma \rangle$ where γ amounts to $turn1B := true$. Indeed, $\mathcal{P}^\alpha$ does not enable P_B to loop forever between 3 and 4.

References

[1] BUCCAFURRI, F. – EITER, T. – GOTTLOB, G. – LEONE, N. (1997): Enhancing Model Checking by AI Techniques, Technical Report, IFIGRR 9702, Institut für Informatik, Univ. Grießen, Germany, 1997.

[2] BURCH, J. – CLARKE, E. – MCMILLAN, K. – DILL, D. – HWANG, J. (1992): Symbolic Model Checking: 10^{120} States and Beyond. *Information and Computation*, Vol. 98(2) pp. 142–170.

[3] CLARKE, E. – EMERSON, E. (1981): Synthesis of Synchronization Skeletons for Branching Time Temporal Logic, In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, LNCS 131.

[4] CLARKE, E. – EMERSON, E. – SISTLA, A. (1986): Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, *ACM Transactions on Programming Languages & Systems*, Vol. 8(2) pp. 244–263.

[5] CLARKE, E. – GRUMBERG, O. – LONG, D. (1994): Verification Tools for Finite-State Concurrent Systems. In: *A Decade of Concurrency - Reflections and Perspectives*, LNCS 803.

[6] CLARKE, E. – GRUMBERG, O. – LONG, D. (1992): Model Checking and Abstraction. In *Proc. ACM Symposium on Principles of Programming Languages*, 1992. ACM Press.

[7] CONSOLE, L. – THESEIDER DUPRÉ, D. – TORASSO, P. (1991): On the Relationship between Abduction and Deduction. *Journal of Logic & Computation*, Vol. 1(5) pp. 661–690.

[8] EITER, T. – GOTTLOB, G. (1995): The Complexity of Logic-Based Abduction. *Journal of the ACM*, Vol. 42(1) pp. 3–42.

[9] GRUMBERG, O. – LONG, D. (1994): Model Checking and Modular Verification. *ACM Transactions on Programming Languages & Systems*, Vol. 16, pp. 843–872.

[10] PETERSON, G. (1981): Myths about the Mutual Exclusion Problem, *Information Processing Letters*, Vol. 12(3) pp. 115–116.

[11] PNUELI, A. (1981): The Temporal Semantics of Concurrent Programs, *Theoretical Computer Science*, Vol. 13, pp. 45–60.

[12] POOLE, D. (1988): A Logical Framework for Default Reasoning. *Artificial Intelligence*, Vol. 36, pp. 27–47.