I F I G
Research
Report

Institut für Informatik

Enhancing Symbolic Model Checking by AI Techniques

Francesco BUCCAFURRI
Thomas EITER
Georg GOTTLOB
Nicola LEONE

IFIG Research Report 9701
September 1997

Institut für Informatik
JLU Gießen
Arndtstraße 2
D-35392 Giessen, Germany
Tel: +49-641-99-32141
Fax: +49-641-99-32149
mail@informatik.uni-giessen.de
www.informatik.uni-giessen.de

Justus-Liebig-
Universität
Giessen

IFIG Research Report
IFIG Research Report 9701, September 1997

Enhancing Symbolic Model Checking by AI Techniques

Francesco Buccafurri[a] Thomas Eiter[c]
Georg Gottlob[b] Nicola Leone[d]

Abstract. Model checking is a fruitful application of computational logic with high relevance to the verification of concurrent systems. While model checking is capable of automatically testing that a concurrent system satisfies its formal specification, it can not precisely locate an error and suggest a repair, i.e., a suitable correction, to the system. In this paper, we tackle this problem by using principles from AI. In particular, we introduce the abstract concept of a system repair problem, and exemplify this concept on repair of concurrent programs and protocols. For the development of our framework, we formally extend the concept of counterexample, which has been proposed in model checking previously, and provide examples which demonstrate the need for such an extension. Moreover, we investigate into optimization issues for the problem of finding a repair, and present techniques which gain in some cases a considerable reduction of the search space for a repair.

[1]DIEMA, Universitá di Reggio Calabria, I-89100 Reggio Calabria, Italy. E-mail: bucca@si.deis.unical.it

[2]Institut für Informationssysteme, Technische Universität Wien, Paniglgasse 16, A-1040 Wien, Austria. E-mail: gottlob@dbai.tuwien.ac.at

[3]Institut für Informatik, Universität Gießen, Arndtstraße 2, D-35392 Gießen, Germany. E-mail: eiter@informatik.uni-giessen.de

[4]Institut für Informationssysteme, Technische Universität Wien, Paniglgasse 16, A-1040 Wien, Austria. E-mail: leone@dbai.tuwien.ac.at

1 Introduction

Model checking, which has been first proposed by Clarke and Emerson [9, 10], is an approach to automated verification of finite-state concurrent systems such as circuit designs and communication protocols. In this approach, specifications are expressed in a propositional temporal logic, and the concurrent system is modeled as a state transition graph, which amounts to a Kripke structure for this logic. Checking whether the system satisfies its specification, given by a logical formula, reduces then to test whether the Kripke structure is a model of the formula.

Model checking has several important advantages over other methods for verification of circuits and protocols, like mechanical theorem provers or proof checkers. The most relevant one is that it is efficient and highly automatic. Recent advances in model checking by using special data structures and algorithms, known as *symbolic model checking*, some of which have more than 10^{120} states [8]. A number of major companies including Intel, Motorola, Fujitsu, and ATT have started using symbolic model checkers to verify actual circuits and protocols. Thus, (symbolic) model checking is nowadays considered to be one of the most fruitful and promising applications of computational logic.

On the other hand, various techniques for diagnostic reasoning on systems have been developed in the field of AI, including logic-based approaches like model-based diagnosis and repair [27]. These approaches utilize general AI principles and are successfully used in different application domains. Our work approaches a new and promising field for application of AI techniques, which is in particular attractive for knowledge representation and reasoning methods, and thus adds to the application perspective of this field [1].

In this paper, we study the enhancement of model checking by abductive reasoning, which is a major technique in AI and knowledge representation, cf. [40, 16, 31, 3, 29, 19, 42, 4]. The work presented does not exhaustively treat this issue, and further work is necessary; however, it is a first step towards an integration of model checking with AI techniques, and may stimulate other work in this direction.

The main contributions of the present paper can be summarized as follows.

- We study the integration of (symbolic) model checking and AI principles. In particular, we introduce the *system repair problem* in the context of Computational Tree Logic (CTL) (a temporal logic used to express the specifications of concurrent systems to be checked), which is formal framework for repairing a concurrent system, described by a Kripke model, at the semantical level. Notice that in a different context, repair was introduced in [23, 24, 41].

  The system repair problem amounts to an interesting *abductive model revision* problem: Determine by abductive reasoning a suitable change of the system (i.e., of its Kripke model) such that the specification is satisfied upon this change. Interestingly, this problem is an intermingled abductive reasoning and theory revision problem, which is best understood as an abductive theory revision problem. In fact, the system repair problem can be modeled as an abductive theory revision problem in the frameworks of [33, 29]. Note that the close relationship between abduction and revision is well-recognized, and its investigation received increasing interest more recently, e.g., [5, 29, 33, 34].

- We show that the proposed framework for system repair can be profitably used, by providing an application to the repair of concurrent programs and protocols. In particular, we describe a program repair problem, in which repair of a concurrent program in terms of changes at the syntactical level (i.e., modifications of the program code) is mapped to changes at the semantical level. As dealing with all possible modifications is clearly unfeasible, we restrict in our approach to some types modifications which seem to be relevant in practice. A repair is then a sequence $\overline{\alpha} = \alpha_1 \cdots \alpha_q$ of basic corrections α_i on the program, such that the modified program satisfies the specification ϕ. Applying AI principles and, in particular, Occam's principle of parsimony, we provide also a notion of minimal solution, that prunes solutions which are not free of redundancy.

- We face the problem of searching for a program repair. In general, the search space for this problem is large, and might still contain a number of candidates, even if only elementary corrections such as inverting the value of an expression or exchanging the name of a variable in a Boolean assignment statement are adopted as repairs.

  In order to alleviate this problem, we develop optimization techniques which sensibly reduce the search space for a repair by exploiting structural information about the failure of the system provided by a counterexample. In particular, we formulate two pruning criteria referred to as correction execution and correction exploitation, which can be evaluated efficiently. In fact, given the program, a collection of candidate repairs, and a counterexample, the candidates violating these principles can be discarded in linear time. As we demonstrate, this may yield considerable savings, and thus applying correction execution and exploitation is an effective pruning method which comes at low computational cost.

- We formally extend the notion of counterexample from [11], which is a heuristically selected computation path from a conceptual counterexample tree (i.e., an evolving branching computation) that gives a hint at the failure of the system. As shown by examples, there are cases in which no single path is a counterexample. We therefore introduce the concept of a multi-path, which enables representation of nested paths. Multi-paths turn out to be a suitable tool for expressing full counterexample trees, which is needed for our purposes.

To give the flavor of theory and application developed in this paper, we discuss a motivating example.

Consider the concurrent program $\mathcal{P}$ in Figure 1. It consists of processes P_A and P_B, which share two common boolean variables x and y. To ensure mutual exclusion of the assignments to x and y, some control variables, *flags* and *turns*, are introduced, following the

Process P_A

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

Figure 1: A concurrent program $\mathcal{P}$

classical Peterson scheme [38], in which each critical section is executed obeying an *entry* and *exit protocol*. There are then two critical sections in each process, one for the assignments to x (statements 5 in P_A and statements 5–11 in P_B), and another one for the assignments to y (statements 12 in P_A and 10 in P_B, respectively); notice that in P_B, the critical section for y is nested into the critical section for y. Each variable $flagiV$ indicates the request of process V to enter critical section i, and *turniB* tells whether such a request by process B in case of simultaneous requests should be granted.

The critical sections have been set up for the purpose of fulfilling some part of the system specification. The complete specification prescribes that P satisfies *mutual exclusion* for assignments to x and y, respectively, and *absence of starvation*. For example, 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 (by setting a flag) must eventually be granted. Clearly, this makes sense only under the hypothesis that the underlying scheduler is *fair*; absence of starvation cannot be ensured if the scheduler always dispatches the same process.

Careful inspection of $\mathcal{P}$ shows that the program is not correct, even under fair schedules; instruction 2 of P_A should be $turn1B := true$. Even in this small example, however, detecting the error is not immediate for the nonexpert. Model checking allows for checking the correctness of $\mathcal{P}$ (and of much larger programs) in a fully automatic way. The specification of the system, mutual exclusion and absence of starvation, can be expressed in the temporal logic $ACTL$ [26]; fair schedules are specified my means of *fairness constraints* [11]. Then, an automatic procedure verifies whether the program meets the specifications or not. If the program is incorrect, however, model checkers usually can not single out the error precisely, and are far from fixing a bug.

By using abductive reasoning, our method goes beyond error checking: it tries to locate a bug and proposes a repair for the program, such that the modified program meets the specification. Our approach considers possible errors both in the left and right side of an assignment as well as the interchange of two successive assignments.

Like abduction, program repair comes at computational cost. Even if we assume the case of a single error in the program and we plausibly restrict in Figure 1 attention to the assignments of control variables, we must consider 12 assignments ($1, 2, 6, 8, 9, 13$ in P_A and $1, 2, 6, 7, 12, 13$ in P_B) and 6 control variables. Thus, 77 attempts of repair (by a single assignment modification or by interchanging two assignments) should be done, each of which requires a call of the model checker to see if it works.

Towards more efficient program repair, we have designed optimization techniques, based on counterexamples [11], for the case of a single error in the program, which is often considered in practice. By applying these techniques, our procedure makes only 17 (instead of 77) attempts in the worst case.

The remainder of this paper is structured as follows. In Section 2, we recall the syntax and the semantics of the logic *CTL*. In Section 3, we address the problem of modifying a system, given in terms of a Kripke structure, such that a given formula holds on it. In the course of this, we introduce the notion of system repair problem, which provides a general framework for the problem of properly changing a system at the semantical level in order to meet a formal specification, given in *CTL*. A system repair problem constitutes a kind of abductive model revision problem, and we outline how such a problem can be represented in the frameworks for abductive theory change proposed in [33, 29]. After that, we consider in Section 4 the corresponding problem at the syntactical level, in particular the one of correcting concurrent programs and protocols. The program repair problem, which is addressed there, resorts at the semantical level to a system repair problem. In Section 5, we then address the problem of finding repairs. For this purpose, we suitably generalize the notion of counterexample described in [11], and show that counterexamples characterize errors. In the subsequent Section 6, we address optimization techniques which, by use of counterexamples, may allow to considerably reduce the number of possible repairs that have to be considered. In particular, we formulate correction execution and correction exploitation, investigate their computational feasibility and show the effectiveness of the techniques on an example. The final Section 7 concludes the paper and states some issues for further work.

In order to increase readability, proofs of technical results except a few short ones have been moved into the appendix.

2 Computational Tree Logic

Computational Tree Logic (*CTL*) is a propositional branching-time temporal logic [9]; see [20, 12] for a rich background on this and further such logics. The semantics of *CTL* is given by *Kripke Structures* which model finite-state systems. *CTL* is used to represent specification in reactive systems. Linear-time features of *CTL* are useful to capture ordering of events in time. Branching time operators allow to take into account the existence of multiple possible future scenarios, starting from a given point of a computation. Indeed, in a branching frame the temporal order defines a tree which branches toward the future. Thus, every point of time has a unique past, but, in general, more than one future. Branching time operators allow us to deal with this form of non-determinism. Using these operators, we will express the truth or falsehood of a certain property as being relative to a given branch of the computation tree, such that we can express both *possible* properties (true in a possible evolution of time in the future) and *necessary* ones (true in all possible computation branches).

CTL is a fragment of the more general logic *CTL** [21], which combines both branching-time and linear-time operators. The branching time operators are **A** and **E**, which intuitively say "for every resp. some computation path", and the basic linear-time operators are **X** (*next time*), **U** (*until*), and **V** (*unless, releases*); further operators are derived from them.

Definition 2.1 Let A be a set of atomic propositions. *CTL* is the set of *state formulas on* A inductively defined as follows:

(1) any atomic proposition $a \in A$ is a state formula.

(2) if ϕ and ψ are state formulas, then $\neg\phi$, $\phi \vee \psi$, and $\phi \wedge \psi$ are state formulas;

(3) if ϕ and ψ are state formulas, then $\mathbf{X}\phi$, $\phi\mathbf{U}\psi$ and $\phi\mathbf{V}\psi$ are path formulas;

(4) if ϕ is a path formula, then $\mathbf{E}(\phi)$ and $\mathbf{A}(\phi)$ are state formulas.

Any formula ϕ which is formed only by (1)–(2) is called *pure state formula.* □

Intuitively, path formulas describe properties of paths because they use temporal operators next time, until and unless, looking forward in a computation path.

For a formal definition of the semantics of *CTL*, special *Kripke structures* are used. Informally, a Kripke structure consists of a labeled finite transition graph.

Definition 2.2 A *Kripke structure* is a quintuple $M = (A, S_0, S, R, L)$ such that:

- A is a finite set of atomic propositions;
- S is a finite set of states;
- $S_0 \subseteq S$ is a finite set of initial states;
- $R \subseteq S \times S$ is a transition relation;
- $L : S \to 2^A$ is a mapping assigning each state of S the set of atomic proposition true in that state; L is called *label function.*

Given a Kripke structure M, we denote by $A(M)$ its set of atomic propositions, by $S_0(M)$ the set of initial states, by $S(M)$ the set of states, by $R(M)$ the transition relation, and, finally, by $L(M)$ the label function. □

Starting from initial states, R generates the (infinite) computation *paths*.

Definition 2.3 A *path* π of a Kripke structure M is an infinite sequence $[s_0, s_1, \cdots, s_i, \cdots]$ such that for each $i \geq 0$ $(s_i, s_{i+1}) \in R$. Given an integer $i \geq 0$ and a path π we denote by $\pi(i)$ the i-th state of π. Given an integer $j \geq 0$ and a path π, the j-suffix π^j of π is the path $[\pi(j), \pi(j+1), \cdots]$ (clearly, $\pi = \pi^0$ and $\pi(i) = \pi^i(0)$). □

The semantics of CTL is defined through an entailment relation $\models$, which can be applied on states s and paths π for evaluating state or path formulas, respectively.

Definition 2.4 The entailment relation $\models$ for state and path formulas on a Kripke structure M is as follows (s and π is a generic state and path in M, respectively):

1. $M, s \models p$, if $p \in L(M)(s)$, for any atomic proposition $p \in A$
2. $M, s \models \neg\phi$, if $M, s \not\models \phi$ (ϕ is a state formula)
3. $M, s \models \phi_1 \vee \phi_2$, if $M, s \models \phi_1$ or $M, s \models \phi_2$ (ϕ_1, ϕ_2 are state formulas)
4. $M, s \models \phi_1 \wedge \phi_2$, if $M, s \models \phi_1$ and $M, s \models \phi_2$ (ϕ_1, ϕ_2 are state formulas)
5. $M, s \models \mathbf{E}(\psi)$, if there exists a path π with $\pi(0) = s$ such that $M, \pi \models \psi$
6. $M, s \models \mathbf{A}(\psi)$, if $M, \pi \models \psi$ for all paths π with $\pi(0) = s$
7. $M, \pi \models \phi$, if $M, \pi(0) \models \phi$ where ϕ is a state formula
8. $M, \pi \models \mathbf{X}\phi$, if $M, \pi^1 \models \phi$
9. $M, \pi \models \phi_1 \mathbf{U} \phi_2$, if there exists an integer $k \geq 0$ such that $M, \pi^k \models \phi_2$ and $M, \pi^j \models \phi_1$, for all $0 \leq j < k$
10. $M, \pi \models \phi_1 \mathbf{V} \phi_2$, if for every $k \geq 0$, $M, \pi^j \not\models \phi_1$ for all $0 \leq j < k$ implies $M, \pi^k \models \phi_2$

We write $M \models \phi$ if $M, s_0 \models \phi$, for every initial state $s_0 \in S_0(M)$, □

Intuitively, a state formula holds along a path, if it is true at its first state; $\phi_1 \mathbf{U} \phi_2$ is true, if ϕ_1 is true along the path until some stage is reached at which ϕ_2 is true; and $\phi_1 \mathbf{V} \phi_2$ is true, if there is no stage such that ϕ_2 is false and ϕ_1 is false at all previous previous stages. Note that $\mathbf{U}$ and $\mathbf{V}$ are dual operators: $\phi_1 \mathbf{U} \phi_2$ is true precisely if $\neg\phi_1 \mathbf{V} \neg\phi_2$ is false.

Two important additional operators, $\mathbf{F}$ (*finally*) and $\mathbf{G}$ (*globally*) are expressed through $\mathbf{U}$ and $\mathbf{V}$.

Definition 2.5 Given a state formula ϕ, the operators **F** and **G** are defined as follows:

- $\mathbf{F}\phi = true\mathbf{U}\phi$
- $\mathbf{G}\phi = false\mathbf{V}\phi \ (\equiv \neg\mathbf{F}\neg\phi)$

where *true* is a boolean tautology and $false$ is a boolean contradiction. □

Thus, coherent with the intuition, $M, \pi \models \mathbf{F}\phi$ means that there exists an integer $k \geq 0$ such that $M, \pi^k \models \phi$, while $M, \pi \models \mathbf{G}\phi$ means that for every $k \geq 0$, $M, \pi^k \models \phi$.

EX, **EG** and **EU** (or, dually, **AX**, **AF**, and **AV**) can be seen as basic time operators of *CTL*. The following equivalences are well-known (see e.g. [12]).

Proposition 2.1

$$
\begin{array}{rclrcl}
\mathbf{AX}\phi &=& \neg\mathbf{EX}(\neg\phi) & \mathbf{A}(\phi\mathbf{U}\psi) &=& \neg\mathbf{E}(\neg\psi\mathbf{U}(\neg\phi \wedge \neg\psi)) \wedge \neg\mathbf{EG}\neg\psi \\
\mathbf{AG}\phi &=& \neg\mathbf{EF}(\neg\phi) & \mathbf{A}(\phi\mathbf{V}\psi) &=& \neg\mathbf{E}(\neg\phi\mathbf{U}\neg\psi) \\
\mathbf{AF}\phi &=& \neg\mathbf{EG}(\neg\phi) & \neg\mathbf{A}(\neg\phi\mathbf{U}\neg\psi) &=& \mathbf{E}(\phi\mathbf{V}\psi)
\end{array}
$$

For modeling fair computations, *Kripke structure with fairness constraints* (*FC-Kripke structure*, for short) have been proposed.

Definition 2.6 An *FC-Kripke structure* M is an expansion of a Kripke Structure $K = (A, S_0, S, R, L)$ by a finite set F of *CTL* formulas, called *fairness constraints*, i.e, $M = (A, S_0, S, R, L, F)$. □

For any FC-Kripke structure M, we denote by $F(M)$ its set of fairness constraints; the others components of M are denoted as for ordinary Kripke structures.

The semantics of *CTL* formulas is adapted to fairness constraints by restricting the path quantifiers to those paths along which every fairness constraint holds infinitely often, which are called *fair paths*. More formally,

Definition 2.7 A path π in a FC-Kripke structure M is *fair*, if for every $\phi \in F$ and $i \geq 0$ there exists an integer $j \geq i$ such that $M, \pi(j) \models \phi$. □

Entailment of state and path formulas from an FC-Kripke structure M is defined analogous to entailment from a Kripke structure, with the only difference that path quantifiers **A** and **E** evaluate to "for all fair paths in M" and "there exists a fair path in M," respectively. Since the notion of entailment will be clear from the context, we will use for both entailment from a Kripke and a FC-Kripke structure the same symbol "$\models$."

3 Abductive Model Revision

In this section, we present an approach for changing a Kripke structure such that it satisfies a given formula ϕ. Our approach is in spirit of methods in the field of theory revision and abductive reasoning, and can be viewed as a semantical approach to changing a system which is represented by some Kripke model; typical such systems are concurrent programs and protocols. This semantical approach can be utilized as the underlying basis of an approach for change at the syntactical level of a system, i.e., its description in some formal specification language. This will be exemplified with the problem of repairing concurrent programs and protocols in the next section.

3.1 System Repair Problem

Given a *CTL* formula ϕ and an FC-Kripke structure M, a model checking technique can be applied for verifying whether ϕ is satisfied by M or not. Actually, *Symbolic Model Checking* [8, 36] gives as the result the set of states where ϕ holds or provides a counterexample, which outlines a case in which ϕ does not hold.

This is, in general, very useful to the protocol or circuit designer, because it aids him or her in understanding which part of the system fails, and is a clue for finding a modification of the system such that the specification, given by formula ϕ, holds. However, symbolic model checking does not provide any methods for *repairing* the system. That is, there is no component which suggests, given that the system does not satisfy the specification ϕ, a possible modification to the system upon which it satisfies ϕ. Clearly, such a component would be desirable in practice. Notice that the notion of repair (or therapy) in the context of model-based diagnosis was introduced in [23, 24] and independently in [41].

For this purpose, we formalize the *system repair problem* as a problem, given by an FC-Kripke structure M and a formula ϕ, whose solution consists of a set of modifications to the transition relation R (additions or deletions of tuples in R), such that ϕ is true in the modified system. As shown by examples later on, a solution of an SRP may give a useful clue of how to properly modify the system.

In what follows, we assume that $M = (A, S_0, S, R, L, F)$ is an FC-Kripke structure and ϕ is a *CTL* formula.

We start with the elementary concept of a *simple modification of* the transition relation, which is addition or a deletion of a state transition.

Definition 3.1 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 $R \setminus \{(s_1, s_2)\}$ otherwise. □

A *modification* of the system M is a *consistent* set of simple modifications of its transition relation R, where consistent means that no simultaneous addition and deletion of a given pair of states is allowed. More precisely,

Definition 3.2 A *modification for M* is a set Γ of simple modifications for R such that Γ contains no tuples $\langle (s_1', s_2'), +\rangle$, $\langle (s_1', s_2'), -\rangle$. Let $\Gamma^+ = \{(s_1, s_2) \mid \langle s_1, s_2, +\rangle \in \Gamma\}$ and $\Gamma^- = \{(s_1, s_2) \mid \langle s_1, s_2, -\rangle \in \Gamma\}$. The set of all modifications for M is denoted by $mod(M)$. The *result* $\Gamma(M)$ *of* Γ is the FC-Kripke structure $(A, S_0, S, R^\Gamma, L, F)$ where $R^\Gamma = \bigcap_{\delta\in\Gamma^-} \delta(R) \cup (\bigcup_{\delta\in\Gamma^+} \delta(R) \setminus R)$. □

Now we introduce the system repair problem. Intuitively, it represents the problem of finding a system modification Γ for M, such that $\Gamma(M)$ satisfies ϕ. In general, Γ must be implemented on a formal description of the system (i.e., the code of concurrent programs and protocols), and not every Γ might be actually feasible; therefore, we add a function $\mathcal{Y}$, which tells whether a particular modification Γ is admissible.

Definition 3.3 A *system repair problem (SRP)* is a triple $\mathcal{S} = \langle M, \phi, \mathcal{Y}\rangle$ where M is an FC-structure, ϕ is a formula on $A(M)$, and $\mathcal{Y}$ is a computable boolean function on $mod(M)$. Any modification Γ such that $\mathcal{Y}(\Gamma) = \mathbf{true}$ is called admissible. □

A SRP $\langle M, \phi, \mathcal{Y}\rangle$ is also called the *repair problem of M w.r.t. ϕ under $\mathcal{Y}$.* The admissibility function $\mathcal{Y}$ is domain-dependent; e.g., in case of a concurrent program, $\mathcal{Y}$ is derived from possible changes to the code of the processes.

A solution of an SRP states how the original system, which does presumably not satisfy ϕ, has to be modified by means of a set Γ of admissible modifications of its transition relation.

Definition 3.4 Given an SRP $\mathcal{S} = \langle M, \phi, \mathcal{Y}\rangle$, a *solution for $\mathcal{S}$* is an admissible modification Γ for M such that $\Gamma(M) \models \phi$. A solution Γ for $\mathcal{S}$ is *minimal*, if there exists no solution Γ' for $\mathcal{S}$ such that Γ' is properly contained in Γ. □

The restriction of arbitrary solutions to minimal ones is natural and implements Occam's principle of parsimony. In general, a solution preference could be used to select preferred solutions as customary in e.g. abductive reasoning, cf. [18].

Example 3.1 Consider the SRP $\mathcal{S} = \langle M, \phi, \mathcal{Y}\rangle$, where M is the FC-Kripke structure as obvious from Figure 2 with unique initial state s_0 and no fairness constraints, $\phi = \mathbf{AGAF}a$, and $\mathcal{Y} \equiv \mathbf{true}$, i.e., each modification Γ is admissible. (Note that $R(M) = \{(s_0, s_0), (s_0, s_1), (s_1, s_1)\}$.) It holds that $M \not\models \phi$: For the path $\pi = [s_0, s_1, s_1, \ldots]$, we have

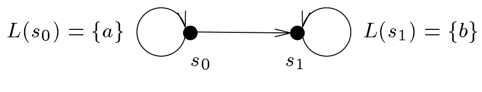

Figure 2: Labeled transition graph

$M, \pi \not\models \mathbf{GAF}a$, as it can be easily seen that $M, \pi^1 \models \mathbf{EG}\neg a$. A solution of $\mathcal{S}$ is the modification $\Gamma_1 = \{\langle s_0, s_1, -\rangle\}$, i.e., delete the transition from s_0 to s_1. Indeed, then $\pi = [s_0, s_0, \ldots]$ is the unique infinite path starting at an initial state, and a is true at each stage of π. Further solutions are e.g. $\Gamma_2 = \{\langle s_1, s_1, -\rangle, \langle s_1, s_0, +\rangle\}$ and $\Gamma_3 = \{\langle s_0, s_1, -\rangle, \langle s_0, s_0, -\rangle\}$. Notice that Γ_1 is minimal, while Γ_2 and Γ_3 are not. In fact, Γ_1 and $\Gamma_4 = \{\langle s_1, s_1, -\rangle\}$ are all minimal solutions of $\mathcal{S}$. $\Box$

Comparing system repair to problems in AI, it appears that an SRP is an interesting kind of *abductive model revision problem*, which involves both theory revision and abductive reasoning. From the definition, an SRP can neither be viewed as a pure theory revision problem (cf. [30, 25]), nor as a pure abductive reasoning task as e.g. in [40, 16, 31, 3, 19, 42, 4]; rather, it is a combined problem and can be best understood (and modeled) as an abductive theory revision problem [33, 29].

On the one hand, an SRP is similar to a theory revision problem. Indeed, the FC-Kripke structure M can be viewed as the set $Th(M)$ of all formulas true on it, and we have to revise this knowledge base by incorporating the formula ϕ (the specification) into it. Here, the revised knowledge base $M \circ \phi$ must amount to a modified FC-Kripke structure M' such that M' entails ϕ.

On the other hand, an SRP involves abductive reasoning: given M and ϕ, we want to find a particular modification $\Gamma \in mod(M)$ such that applying Γ to M yields a structure $M' = \Gamma(M)$ in which ϕ is true; thus, we *abduce* a solution Γ for the SRP in terms of changes to the transition relation. This can be formalized in a proper logical language. However, this does not mean an SRP is a genuine abductive reasoning task; in fact, abduction is applied in case of incomplete knowledge, which is in this view about the suitable changes for transforming M into M'. This is a somewhat unnatural state-oriented view, though, since like in planning, we proceed from one state (M) to another (M'), and the transition is specified in the domain theory using frame axioms.

3.2 System repair and abductive theory revision

More naturally, a SRP can be viewed as an abductive theory revision problem. We outline in the following how this is possible in the frameworks of [33] and [29].

SRP in Lobo and Uzcátegui's framework. In [33], the following scenario is considered. Given an abductive domain theory Σ, a knowledge base K, a formula ω (all in a finite language), and a revision operator $\circ$, a suitable formula γ is an *explanation* for ω (w.r.t. K, Σ and $\circ$), if the knowledge base $K \circ (\Sigma \wedge \gamma)$ entails ω. Suitability of γ means that γ is formed over a specified set Ab of abducible atoms. An explanation γ can be seen as a proper abductively inferred revision for incorporating ω into the knowledge base.

An SRP $\mathcal{S} = \langle M, \phi, \mathcal{Y}\rangle$ can be modeled in this framework as follows. It is possible to express the entailment problem $M \models \phi$ as an inference problem $K_M \models f(\phi)$ of a formula $f(\phi)$ from a knowledge base K_M in a suitable logic $\mathcal{L}$, where K_M describes M and $f(\phi)$ is a translation of ϕ into $\mathcal{L}$; e.g., $\mathcal{L}$ could be transitive closure logic if M is serial (i.e., each state has a successor) and no fairness constraints are present [28], or a similar extension of first-order logic with generalized quantifiers in the general case. Following [28] and similar translations of propositional modal logic into first-order logic, K_M can be constructed as a set of literals describing the components of M; the states are constants, ground literals $\pm R(s, s')$ describe the transition relation, and ground literals $\pm P_a(s)$ represent the label function, where $P_a(s)$ means that $a \in L(s)$, for each atom $a \in A(M)$.

Now choose Ab to be the set of all ground atoms on R, and let the domain theory Σ be empty. Furthermore, let us restrict acceptable abductive explanations to formulas γ which are conjunctions of literals.

Then, an acceptable abductive explanation γ for the formula $f(\phi)$ according to K_M and any reasonable revision operator $\circ$ (see [30, 25]), is a conjunction of ground literals on R. Any such γ corresponds to a set $\Gamma \in mod(M)$ as follows: For each positive literal $R(s, s')$ in γ, the tuple $\langle s, s', + \rangle$ is in Γ, and for each negative literal $\neg R(s, s')$ in γ, the tuple $\langle s, s', - \rangle$ is in Γ. If Γ is admissible, then it is a solution of $\mathcal{S}$; call γ admissible in this case. Vice versa, each solution Γ of the SRP $\mathcal{S}$ corresponds to an admissible explanation γ in the same way. Thus, there is a one-to-one logical correspondence between admissible explanations γ of $f(\phi)$ and solutions Γ of $\mathcal{S}$. In particular, modulo $\mathcal{Y}$ the minimal solutions of $\mathcal{S}$ correspond to the most general admissible explanations γ, i.e. any admissible explanation γ' with $\gamma \models \gamma'$ satisfies $\gamma' \models \gamma$. The disjunction of all these γ is an abductive explanation, provided $\circ$ satisfies some property [33], and amounts to the collection of all minimal solutions of $\mathcal{S}$.

Thus, an SRP can be modeled as (slightly constrained) abductive revision problem as described in [33].[1] We remark that in the above modeling, the domain theory Σ is empty. Of course, we could have set Σ to the part of K_M not involving R, but this would not make a difference. Moreover, in some cases the admissibility function $\mathcal{Y}$ can be easily expressed in the domain theory.

SRP in Inoue and Sakama's framework. In [29], an extended form of abduction is proposed, which is employed for an abductive framework of nonmonotonic theory change. The framework is detailed for autoepistemic logic, but it can be analogously based on other logics as well. In this approach, an abductive framework is a pair $\langle T, H \rangle$ of theories T and H, where T is the background theory (containing domain and factual knowledge) and H is a set of generic hypotheses; an explanation for a formula γ given $\langle T, H \rangle$ is a pair (I, O) such that I and O are instances of formulas in H, $(T \cup I) \setminus O$ logically entails γ, and $(T \cup I) \setminus O$ is consistent. An explanation (I, O) is minimal, if every explanation (I', O') with $I' \subseteq I$ and $O' \subseteq O$ is identical to (I, O).

The salient point in Inoue and Sakama's concept of abductive explanation is that formulas may also be removed from the background theory T, rather than only added. This is motivated by their observation that in a nonmonotonic context, it may be necessary to remove formulas from T in order to find an explanation for a formula γ.

An SRP $\mathcal{S} = \langle M, \phi, \mathcal{Y} \rangle$ can be modeled in Inoue and Sakama's framework as follows. As described above, the entailment problem $M \models \phi$ can be expressed as an inference problem $K_M \models f(\phi)$ in a suitable logic $\mathcal{L}$ (e.g., transitive closure logic). If we take K_M as background

[1] In fact, Lobo and Uzcátegui work in a finite propositional language; their framework can be extended for the slightly more general setting here.

theory T and the set $\{R(x,y), \neg R(x,y)\}$ as generic hypotheses H, then the solutions of $\mathcal{S}$ naturally correspond to the explanations of $\gamma = f(\phi)$ obtained from $\langle T, H\rangle$ as follows.

If Γ is a solution of $\mathcal{S}$, then the pair $(I_{\mathcal{S}}, O_{\mathcal{S}})$ is an explanation of γ, where

$$\begin{aligned} I_{\mathcal{S}} &= \{R(s,s') \mid (s,s') \in \Gamma^{+}\} \cup \{\neg R(s,s') \mid (s,s') \in \Gamma^{-}\}, \\ O_{\mathcal{S}} &= \{\neg R(s,s') \mid (s,s') \in \Gamma^{+}\} \cup \{R(s,s') \mid (s,s') \in \Gamma^{-}\}. \end{aligned}$$

On the other hand, if (I,O) is an explanation of γ, then the set $\Gamma_{(I,O)} = \{\langle s, s', +\rangle \mid R(s,s') \in I \setminus O\} \cup \{\langle s, s', -\rangle \mid \neg R(s,s') \in I \setminus O\}$ is a solution of $\mathcal{S}$, provided $\Gamma_{(I,O)}$ is admissible in terms of $\mathcal{Y}$. Notice that this establishes a natural one-to-one logical correspondence between minimal solutions and minimal admissible explanations, where the admissible explanations (I,O) are those such that $\Gamma_{(I,O)}$ is admissible. As in the previous case, the admissibility function $\mathcal{Y}$ may be expressed in some cases in the background theory T.

4 Repair of Concurrent Programs

In this section, we apply the model from above to concurrent programs and protocols. A concurrent program consists of a number of processes running in parallel. In [39], propositional temporal logic is used for representing specifications for concurrent programs and for defining a technique to verify such a specification. We adopt Pnueli's model [39] for representing concurrent programs, in order to have a clear and synthetic way of representing the semantics of concurrent programs. We then define the FC-Kripke structure of a concurrent program under the assumption that shared variables have a boolean domain. The transition relation is defined referring to an *asynchronous model* with *interleaving* [36]. Specifications for the program are described by using both *CTL* formulas and fairness constraints. Fairness constraints allow to express properties that must be verified infinitely often along paths (which are not expressible in *CTL*). Such properties are frequently required in the context of concurrency, e.g., for the fair scheduling of processes.

Consider then the FC-Kripke structure of a concurrent program $\mathcal{P}$; denote this structure by $M_F(\mathcal{P})$. Let ϕ be a *CTL* formula comprising all specifications required for $\mathcal{P}$. Suppose that ϕ is not satisfied, that is $M_F(\mathcal{P}) \not\models \phi$. The problem of modifying the original program in order to make ϕ true is nontrivial.

In this section, we deal with this issue under particular assumptions about error types. In particular, we address the cases in which errors are faulty assignments and disordered successive assignments. We show that this problem has a direct mapping into the system repair problem from the previous section. Determining a program repair and optimization issues will be dealt with in subsequent sections.

4.1 Concurrent programs and FC-Kripke structures

A concurrent program $\mathcal{P}$ is composed of a finite collection $P_1, \ldots, P_n$ of processes running in parallel. We refer to a model with shared memory; hence, all variables $x_1, x_2, \ldots, x_l$ in $\mathcal{P}$, which we denote by $\mathbf{x}$, are accessible to all processes.

According to Pnueli's model [39], each process P_i can be represented by a directed labeled graph $G(P_i) = \langle N_i, E_i \rangle$, the *graph of* P_i, where N_i is the set of nodes and E_i is the set of labeled directed edges.

The nodes N_i are an initial segment $1, 2, \ldots, m_i$ of the integers. They represent the break points in the code of P_i, which are the points before the code and between successive statements in ascending order. We denote by $stmnt^{\mathcal{P}}(i, b)$ the statement immediately after break point b in the code of P_i; if clear from the context, $\mathcal{P}$ and i are omitted. The code between two subsequent break points is considered atomic in the parallel execution.

We suppose that the following types of statements are available:

(1) empty statement, denoted by ϵ ;

(2) assignment statement: $x_i := g(\mathbf{x})$, where $g(\mathbf{x})$ is an expression over variables in $\mathbf{x}$ compatible with the type of x_i;

(3) jump statement: **goto** *break_point*;

(4) conditional statement: **if** c($\mathbf{x}$) **then** *stmnt*, where $c(\mathbf{x})$ is a boolean expression over variables in $\mathbf{x}$; and,

(5) compound statement: **begin** *stmnt-1*; … *stmnt-n* **end**.

In order to assure that each process P_i is nonterminating (that is usual in this framework), we assume that the last statement of P_i is an unconditional jump. An infinite loop b : **goto** b can easily be added at the end of a process.

The arcs E_i correspond to the possible execution flow of the program. From every break point b, there is at least one arc leading to another break point, depending on the type of $stmnt(b)$. Moreover, each arc a is labeled with a command $l(a)$, which is a pair $(c(\mathbf{x}), stmnt)$ of a boolean condition $c(\mathbf{x})$ (the *guard*) and a statement *stmnt*, which is either ϵ or an assignment.

- if $stmnt(b)$ is ϵ or an assignment, an arc $b \to b{+}1$ is present, labeled with $(\mathbf{true}, stmnt(b))$;
- If $stmnt(b)$ is a jump statement **goto** p, then an arc $b \to p$ is present, labeled with $(\mathbf{true}, \epsilon)$;
- If $stmnt(b)$ is a conditional statement **if** $c(\mathbf{x})$ **then** p: *stmnt*, where p is a break point, then arcs $b \to p$ and $b \to q$ are present, where q is the first break point after *stmnt* (note that $p = b + 1$). The labels of the arcs are $l(b, p) = (c(\mathbf{x}), \epsilon)$ and $l(b, q) = (\neg c(\mathbf{x})), \epsilon)$, respectively;
- If $stmnt(b)$ is a compound statement **begin** *stmnt-1*; b_2: *stmnt-2*; … b_n:*stmnt-n* **end**, then consider b_1: *stmnt-1*; … b_n: *stmnt-n*, where *stmnt-1* inherits its break point $b_1 = b$ from $stmnt(b)$.

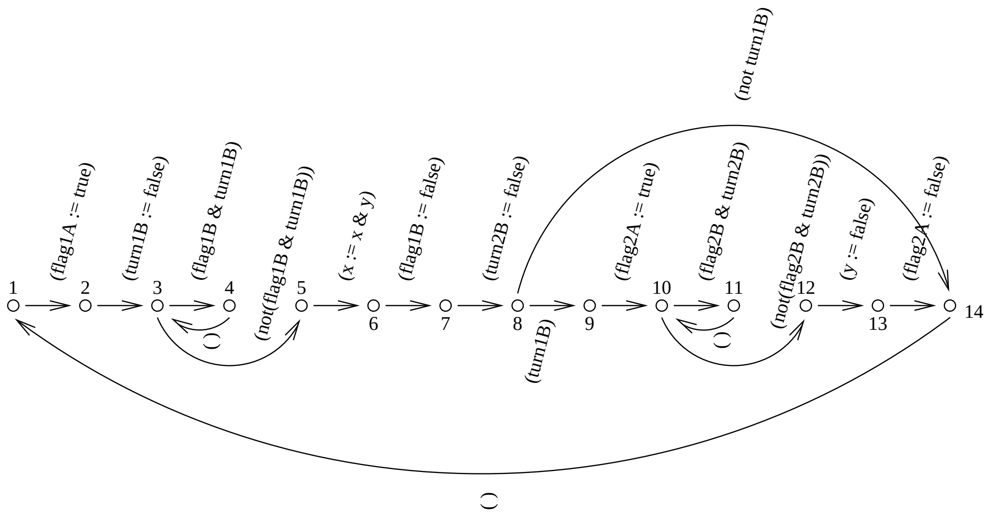

Figure 3: Graph $G(P_A)$ for the process P_A from Figure 1

Thus, more than one arc may leave from a node in the graph, reflecting the different execution paths of a process. For convenience, we sometimes omit **true** and ϵ in commands; in particular, () is $(\mathbf{true}, \epsilon)$. An example of a process graph is shown in Figure 3.

Remark. The above language for programs is elementary, but can be easily enriched by further types of statements. E.g., conditional statements with else-branches of the form "**if** $c(\mathbf{x})$ **then** *stmnt-1* **else** *stmnt-2*" or while-loops "**while** $c(\mathbf{x})$ **do** *stmnt*" may be added. Such statements are straightforward translated into equivalent sequences of elementary statements using jump-statements as usual. The graph of a process in such a syntactically enriched language is then given by the graph of the transformed program. Alternatively, the process graph could be defined directly, without resorting to a low level transformation; however, the definition becomes more involved. In order to keep the treatment simple, we refrain from explicitly considering an enriched syntax.

If the program contains also synchronization primitives such as semaphores, the synchronization statement can be easily represented by using guards. For instance, if s is a semaphore, the wait primitive $P(s)$ corresponds to the command $(s > 0, s := s - 1)$, and the signal primitive $V(s)$ corresponds to $(\mathbf{true}, s := s + 1)$.

Note that for this kind of statement, the disjunction between guards of all arcs leaving a node, i.e., the exit condition from this node, is not necessarily true (different from the previous statements); because of synchronization statements, a process can remain trapped in a node until some condition is verified. $\square$

Now we provide a formal representation of the execution of a concurrent program. Intuitively, the execution of a single process corresponds to a traversal of the process graph driven by the result of the evaluation of the guards and involving all the actions (assignments) defined in the commands labeling arcs. The concurrent execution can be represented as the interleaved execution of all processes under the assumption that any single command is atomic. If we want to faithfully model a possible interference between the fetching and storing of operands, we may have to replace an assignment statement by a chain of assignments [39]. For example, an assignment $x := x_1 \vee x_2$, where the evaluation of $x_1 \vee x_2$ should be not atomic, can be modeled by $z_1 := x_1$, $z_2 := x_2$; $x := z_1 \vee z_2$, where z_1 and z_2 are temporary variables and evaluating $z_1 \vee z_2$ is atomic (see [39] for a detailed discussion on this topic).

Formally, a concurrent program $\mathcal{P} = P_1, \ldots, P_n$ can be modeled using FC-Kripke structures as follows. Let $\mathbf{x} = x_1, \ldots, x_p$ be the variables of $\mathcal{P}$, which range over the domains $D_1, \ldots, D_p$, resp. For each process P_i, let $N_i = \{1, 2, \ldots, m_i\}$ be the set of its break points.

Note that specifications for the program are expressed both by *CTL* formulas and by fairness constraints.

In what follows, we assume that all variables x_i are Boolean. Clearly, the general case of finite domains D_i can be modeled with Boolean variables. For each x_i, an atomic proposition x_i^d for each value $d \in D_i$ can be used, corresponding to "$x_i = d$;" alternatively, $\lceil \log |D_i| \rceil$ many Boolean variables $x_{i,j}$ allow for a binary representation of the value of x_i. As observed in [11], automatic techniques of model checking can be applied only in the case of small finite domains, due to the resulting size of the state space. E.g., in programs which encode concurrent protocols, variables usually represent control flags, switches, and similar objects whose values are from small discrete domains. Hence, such programs are suitable candidates for verification and repair as developed in the sequel.

Definition 4.1 Let $\mathcal{P}$ be a program. The FC-Kripke structure $M_F(\mathcal{P}) = (A, S, S_0, R, L, F)$, (the *Kripke structure* of $\mathcal{P}$, if F is understood), is as follows:

– *set of atomic propositions A.* $A = \mathit{Vars} \cup B \cup E$, where

- $\mathit{Vars} = \{x_1, \ldots, x_n\}$ is the set of all variables in $\mathcal{P}$.
- $B = \{b_i^k \mid 1 \leq i \leq n, 1 \leq k \leq m_i\}$. For each process P_i and possible break point $k \in \{1, \ldots, m_i\}$ of P_i, an atomic proposition b_i^k exists, which is intuitively true if P_i is currently at break point k.
- $E = \{e_1, \ldots, e_n\}$. The atomic proposition e_i tells whether process i was executed in the latest execution step in the system.

– *set of states S.* $S = N_1 \times \cdots \times N_n \times D_1 \times \cdots \times D_p \times \{1, \ldots, n\}$. Hence, a state is a tuple of break points (one for each process, i.e., a state of its execution flow), an assignment to the program variables, and process number. As defined later, the process number tells which process was executed in the last step before reaching this state.

Given a state $s \in S$, we denote by $\mathbf{b}(s)$ the n-tuple of break points appearing in s, by $\mathbf{v}(s)$ the p-tuple of values of variables appearing in s, and by $\mathbf{ex}(s)$ the last component of s. Given a tuple t, we denote by t_i the i-th element of t. Hence, $\mathbf{b}(s)_i$ is the break point of P_i at s, and $\mathbf{v}(s)_i$ is the value of variable x_i at s.

– *set of initial states* S_0. The set S_0 contains only states s such that $\mathbf{v}(s)_i = 1$, for all i, i.e., all processes are at their entry break points, and $\mathbf{ex}(s) = 1$. (We arbitrarily choose process number 1, since it is, by the meaning of $\mathbf{ex}(s)$, not relevant.)

– *transition relation* R. Intuitively, at state s_1, one of the processes P_k is enabled for traversing the next arc in $G(P_k)$ in the computation.

A pair $(s_1, s_2) \in S \times S$ belongs to R iff:

- there exists an arc a in $G(P_{\mathbf{ex}(s_2)})$ directed from break point $\mathbf{b}(s_1)_{\mathbf{ex}(s_2)}$ to break point $\mathbf{b}(s_2)_{\mathbf{ex}(s_2)}$;
- $\mathbf{b}(s_1)_i = \mathbf{b}(s_2)_i$, for each $1 \leq i \leq n$, $i \neq \mathbf{ex}(s_2)$;
- for the label $l(a) = (c(\mathbf{x}), stmnt)$ of arc a, $c(\mathbf{v}(s_1))$ is true and (i) if $stmnt = \epsilon$, then $\mathbf{v}(s_1) = \mathbf{v}(s_2)$; (ii) if $stmnt$ is an assignment $x_i := g(\mathbf{x})$, then $\mathbf{v}(s_2)_j = \mathbf{v}(s_1)_j$, for all $j \neq i$, and $\mathbf{v}(s_2)_i = g(\mathbf{v}(s_1))$; denote this by $\mathbf{v}(s_2) = g(\mathbf{v}(s_1))$.

We say that (s_1, s_2) *executes* the command $(c(\mathbf{x}), stmnt)$, and that s_2 is reached from s_1 *by executing* $stmnt(\mathbf{ex}(s_2), \mathbf{b}(s_1)_{\mathbf{ex}(s_2)})$; notice that for each P_k, a unique s_2 is reached from s_1 by executing $stmnt(k, \mathbf{b}(s_1)_k)$.[2]

– *label function* L. The label function associates with every state s the set of atomic propositions

$$L(s) = L_{\mathbf{x}}(s) \cup L_{\mathbf{b}}(s) \cup L_{\mathbf{ex}}(s),$$

where $L_{\mathbf{x}}(s) = \{x_i \in \mathbf{x}, \mid \mathbf{v}(s)_i = \mathbf{true}\}$, $L_{\mathbf{b}}(s) = \{b_i^{\mathbf{v}(s)_i} \in B \mid 1 \leq n\}$, and $L_{\mathbf{ex}}(s) = \{e_{ex(s_2)}\} \subseteq E$.

– *fairness constraints* F. F is a set of fairness constraints. □

Example 4.1 Let us reconsider program $\mathcal{P}$ in Figure 1 and see how an FC-Kripke structure $\mathcal{M}_F(\mathcal{P})$ for it looks like.

The set of states is $S = \{1, \ldots, 14\} \times \{1, \ldots, 14\} \times \{true, false\}^8 \times \{1, 2\}$, since both P_A and P_B have 14 break points, there are eight variables x_i in $\mathbf{x}$ (x, y, $flagiV$ and $turniB$, where $i = 1, 2$ and $V = A, B$), and two process numbers ($A = 1$ and $B = 2$, say). Thus, $|S| = 14^2 \cdot 2^8 \cdot 2 = 100,352$, i.e., there are roughly 10^5 states; among these, there are $2^8 = 256$ potential initial states. If we assume that all program variables are false at the beginning of a computation, there is a unique initial state.

The set A contains $8 + 2 \cdot 14 + 2 = 38$ atomic propositions: the eight variables of $\mathbf{x}$, the 28 variables b_i^k, and two variables e_A and e_B.

The transition relation R and the label function L can be readily defined from this and the process graphs for P_A (cf. Figure 3) and P_B. For example, the pair (s_1, s_2) is in R, where s_1 is such that $\mathbf{v}(s_1) = (7, 8)$, all variables are false, and $\mathbf{ex}(s_1) = B$, and s_2 is such that

[2] If semaphores would be allowed, no such s_2 might exist for P_k, which means that $stmnt(m, k)$ is not executable at s_1.

$\mathbf{v}(s_2) = (8, 8)$, all variables except turn2B are false, and $\mathbf{ex}(s_2) = A$. The labels of the states s_1 and s_2 are $L(s_1) = \{b_1^7, b_2^8, e_B\}$ and $L(s_2) = \{b_1^8, b_2^8, turn2B, e_A\}$, respectively.

Plausible fairness constraints for $\mathcal{P}$ are $F = \{e_A, e_B\}$, which guarantee fair scheduling. (Recall that a path π satisfies a fairness constraint ϕ iff ϕ is true infinitely often on that path. For instance, because of e_A process A will be scheduled infinitely often.)

In the rest of our running example, we adopt this set of constraints for $\mathcal{P}$. □

4.2 The repair problem

In this section, we consider the problem of repairing a concurrent program $\mathcal{P} = P_1, \ldots, P_n$. We assume that $\mathcal{P}$ with its variables $\mathbf{x}$, a set of fairness constraints F (pure state formulas), the Kripke structure $M_F(\mathcal{P})$, and a CTL formula ϕ on $A(M_F(\mathcal{P}))$ are given.

The correctness of program $\mathcal{P}$ refers to $M_F(\mathcal{P})$, where the formulas in F encode assumptions on the program execution; "unfair" computation paths are excluded.

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

Example 4.2 (continued) For $\mathcal{P}$ in Figure 1, define

$$\begin{array}{rrcl} & \phi & = & \phi_{flags} \wedge \phi_{crit}, \\ \text{where} & \phi_{flags} & = & \displaystyle\bigwedge_{i=1,2,\, V=A,B} \mathbf{AG}(flagiV \rightarrow \mathbf{AF}\neg flagiV), \\ & \phi_{crit} & = & \mathbf{AG}(\neg(b_1^{12} \wedge b_2^{10})) \wedge \mathbf{AG}(\neg(b_1^5 \wedge (b_2^5 \vee b_2^{11}))). \end{array}$$

Informally, ϕ says that in every computation, P_V must eventually exit the critical section i after entering it, and that the two processes cannot be simultaneously in a critical section. (E.g., $\mathbf{AG}(\neg(b_1^{12} \wedge b_2^{10}))$ in ϕ_{crit} requires that the processes A and B do not execute the assignments to the variable y, namely instruction 12 of A and instruction 10 of B, at the same time.)
For $F = \{e_A, e_B\}$ (fair scheduling), $M_F(\mathcal{P}) \not\models \phi$; this is not immediate. □

Since we assumed that errors are present in terms of incorrect assignments or assignments in wrong order, a solution of the program repair problem will be a sequence of assignment modifications and assignment interchanges. An assignment modification may affect each side of an assignment. In particular, it will either

- replace the right hand side of an assignment by a constant (**true** or **false**), or
- replace the variable on the left hand side of an assignment by another one.

Next, we will formally define the notion of program correction. To this end, we introduce some preliminary notation.

Definition 4.2 For any assignment statement χ, we denote by $var(\chi)$ the variable of the left hand side of χ and by $expr(\chi)$ the expression of the right hand side of χ.

An assignment χ is called *simple* if either $expr(\chi) = \textbf{true}$ or $expr(\chi) = \textbf{false}$. □

The next definition formalizes the notion of single program correction, which is an atomic change to the program. A (general) program correction will be then obtained as a sequence of single corrections (which are applied one by one in the specified order).

Definition 4.3 A *single (program) correction* for $\mathcal{P}$ is a 3-tuple $\langle k, b, \gamma \rangle$, where $k \in \{1, \dots, n\}$ is a process number, $b \in N_k$ is a break point of the process P_k, and $\gamma \in \{swap, r\text{-}change\} \cup \{l\text{-}change(x_j) \mid x_j \in \mathbf{x}\}$, is a modification such that the following holds:

- if $\gamma = r\text{-}change$, i.e., it is a right side modification, then $stmnt(k, b)$ is a simple assignment;
- if $\gamma = l\text{-}change(x_j)$, for some variable $x_j \in \mathbf{x}$, i.e., a left side modification, then $stmnt(k, b)$ is an assignment with $var(stmnt(k, b)) \neq x_j$; and
- if $\gamma = swap$, i.e., an assignment interchange, then both $stmnt(k, b)$ and $stmnt(k, b+1)$ are assignments.

The *modification* of $\mathcal{P}$ by a single correction $\alpha = \langle k, b, \gamma \rangle$, denoted by $\mathcal{P}^\alpha$, is the concurrent program obtained from $\mathcal{P}$ by changing the code of process P_k in the following way:

- if $\gamma = r\text{-}change$ (i.e., it is a right side modification), the assignment $stmnt^{\mathcal{P}}(k, b)$ is replaced by the assignment $var(stmnt^{\mathcal{P}}(k, b)) := not(expr(stmnt^{\mathcal{P}}(k, b)))$;
- if $\gamma = l\text{-}change(x_j)$ (i.e., it is a left side modification with variable x_j), the assignment $stmnt^{\mathcal{P}}(k, b)$ is replaced by the assignment $x_j := expr(stmnt^{\mathcal{P}}(k, b))$;
- if $\gamma = swap$ (i.e., it is an interchange modification), the assignment $stmnt^{\mathcal{P}}(k, b)$ is replaced by the assignment $stmnt^{\mathcal{P}}(k, b + 1)$ and the assignment $stmnt^{\mathcal{P}}(k, b + 1)$ is replaced by the assignment $stmnt^{\mathcal{P}}(k, b)$. □

Some remarks about the kinds of corrections we consider here are in order. In general, a change on the right hand side of an assignment may involve a new variable or any expression; in lack of any information about which of those changes are meaningful in a particular context, and considering the in general tremendously large number of functionally different expressions, we do not consider such changes here. However, our framework could be extended to handle such modifications as well.

Furthermore, a disordering of two statements which are not assignments seems to be a programming error which is less frequent in practice; moreover, considering respective corrections is more involved and restricts the use of optimization techniques we develop later. Therefore, we do not consider the interchange of arbitrary statements.

Example 4.3 The modification of the program $\mathcal{P}$ of Figure 1, with the single program correction $\alpha_1 = \langle 2, 12, \textit{l-change}(flag1A)\rangle$ is the program obtained from $\mathcal{P}$ by replacing the statement 12 of Process B by the assignment $flag1A := false$. The 3-tuple $\langle 1, 2, swap\rangle$ is not a correction for $\mathcal{P}$, since the statement 3 of P_A is not an assignment. □

A complex correction for $\mathcal{P}$ is a sequence of single corrections. The modification of $\mathcal{P}$ by a complex correction is obtained by applying the single corrections in order.

Definition 4.4 Let $\overline{\alpha} = \alpha_1 \cdots \alpha_q$ be a sequence of single corrections for $\mathcal{P}$. Then, the modification $\mathcal{P}^{\overline{\alpha}}$ of $\mathcal{P}$ by the complex correction $\overline{\alpha}$ is recursively defined as the program $\mathcal{P}^{\alpha_1 \cdots \alpha_q} = (\mathcal{P}^{\alpha_1 \cdots \alpha_{q-1}})^{\alpha_q}$, if $q \geq 1$, and as $\mathcal{P}^{\emptyset} = \mathcal{P}$ if $q = 0$. The *length* of the correction $\overline{\alpha}$, denoted $length(\overline{\alpha})$, is the number q of single corrections in it. □

Note that a modification of $\mathcal{P}$ by a single correction $\langle k, b, \gamma\rangle$ affects the code of process P_k. It induces a change of the graph $G(P_k)$ and, as a consequence, of the FC-Kripke structure associated with $\mathcal{P}$. In case of an assignment correction, the only change in $G(P_k)$ is the label corresponding to the assignment $stmnt(b)$. In case of an assignment interchange, both labels corresponding to $stmnt(b)$ and $stmnt(b+1)$ are changed. This merely affects the transition relation R of $M_F(\mathcal{P})$. The following proposition is therefore easily derived by an inductive argument.

Proposition 4.1 *For any correction $\overline{\alpha}$, the FC-Kripke structures $M_F(\mathcal{P})$ and $M_F(\mathcal{P}^{\overline{\alpha}})$ coincide on A, S_0, S, L, and F.*

(Recall that fairness constraints are fixed.) Since a correction $\overline{\alpha}$ on $\mathcal{P}$ only affects R, $\overline{\alpha}$ can be viewed, according to Definition 3.2, as a modification of the system $M_F(\mathcal{P})$; there exists a modification Γ for $M_F(\mathcal{P})$ such that $R^{\overline{\alpha}} = R^{\Gamma}$. On the other hand, given a modification Γ of $M_F(\mathcal{P})$, in some cases R^{Γ} can be obtained by a correction $\overline{\alpha}$ and considering the new transition relation $R^{\overline{\alpha}}$.

Definition 4.5 A correction $\overline{\alpha}$ for $\mathcal{P}$ *induces* a modification $\Gamma \in mod(M_F(\mathcal{P}))$, if $R^{\overline{\alpha}} = R^{\Gamma}$.
Let $\mathcal{Y}_{\mathcal{P}} : mod(M_F(\mathcal{P})) \longrightarrow \{\mathbf{true}, \mathbf{false}\}$ be the boolean function such that for every $\Gamma \in mod(M_F(\mathcal{P}))$, we have $\mathcal{Y}_{\mathcal{P}}(\Gamma) = \mathbf{true}$ if there exists a correction $\overline{\alpha}$ for $\mathcal{P}$ such that $\overline{\alpha}$ induces Γ, and $\mathcal{Y}_{\mathcal{P}}(\Gamma) = \mathbf{false}$ otherwise. □

Note that, by our assumptions, the function $\mathcal{Y}_{\mathcal{P}}$ is clearly computable.
Next we define a repair problem $\mathcal{P}$ and show how it can be solved in terms of the abductive solution of a SRP from the previous section.

Definition 4.6 A *program repair problem (PRP)* is a triple $\mathcal{R} = \langle \mathcal{P}, F, \phi\rangle$, where $\mathcal{P}$ is a concurrent program, F is a set of fairness constraints on $A(M(\mathcal{P}))$, and ϕ is a formula on $A(M(\mathcal{P}))$. □

A PRP $\mathcal{R} = \langle \mathcal{P}, F, \phi\rangle$ is also called the *program repair problem for $\mathcal{P}$ w.r.t. ϕ under F*. The solution of a PRP can be given in terms of a solution to a SRP as follows.

Definition 4.7 Given a PRP $\mathcal{R} = \langle \mathcal{P}, F, \phi \rangle$, a *solution for* $\mathcal{R}$ is a solution Γ for the SRP $\mathcal{S} = \langle M_F(\mathcal{P}), \phi, \mathcal{Y}_{\mathcal{P}} \rangle$. A *repair* for $\mathcal{R}$ is any correction $\overline{\alpha}$ for $\mathcal{P}$ that induces a solution Γ for $\mathcal{R}$. □

The following proposition is immediate from Definitions 3.4 and 4.7.

Proposition 4.2 *For any PRP problem* $D = \langle \mathcal{P}, F, \phi \rangle$*, a program correction* $\overline{\alpha}$ *is a repair for* D *if and only if* $M_F(\mathcal{P}^{\overline{\alpha}}) \models \phi$.

The process of finding an abductive solution is commonly guided by some rationality principle which aims at pruning solutions that are less plausible. In particular, following Occam's principle of parsimony, solutions are pruned which are not free of redundancy, i.e., if it is possible to remove hypotheses while preserving the solution property.

This criterion is often strengthened by further restricting the preferred solutions to those which have a minimal cost (or, dually, a highest probability); if no cost information is available, solutions containing a smallest number of hypotheses are often selected.

Following this lead, we adopt as rationality criterion the length of a complex correction.

Definition 4.8 Given a PRP $\mathcal{R} = \langle \mathcal{P}, F, \phi \rangle$, a repair $\overline{\alpha}$ for $\mathcal{R}$ is *minimal*, if there is no repair $\overline{\beta}$ for $\mathcal{R}$ such that $length(\overline{\beta}) < length(\overline{\alpha})$. □

Of course, alternative notions of minimal repair could be acceptable. For example, if we view the PRP as the underlying system repair problem, we could have accepted those corrections $\overline{\alpha}$ as minimal repairs such that the system modification Γ induced by $\overline{\alpha}$ is minimal in the sense of Definition 3.4. This would require, however, that the user is aware of the formal representation of programs by means of FC-Kripke structures, in order to interpret minimality of repairs in the right terms; moreover, statements in a program are the atomic units of the user language, which suggests that repairs should be understood on the basis of this language. And, finally, notice that the semantical level, some complex repair might be minimal, while it is not at the syntactical level (e.g., if it contains a sequence of operations which cancel in their effects). Therefore, adopting the notion of minimal repair as in Definition 4.8 is preferable in this particular application.

Example 4.4 Consider the PRP $\mathcal{R} = \langle \mathcal{P}, F, \phi \rangle$, where $\mathcal{P}$ is the program of the running example (Figure 1) and ϕ, F are as in Example 4.2. As pointed out in Example 4.2, $M_F(\mathcal{P}) \not\models \phi$. It is possible to verify that, in this case, there is an error occurring in the second statement of process P_A. Indeed, it should read $turn1B := true$. Hence, the single correction $\alpha = \langle 1, 2, r\text{-}change \rangle$ is a repair for D. Clearly, since $length(\alpha) = 1$, it is also minimal. □

Regarding complex corrections, we note that the case of a single error in a program is very relevant in practice. It is frequent and is usually examined first by human trouble shooters as well as many diagnostic systems. And, many such systems are not capable of handling complex errors at all. In the case of a single error, a minimal repair will be a sequence $\overline{\alpha} = \alpha_1$ of length 1, since a single program correction will be sufficient to fix the single error in the program. Hence, we focus with respect to practice on such minimal repairs. Our results for optimization in finding a repair are tailored for this case; they hold, suitably generalized, in the context of complex repairs, but are naturally weaker in effect.

5 Repairs and Counterexamples

After having defined the concepts of repair problem and solution, the upcoming issue is how to find a solution for a given problem; a suitable algorithm for this task is desired.

There is a simple brute force algorithm for solving the repair problem: check for each possible correction $\overline{\alpha}$ whether it is a repair, i.e., whether $M_F(\mathcal{P}^{\overline{\alpha}}) \models \phi$, in a systematic enumeration of all possible corrections until such a $\overline{\alpha}$ is found; by respecting cancellation effects of single corrections in complex corrections, the search space is finite.

Clearly, this algorithm is inefficient in general. Even if we restrict to single correction solution candidates $\overline{\alpha} = \alpha_1$, quite a number of different tests may have to be made until the desired answer is obtained.

In order to reduce the number of cases that have to be considered, we develop techniques which restrict the search space by exploiting counterexamples. Informally, a counterexample for a PRP $\mathcal{R} = \langle \mathcal{P}, F, \phi \rangle$ is a portion of a branching computation tree which witnesses that ϕ fails. Given a counterexample, our technique identifies corrections α under which the counterexample is invariant, i.e., still apply if α is implemented. Such α's are useless as corrections and can be discarded. It happens that this way, the space of candidate repairs may be drastically reduced.

Our concept of counterexample extends the one presented in [11] for the purpose of symbolic model checking in the logic $ACTL$, which is a fragment of CTL. There, a procedure for counterexample construction is outlined (see also [14]), which returns as a result a single path in M. This path is in general not a counterexample per se, but rather a heuristically selected path from a counterexample tree which gives some intuition why the formula fails. The main reason for restricting the return value to a single path is that understanding counterexample trees (and to represent them graphically) seems to be difficult.[3]

As shown below, there are simple formulas in $ACTL$ for which no single path is a counterexample. This is due to the possible presence of nested path quantifiers and disjunction in a formula ϕ. A single path from the counterexample tree, as returned by the symbolic model checking procedure, or even the full counterexample tree, if it is represented in the customary way, might not be much instructive why a formula fails. The reason is that important structural information about the paths in a simple tree representation is missing, namely how they are nested.

For our purposes, a full counterexample tree is needed, because otherwise a repair of the program may not be found. In order to overcome the representational problem with counterexample trees, we introduce the concept of a multi-path, which allows for a structured representation of paths. Counterexamples are then particular multi-paths. Our formalization of counterexamples seems to be an appropriate extension of path counterexamples as described in [11]. In fact, while an $ACTL$ formula may lack a path counterexample, there always exists a multi-path counterexample if it fails on a Kripke structure.

In the remainder of this paper, we restrict our attention to the fragment $ACTL$ of CTL. In this fragment, only universal path quantifiers are allowed, and negation is restricted to pure

[3] K. McMillan, personal communication.

state formulas.[4] For instance, the specification ϕ in our running example (see Example 4.2) is an $ACTL$ formula; however, the formula $\psi = \mathbf{E}\,\mathbf{F}(turn1B \wedge \neg turn2B)$, which states that there is some computation such that $turn1B$ is true and $turn2B$ false at some point, is not an $ACTL$ formula. Notice that $ACTL$ is considered to be an important and highly relevant fragment of CTL, as it allows for *abstraction* and *compositional reasoning* [13, 26]. Moreover, we assume in the following that fairness constraints in FC-Kripke structures are pure state formulas.

5.1 Counterexamples

Informally, a multi-path represents an infinite tree T, by using a vertical axis rather than the usual recursion from a node to its descendants. The branches of T are infinite paths; the axis is a distinguished main path of the tree, from which other paths spring off. These paths are main paths of subtrees of T. This view gives rise to an inductive definition of multi-paths. The main advantage of this concept is preservation of the nesting of paths, which is lost in the usual tree definition. Moreover, for a class of multi-paths which is sufficient for our purposes, effective finite representations exist.

Preliminary to the formal definition of multi-paths, we introduce multi-sequences.

Definition 5.1 Let S be the set of states. Then,

- for every state $s \in S$, $\Pi = s$ is a finite multi-sequence in S;
- if $\Pi_0, \Pi_1, \ldots$ are countably infinite many multi-sequences in S, then $\Pi = [\Pi_0, \Pi_1, \ldots]$ is a multi-sequence in S.

For any multi-sequence Π, its i-th element is denoted by $\Pi(i)$, for all $i \geq 0$; moreover,

$$\begin{aligned}
\text{its } \textit{depth} \text{ is } d(\Pi) &= \begin{cases} 0, & \text{if } \Pi = s, \\ \sup_{i\geq 0} d(\Pi(i)) + 1, & \text{otherwise,} \end{cases} \\
\text{and its } \textit{origin} \text{ is } or(\Pi) &= \begin{cases} \Pi(0), & \text{if } \Pi = s, \\ or(\Pi(0)), & \text{otherwise.} \end{cases}
\end{aligned}$$

□

Next we introduce the notion of *main sequence* of a multi-sequence. Informally, it is the sequence formed by the origins of all elements in a multi-sequence.

Definition 5.2 Given a multi-sequence Π, the *main sequence of* Π, denoted by $\mu(\Pi)$, is

- s, if $\Pi = s$ is finite;

[4] Some authors restrict negation to atoms, which yields formulas in negation normal form (NNF) [26]. While semantically equivalent, the syntactically larger class is more convenient. Moreover, conversion into NNF is simple.

- the sequence $[or(\Pi(0)), or(\Pi(1)), or(\Pi(2)), \ldots]$, otherwise. □

Multi-paths are multi-sequences which model nested paths in M.

Definition 5.3 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. A multi-path Π is *fair*, if Π is finite or if Π is infinite and $\mu(\Pi)$ is a fair path and every $\Pi(i)$ $i \geq 0$, is a fair multi-path.

The main sequence of a multi-path Π is called the *main path* of Π. □

Here, a single state is considered as a fair multi-path, which turns out to be technically convenient later.

Note that multi-paths generalize paths. Indeed, a path can be seen as an infinite multi-path Π such that each element $\Pi(i)$ is a state. Fairness of paths is generalized accordingly.

An infinite multi-path Π represents intuitively an evolving computing tree, whose branches are the main path $\mu(\Pi)$ and all paths of form $\pi_0\pi_1$ where $\pi_0 = \mu(\Pi)(0), \ldots, \mu(\Pi)(i-1)$ is a finite prefix of $\mu(\Pi)$ and π_1 is a branch of the multi-path $\Pi(i)$, where $\Pi(i)$ must be infinite.

Example 5.1 Assuming proper M, the multi-sequence $\Pi = [[s_0, s_1, s_1, \ldots], s_2, s_2, \ldots]$ is a multi-path, which represents two paths $\pi_1 = [s_0, s_1, s_1, \ldots]$ and $\pi_2 = [s_0, s_2, s_2, \ldots]$ starting at s_0 (Figure 4). π_2 is the main path $\mu(\Pi)$ of Π. The multi-path $\Pi = [[s_0, s_1, s_1, \ldots], s_2, [s_0, s_1, s_1, \ldots], s_2, [s_0, s_1, s_1, \ldots], \ldots]$ has main path $\mu(\Pi) = [s_0, s_2, s_0, s_2, \ldots]$ and represents the computation tree in which from $\mu(\Pi)$ at every even state number a path $[s_0, s_1, s_1, \ldots]$ branches off; hence, Π contains besides $\mu(\Pi)$ all paths of form $[(s_0, s_2)^i, s_0, s_1, s_1, \ldots]$. □

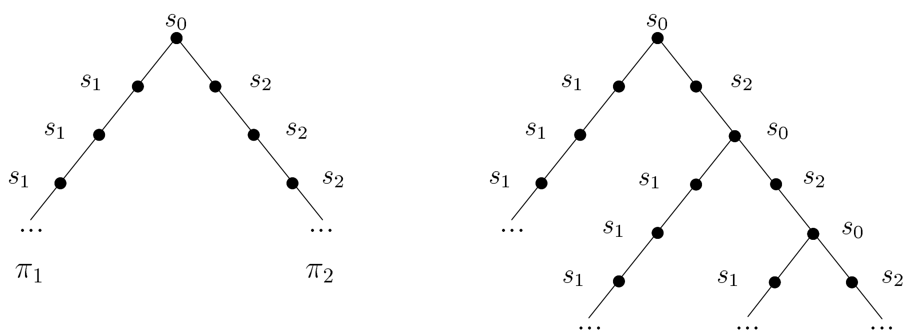

Figure 4: Branching paths

An important note is that in general, a multi-path Π may not directly reflect in its structure a truly branching computation tree. In fact, the definition allows fake branching, in the sense that two nested branching paths may amount to the same path in the structure. For example, in the multi-path $\Pi = [s_0, s_1, [s_2, s_3, s_4, \ldots]s_3, s_4, \ldots]$, the branch $s_2, s_3, s_4, \ldots$ is identical to the remainder of the main path $s_2, s_3, s_4, \ldots$. This is not a shortcoming of our definition, but an important feature; it allows to express that a particular path is a subpath of another one. In an extended vocabulary for multi-paths, this could be expressed more elegantly; however, we disregard such an extension here.

We are now prepared to formalize the notion of counterexample. Intuitively, a counterexample for a formula ϕ is a multi-path Π originating at an initial state such that ϕ is not true along Π. Since counterexamples are defined inductively, we need the concept of a local counterexample, which may origin at an arbitrary state rather than only at an initial state. For the technical definition of local counterexamples, we use an operation for merging two multi-paths into a single one.

Definition 5.4 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 by $\Pi_1 * \Pi_2$, is the multi-path recursively defined as follows:

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

Intuitively, the trees represented by Π_1 and Π_2 are merged at their common root.

Example 5.2 Merging $\Pi = [[s_0, s_{1_1}, s_{1_2}, \ldots], s_{2_1}, s_{2_3}, \ldots]$ and $\Pi' = [s_0, s_{3_1}, s_{3_2}, \ldots]$ yields

$$\begin{aligned} \Pi * \Pi' &= [\Pi, s_{3_1}, s_{3_2}, \ldots] = [[[s_0, s_{1_1}, s_{1_2}, \ldots], s_{2_1}, s_{2_2}, \ldots], s_{3_1}, s_{3_2}, \ldots], \quad \text{while} \\ \Pi' * \Pi &= [\Pi' * [s_0, s_{1_1}, s_{1_2}, \ldots], s_{2_1}, s_{2_2}, \ldots] \\ &= [[\Pi', s_{1_1}, s_{1_2}, \ldots], s_{2_1}, s_{2_2}, \ldots] \\ &= [[[s_0, s_{3_1}, s_{3_2}, \ldots], s_{1_1}, s_{1_2}, \ldots], s_{2_1}, s_{2_2}, \ldots]. \end{aligned}$$

These merges essentially represent the same branching of three paths $\pi_i = [s_0, s_{i_1}, s_{i_2}, \ldots]$ for $i = 1, 2, 3$, starting from s_0. $\Box$

Note that merging Π_1 and Π_2 by adding Π_1 as first element to Π_2 does not work, since in general, this leads to a set of paths different from those in Π_1 and Π_2; the result may even not be a multi-path.

Definition 5.5 Let M be a FC-Kripke structure and ϕ be an $ACTL$ formula on $A(M)$. A multi-path Π in M is a *local (l-) counterexample for* ϕ if, depending on the structure of ϕ, the following holds:

- if ϕ is a pure state formula: $\Pi = s$ is a state and $M, s \not\models \phi$;
- otherwise, if
  1. $\phi = \mathbf{A}(\phi_1 \mathbf{U} \phi_2)$: Π is an infinite fair multi-path and either
     - 1.1 there exists $k \geq 0$ such that $\Pi(k)$ is an l-counterexample for $\phi_1 \vee \phi_2$, $\Pi(i)$ is an l-counterexample for ϕ_2, for each $0 \leq i \leq k$, and $\Pi(j)$ is a state, for $j > k$; or

1.2 $\Pi(i)$ is a l-counterexample for ϕ_2, for each $i \geq 0$;

2. $\phi = \mathbf{A}(\phi_1 \mathbf{V} \phi_2)$: 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 ϕ_1, $\Pi(k)$ is an l-counterexample for ϕ_2, and every $\Pi(\ell)$ is a state, for $\ell > k$;
3. $\phi = \mathbf{AX}\phi_1$: Π is an infinite fair multi-path, $\Pi(1)$ is an l-counterexample for ϕ_1, and $\Pi(i)$ is a state, for each $i \neq 1$;
4. $\phi = \phi_1 \vee \phi_2$: $\Pi = \Pi_1 * \Pi_2$, where Π_i, $i = 1, 2$, is an l-counterexample for ϕ_i;
5. $\phi = \phi_1 \wedge \phi_2$: Π is an l-counterexample for either ϕ_1 or ϕ_2. $\Box$

Recall that $M \not\models \phi$ if there exists an initial state s_0 at which ϕ is false. Hence, we introduce a notion of "global" counterexample.

Definition 5.6 Let M be a FC-Kripke structure and ϕ be a formula on $A(M)$. Any l-counterexample Π for ϕ in M such that $or(\Pi) \in S_0(M)$ is called a *counterexample for ϕ in M*. $\Box$

Let us consider some examples. (A more involved example appears in Section 6.4.)

Example 5.3 Let M be the FC-Kripke structure that amounts to the labeled transition graph in Figure 5, where s_0 is the unique initial state, $A = \{a_1, a_2\}$, and $F = \emptyset$, and consider the formula $\phi = \mathbf{AF}a_1$.

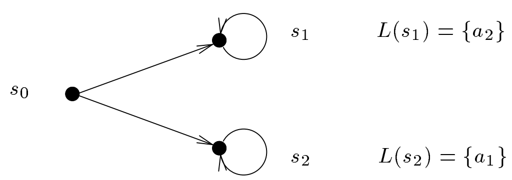

Figure 5: Labeled transition graph

It holds that $M \not\models \phi$: Along the path $\pi = [s_0, s_1, s_1, \ldots]$, the atom a_1 is false at each state, which means $M, \pi^i \models \neg a_1$, for every $i \geq 0$, i.e., $M, \pi \models \mathbf{G}\neg a_1$. Thus, π witnesses the failure of ϕ in M. As easily checked, π is a counterexample of ϕ.

Consider next the formula $\psi = \mathbf{AGAF}a_1$. Also this formula is false on M. Intuitively, this is witnessed by path π again. However, from the formal definition, π is not a counterexample of ψ, as it does not respect witness paths for the subformula $\mathbf{AF}a_1$ of ψ. The multi-path $\Pi = [[s_0, s_1, \ldots], s_1, s_1, \ldots]$ is a proper counterexample for ψ according to the definition, as well as any multi-path $[(s_1,)^i, \ldots, [s_1, s_1, \ldots], s_1, s_1, \ldots]$, where $i \geq 0$.

Finally, also the formula $\rho = \mathbf{AFAG}a_1$ is false in M, and again the path $\pi = [s_0, s_1, s_1, \ldots]$ shows this. Formally, the multi-path $[[s_0, s_1, s_1, \ldots], [s_1, s_1, \ldots], [s_1, s_1, \ldots], \ldots]$ is a counterexample for ρ; in fact, it is the unique counterexample. $\Box$

As mentioned above, in many cases a counterexample for a formula is (essentially) a single path. However, there are cases in which a true computation tree is required.

Example 5.4 Consider the FC-structure M from Example 5.3, but now the formula $\phi = \mathbf{AF}a_1 \vee \mathbf{AF}a_2$.

Clearly, $M \not\models \phi$: For every a_i, $i = 1, 2$, there is an infinite path $\pi_i = s_0, s_i, s_i, \ldots$ which never reaches a state at which a_i is true; hence, every disjunct $\mathbf{AF}a_i$ in ϕ is false. A counterexample for ϕ is the multi-path $\Pi = [[s_0, s_1, s_1, \ldots], s_2, s_2, \ldots]$, which results by merging the π_i's into $\Pi = (\pi_1 * \pi_2)$. Notice that no counterexample for ϕ exists that is an ordinary path, and that $\pi_1 * \pi_2$, $\pi_2 * \pi_1$ are the only (isomorphic) counterexamples. In this spirit, examples of concurrent programs $\mathcal{P}$ can be given which do not satisfy $\phi = \mathbf{AF}a_1 \vee \mathbf{AF}a_2$ and such that no single path is a counterexample. □

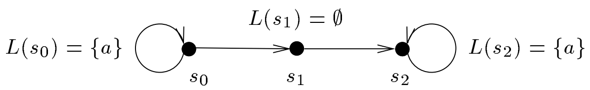

Figure 6: Another labeled transition graph

Example 5.5 Let M be the FC-Kripke structure that amounts to the labeled transition graph in Figure 6, where s_0 is the unique initial state and $F = \emptyset$. Consider the formula $\phi = \mathbf{AFAG}a$. It is easy to verify that $M \not\models \phi$. Indeed, there is a path $\pi = [s_0, s_0, \ldots]$ starting from the initial state where *always* the nested formula $\mathbf{AG}a$ does not hold, as, for each $i \geq 0$, there exists a path starting at $\pi(i)$ where *sometimes* a is not true (e.g., on the path $[s_0, s_1, s_2, s_2, \ldots]$ a is never true).

The multi-path $\Pi = [[s_0, s_1, s_2, s_2, \ldots], [s_0, s_1, s_2, s_2, \ldots], \ldots]$ is a counterexample for ϕ.

Note that no single path is a counterexample for ϕ. Indeed, for each $i \geq 0$, the suffix π^i of a counterexample path π for ϕ must be a witness of $\neg\mathbf{AG}a = \mathbf{EF}\neg a$. Thus, s_1 must occur in π^i. Now, every (single) path in M has a suffix cycle which involves either s_0 or s_2. Therefore, for each path π there exists $k \geq 0$ such that either $\pi(i) = s_0$ for each $i \geq k$, or $\pi(i) = s_2$ for each $i \geq k$. Because a is true in both s_0 and s_2, no path can witness the falsity of $\mathbf{AG}a$. □

Besides the above very simple examples, many other cases can be found in which any counterexample is a truly branching computation tree. They include formulas of form $\mathbf{AF}(\mathbf{AG}\phi \vee \mathbf{AG}\neg\phi)$, which informally state that any computation must commit at some point about a condition ϕ, and $\mathbf{AF}(\phi \vee \mathbf{AG}\psi)$, which state that at some stage ϕ is true or, regardless of how the computation proceeds, ψ holds.

Our next result states that the existence of an l-counterexample for a formula ϕ at a state s in a FC-Kripke structure M implies that ϕ is not true at s in M and vice versa. As a consequence, ϕ is not true in M if and only if there exists a counterexample for ϕ. In fact, a counterexample has as its origin an initial state of the FC-Kripke structure.

Theorem 5.1 *Let M be a FC-Kripke structure, ϕ a formula on $A(M)$, and $s \in S(M)$. Then, $M, s \not\models \phi$ if and only if there exists an l-counterexample Π for ϕ such that $or(\Pi) = s$.*

An immediate consequence of the previous proposition is the following.

Corollary 5.2 *For any FC-Kripke structure M and formula ϕ on $A(M)$, $M \not\models \phi$ if and only if there exists at least one counterexample Π for ϕ in M.*

5.2 Representation and computation of counterexamples

Counterexamples are infinite objects, and therefore it is not a priori clear that there is a finite representation for them. Fortunately, it is possible to finitely represent a relevant class of multi-path counterexamples that is sufficient for our purposes, in a similar way as a relevant class of path counterexamples.

If a path counterexample exists for an $ACTL$ formula ϕ, then a path counterexample π exists for ϕ which is finitely representable [14]. In particular, it is easy to see that if a path counterexample exists for an $ACTL$ formula ϕ, then a path counterexample π exists for ϕ which consists of a finite prefix $\pi(0), \ldots, \pi(k-1)$ and a finite cycle $\pi(k), \ldots, \pi(k+m)$, such that $\pi(\ell) = \pi(k + f_{k,m}(\ell))$ for all $\ell \geq k$, where $f_{k,m}(\ell) = (\ell - k) \bmod m$ [14]. Each fairness constraint must be satisfied at least once on this cycle. Thus, such a path counterexample can be represented as a simple expression of the form $Prefix(Cycle)^\infty$ over the alphabet of all states.

For example, the path $\pi = [s_0, s_2, s_2, s_2, \cdots]$, which loops at state s_2, can be represented by the expression $[s_0, (s_2,)^\infty]$, and the path $\sigma = [s_0, s_1, s_0, s_1, s_3, s_2, s_0, s_1, s_2, s_0, s_1, \cdots]$, which loops between s_2, s_0, and s_1, can be represented by $[s_0, s_1, s_0, s_1, s_3, (s_2, s_0, s_1,)^\infty$ (if we omit the redundant square brackets and commas, simply by $s_0 s_1 s_0 s_1 s_3 (s_2 s_0 s_1)^\infty$).

Similarly, if a multi-path counterexample exists for an $ACTL$ formula ϕ, then a multi-path counterexample π exists for ϕ which is finitely representable. To see this, first note that each counterexample is, from its definition, a multi-path whose depth is finite. Using an inductive argument, it can be seen that if a counterexample of certain depth d exists, then a counterexample Π of depth less or equal d exists which consists of a finite prefix $\Pi(0), \cdots, \Pi(k-1)$ and a (finite) cycle $\Pi(k), \ldots, \Pi(k+m) = \Pi(k)$ such that $\Pi(\ell) = \Pi(k + f_{k,m}(\ell))$ for all $\ell \geq k$, where again $f_{k,m}(\ell) = (\ell - k) \bmod m$. This counterexample can be represented as an expression of form $Prefix(Cycle)^\infty$ over the alphabet of states and counterexamples of depth smaller d; thus, if we bottom out the representation, it can be represented as a nested expression built over an alphabet of states and paths (or even states alone, if we wish).

For example, the counterexample $\Pi = [[s_0, s_1, s_1, \ldots], s_2, s_2, \ldots]$ in Example 5.4 can be represented by $[s_0(s_1)^\infty, (s_2,)^\infty]$, and the counterexample $\Pi = [[s_0, s_1, s_2, s_2, \ldots], [s_0, s_1, s_2, s_2, \ldots], \ldots]$ in Example 5.5 by $[(s_0 s_1 (s_2)^\infty,)^\infty]$ (here $s_0(s_1)^\infty$ and $s_0 s_1 (s_2)^\infty$ are the simplified path expressions $[s_0, (s_1,)^\infty]$ and $[s_0, s_1, (s_2,)^\infty]$, respectively).

We face now the problem of computing counterexamples and provide a method, based on a procedure of [14], for constructing a counterexample. Briefly, given an $ACTL$ formula

ϕ and a state s (which is usually an initial state of the FC-Kripke structure), the procedure described in [14] produces a single path, starting from s, which demonstrates or hints at the failure of ϕ. In our framework, for most $ACTL$ formulas this single path corresponds to the main path of a counterexample; the structure of a counterexample itself depends on nestings of $\mathbf{A}$ in ϕ. For instance, if ϕ is a pure state formula, then a counterexample for ϕ in M is simply an initial state; a counterexample for a formula $\mathbf{AG}\phi$ is a path; and, a counterexample for a formula $\mathbf{AG}(\phi_1 \vee \mathbf{AG}\phi_2)$ consists of two nested paths: the first starts from an initial state, and the second branches off at a certain point of the first.

Therefore, if the $ACTL$ formula ϕ has only one (universal) quantifier, then it admits path counterexamples that can be correctly (and efficiently) computed by using the procedure designed in [14].[5]

Formulas with more than one universal quantifier do not admit, in general, path counterexamples (see Examples 5.4 and 5.5) and the procedure of [14] cannot be employed to compute counterexamples of such formulas (as it computes simple paths; while branching multi-paths are required). A suitable extension of the algorithm of [14] is needed for computing multi-path counterexamples in the general case.

Examples 5.4 and 5.5 show the two basic sources that imply the need of multi-paths in counterexamples: *disjunction* and *nested quantifiers*. Multi-path counterexamples of a formula consisting of the disjunction of atomic (i.e., with one quantifier) $ACTL$ formulas, say $\phi = \mathbf{AF}a_1 \vee \mathbf{AF}a_2$, can be computed very easily: (i) compute an initial state, say s_0, on which $\phi = \mathbf{AF}a_1 \vee \mathbf{AF}a_2$ is false (e.g., by computing the suitable fixpoint for the negation of ϕ, i.e., $\mathbf{EG}\neg a_1 \wedge \mathbf{EG}\neg a_2$); (ii) call the procedure of [14] on $\mathbf{AF}a_1$ and on $\mathbf{AF}a_2$ with s_0 as the initial state; (iii) create the multi-path counterexample for ϕ by merging the two path counterexamples for $\mathbf{AF}a_1$ and $\mathbf{AF}a_2$, respectively, returned by the procedure. The computation of multi-path counterexamples of $ACTL$ formulae with nested quantifiers is harder and requires a recursive extension of the procedure of [14]. A naive way to make this extension is to recursively call (top-down on the structure of the formula) the procedure of [14] to build the counterexamples of the nested $ACTL$ subformulas. For instance, a (multi-path) counterexample of the formula $\phi = \mathbf{AFAG}a$ of Example 5.5 can be computed as follows. Call first the procedure of [14] on ϕ; this procedure returns a fair path π such that, for each integer i, $\pi(i)$ is a state where $\mathbf{EF}\neg a$ is true (i.e., $\mathbf{AG}a$ is false). Then, for each integer i, call the procedure again to compute a path counterexample, for $\mathbf{AG}a$ starting from $\pi(i)$ (i.e., with $\pi(i)$ as the initial state), say π_i. The multi-path Π such that $\Pi(i) = \pi_i$ is a multi-path counterexample for $\phi = \mathbf{AFAG}a$.[6] Other nested $ACTL$ formulae can be treated in a similar way.

It is worth noting that several optimizations are possible in the computation of counterexamples. Relevant optimizations can be designed by singling out cases of nested $ACTL$ formulas where paths are sufficient to witness the falsity. For instance, $\mathbf{AFAF}\psi$, $\mathbf{AGAF}\psi$,

[5]Note that this procedure is applicable here, as the Kripke structure of a program has always a total transition relation by construction.

[6]Note that the path π returned by [14] is finitely represented as a prefix and cycle (see above). Consequently, a finite number of calls to the procedure is sufficient, and the multi-path Π is finitely represented.

AXAGψ, **AGAG**ψ (which is equivalent to **AG**ψ), where ψ is a propositional formula, admit path counterexamples. A detailed analysis of the (nested) formulas that admit path counterexamples leads beyond this paper, and is carried out elsewhere [7].

6 Optimization Techniques for Repair

In the previous sections, we have introduced the problem of program repair and the concept of counterexample. In this section, we present some optimization techniques which allow to cut the search space for a repair. The techniques utilize counterexamples from above, and are most effective in the case of single correction repairs; an important aspect is that they are efficiently (linear time) applicable.

The basic observation underlying our techniques is that a counterexample must contain certain transitions which prove the failure of a formula ϕ. Any repair must remove these transitions, i.e., avoid that such transitions take place. Thus, if a correction leaves these transitions unchanged, then it can not amount to a repair. By determining such corrections from a counterexample at hand, a number of useless corrections might be excluded.

Before we present the particular techniques that we have developed in this spirit, we have to introduce a suitable notion of equivalence between multi-paths. In the rest of this section, $\mathcal{P} = P_1, \ldots, P_n$ is a concurrent program on variables $\mathbf{x}$ having a set F of pure state formulas as fairness constraints, and ϕ is an $ACTL$ formula.

6.1 Equivalent multi-paths and weak corrections

Definition 6.1 For any formula ϕ, we denote by $ap(\phi)$ the set of atomic propositions in ϕ. For any set F of formulas, we denote $ap(F) = \bigcup_{\phi \in F} ap(\phi)$. □

Definition 6.2 Let M and M' be two FC-Kripke structures such that $F(M) = F(M')$, and let $A' \subseteq A(M) \cap A(M')$. Then,

(i) states $s \in S(M)$ and $s' \in s(M')$ are *equivalent on* A', denoted $s \approx_{A'} s'$, if $L(M)(s) \cap A' = L(M')(s') \cap A'$;

(ii) multi-paths Π in M and Π' in M', are *equivalent on* A', denoted $\Pi \approx_{A'} \Pi'$, if either

- (ii.1) both $\Pi = s$ and $\Pi' = s'$ are states and $s \approx_{A'} s'$, or
- (ii.2) both Π and Π' are infinite and $\Pi(i) \approx_{A'} \Pi'(i)$, for every $i \geq 0$. □

Intuitively, if Π and Π' are equivalent on A' then, for every formula ϕ with $ap(\phi) \subseteq A'$, we have that ϕ is true on Π if and only if ϕ is true on Π'.

The next proposition is the basis of later results. It states a transfer results for counterexamples between structures: Given a counterexample Π for a formula ϕ in M, any multi-path in a structure M' that is equivalent to Π on the variables of ϕ is a counterexample for ϕ in M'. As a consequence, if M' is the result of a repair to the FC-Kripke structure of a program M w.r.t. ϕ, it is impossible to find in M' a "pattern of behavior" equivalent on $ap(\phi)$ to any counterexample in M. This is captured formally by the next propositions.

Proposition 6.1 *Let M and M' be two FC-Kripke structures which coincide on S, S_0, L, and F. Let Π be an l-counterexample in M for a formula ϕ on $ap(\phi) \subseteq A(M)$. If Π' is a fair multi-path in M' such that $\Pi \approx_{ap(\phi)} \Pi'$, then Π' is an l-counterexample for ϕ in M'.*

We next introduce the concept of a weak correction for a counterexample, that singles out corrections that are certainly useless to fix the error (i.e., modifying the concurrent program by any set of weak corrections would not allow to entail the *CTL* formula at hand). Prior to this, we fix a notation for the variables which are touched by a single correction.

Definition 6.3 For any single correction $\alpha = \langle k, b, \gamma\rangle$ for $\mathcal{P}$, $V(\alpha)$ is the following set of variables:

$$V(\alpha) = \begin{cases} \{x_j, var(stmnt(k,b))\}, & \text{if } \gamma = l\text{-}change(x_j); \\ \{var(stmnt(k,b))\}, & \text{if } \gamma = r\text{-}change; \\ \{var(stmnt(k,b)), var(stmnt(k,b+1))\}, & \text{if } \gamma = swap. \end{cases}$$

For any complex correction $\overline{\alpha} = \alpha_1 \cdots \alpha_q$, let $V(\overline{\alpha}) = \bigcup_{i=1}^{q} V(\alpha_i)$. □

Definition 6.4 Let Π be a fair multi-path in $M_F(\mathcal{P})$ such that $or(\Pi) \in S_0$. A correction $\overline{\alpha}$ for $\mathcal{P}$ is called *weak w.r.t.* Π, if there exists a fair multi-path Π' in $M' = M_F(\mathcal{P}^{\overline{\alpha}})$ such that $or(\Pi') \in S_0$ and $\Pi \approx_{A'} \Pi'$ for $A' = A(M) - V(\overline{\alpha})$. □

Intuitively, a weak correction for Π does not modify the set of formulas holding on Π (if we see Π' as the "image" of Π under the correction $\overline{\alpha}$), apart from formulas involving propositions in $V(\overline{\alpha})$ (i.e., that are explicitly modified by $\overline{\alpha}$).

The next proposition states that a program repair not involving variables from ϕ cannot be a weak correction w.r.t any counterexample of ϕ. In fact, if a repair is a weak correction for some counterexample, the modified program will produce the same counterexample for the formula in the modified system. Roughly speaking, counterexamples have to be modified by a correction if we hope to repair the system through it. This result will be utilized later.

Proposition 6.2 *Let $\overline{\alpha}$ be a repair for the PRP $\mathcal{R} = \langle \mathcal{P}, F, \phi\rangle$, such that $V(\overline{\alpha}) \cap ap(\phi) = \emptyset$. Then, for every counterexample Π for ϕ in $M_F(\mathcal{P})$, $\overline{\alpha}$ is not weak w.r.t. Π.*

Proof. Towards a contradiction, suppose $\overline{\alpha}$ is a weak correction w.r.t. Π. Hence, there exists a fair multi-path Π' in $M' = M_F(\mathcal{P}^{\overline{\alpha}})$ such that $\Pi \approx_{A'} \Pi'$ where $A' = A \setminus V(\overline{\alpha})$. By hypothesis on ϕ, $ap(\phi) \subseteq A'$, and thus Proposition 6.1 implies that Π' is a counterexample for ϕ in M'. Hence, by Corollary 5.2, $\overline{\alpha}$ is not a repair for $\mathcal{R}$, which is a contradiction. □

The consequences of this proposition will be exploited in Section 6.3 for optimization issues (where the result is used to prove the main theorem). Intuitively, if we know that a correction is weak w.r.t. a counterexample, then we can discard it from the "candidates" repair, unless it touches explicitly the atomic propositions in $ap(\phi)$.

6.2 Correction execution

The first optimization method respects whether the counterexample passes through a statement that has been modified; if not, then it is still valid, and the respective correction is ruled out as a repair.

Informally, given a counterexample Π and a single correction $\alpha = \langle k, m, \gamma \rangle$, we say that Π *executes* α if there exists some path in Π along which process P_k passes through the break points affected by α. Here, we must take into account that a computation path may pass only through part of them, due to jumps; it could pass break point m but not $m+1$.

We next prove formally that if a correction is a repair, then it must be executed by any counterexample. For that, we must formalize correction execution.

Definition 6.5 Let $\alpha = \langle k, m, \gamma \rangle$ be a correction. A path π *executes* α, if there exists $i \geq 0$ such that $\mathbf{ex}(\pi(i+1)) = k$ (i.e., the transition from i to $i+1$ executes P_k) and, further

- $\gamma \in \{r\text{-}change, l\text{-}change(x_j)\}$ and $\mathbf{b}(\pi(i))_k = m$, or
- $\gamma = swap$ and $\mathbf{b}(\pi(i))_k \in \{m, m+1\}$.

A multi-path Π *executes* α, if Π is infinite and, further, either the main path $\mu(\Pi)$ of Π executes α, or there exists an integer $i \geq 0$ such that $\Pi(i)$ executes α.

A path π (resp. multi-path Π) *executes* a correction $\overline{\alpha} = \alpha_1 \cdots \alpha_q$, if it executes α_i, for some $i = 1, \ldots, q$. □

We note a couple of simple lemmas, which are useful in the proof of the next result.

Lemma 6.3 *Let $\alpha = \langle k, m, \gamma \rangle$ be a single correction, and let $(s_1, s_2) \in R$. If (i) $\gamma \in \{r\text{-}change, l\text{-}change(x_j)\}$ and $\mathbf{b}(s_1)_k \neq m$, or (ii) $\gamma = swap$ and $\mathbf{b}(s_1)_k \notin \{m, m+1\}$, then $(s_1, s_2) \in R^\alpha$.*

Proof. Observe that the effect of changes to $G(P_k)$ by α is restricted to labels of arcs which are leaving m in case (i) and leaving m or $m+1$ in case (ii). □

Lemma 6.4 *Let $\overline{\alpha}$ be a correction for $\mathcal{P}$, and let π be a path in $M_F(\mathcal{P})$ which does not execute $\overline{\alpha}$. Then π is also a path in $M_F(\mathcal{P}^{\overline{\alpha}})$.*

Proof. If π does not execute $\overline{\alpha} = \alpha_1 \cdots \alpha_q$, then π does not execute any of $\alpha_1, \ldots, \alpha_q$. Thus, by Lemma 6.3, we obtain that π is a path in $\mathcal{P}^{\alpha_1}$. By a repeated argument, we obtain that π is a path in $\mathcal{P}^{\alpha_1 \cdots \alpha_i}$, for every $i = 1, \ldots, q$. □

Lemma 6.5 *Let $\mathcal{R} = \langle \mathcal{P}, F, \phi \rangle$ be a PRP for which some repair exists. Then, every counterexample Π for ϕ in $M_F(\mathcal{P})$ is infinite.*

The next theorem states that, given a PRP $\mathcal{R}$ and a repair $\overline{\alpha}$ for it, every counterexample executes $\overline{\alpha}$.

Theorem 6.6 *Let $\overline{\alpha}$ be a repair for the PRP $\mathcal{R} = \langle \mathcal{P}, F, \phi \rangle$. Then, every counterexample Π for ϕ in $M_F(\mathcal{P})$ is infinite and executes $\overline{\alpha}$.*

This result is intuitive; if a counterexample does not touch the statements which have been corrected, the same multi-path will be present in the Kripke structure of the corrected program. As a consequence of Theorem 6.6, while looking for possible repairs, we can rule out a priori any correction that is not executed on a counterexample.

An important observation is that correction execution can be tested efficiently. In fact, this is possible in linear time if counterexamples are properly represented, e.g. in the scheme of Section 5.2.

Proposition 6.7 *Given a concurrent program $\mathcal{P}$, a collection $\mathcal{C}$ of complex corrections $\overline{\alpha} = \alpha_1 \cdots \alpha_q$, and a counterexample Π, the corrections in $\mathcal{C}$ which are not executed by Π can be discarded in linear time.*

Thus, checking for correction execution is a low cost pruning principle which can benefit in reduction of the search space.

6.3 Correction exploitation

We can identify another important property that repairs must verify, and exploit it for further optimization.

Recall that a correction $\overline{\alpha}$ involves a set of variables, denoted by $V(\overline{\alpha})$. Then, only computations that evaluate some variable in $V(\overline{\alpha})$ can be influenced by the correction $\overline{\alpha}$. Indeed, the values of variables not in $V(\overline{\alpha})$ and guards of labels in the process graphs can change only if variables whose values are affected by the correction $\overline{\alpha}$ are referenced. A path that does not evaluate variables of $V(\overline{\alpha})$ will be transformed by the correction into an equivalent path on $A \setminus V(\overline{\alpha})$; the same happens to a multi-path. Hence, a further property that a repair $\overline{\alpha}$ must satisfy is that all counterexamples have to evaluate some variable in $V(\overline{\alpha})$. We formalize this intuition next.

Definition 6.6 A path π *exploits* a correction $\alpha = \langle k, m, \gamma \rangle$, if there exists an integer $i \geq 0$ such that the command executed in $(\pi(i), \pi(i+1))$ evaluates [7] some variable in $V(\alpha)$.
A multi-path Π *exploits* a correction $\alpha = \langle k, m, \gamma \rangle$, if Π is infinite and, furthermore, either $\mu(\Pi)$ exploits α or there exists an integer $i \geq 0$ such that $\Pi(i)$ exploits α.
A path π (resp. multi-path Π) *exploits* a correction $\overline{\alpha} = \alpha_1 \cdots \alpha_q$, if it exploits α_i for some $i = 1, \ldots, q$. $\Box$

The next result states that under certain conditions, any counterexample must exploit a given repair. It is based on the fact that if a path π does not exploit a single correction α, then we can find a path π' in the modified structure which is modulo $V(\alpha)$ equivalent to π and starts from any given state equivalent to the initial state of π (see Lemma 7.1 in the appendix).

[7] Note that an assignment statement *evaluates* the variables appearing in its right side; a conditional statement *evaluates* the variables appearing in its **if** condition.

Theorem 6.8 *Let $\overline{\alpha}$ be a repair for the PRP $\mathcal{R} = \langle \mathcal{P}, F, \phi \rangle$ such that $V(\overline{\alpha}) \cap (ap(\phi) \cup ap(F)) = \emptyset$. Then, every counterexample Π for ϕ in $M_F(\mathcal{P})$ exploits $\overline{\alpha}$.*

Notice that the condition $V(\overline{\alpha}) \cap ap(F) = \emptyset$ is not much restrictive, since F should be concerned about fair executions (and speak about processes and break points) rather than the value of program variables, which should be done in the specification ϕ. On the other hand, the condition $V(\overline{\alpha}) \cap ap(\phi) = \emptyset$ is more restrictive, but cannot be removed in general. For example, if ϕ says that some variable *ok* must always be committed to true in a computation ($\mathbf{AFAG}ok$), a correction of $ok := false$ to $ok := true$ might eliminate all counterexamples, even if no computation references the value of *ok*.

An important point is that like correction execution, correction exploitation is a pruning principle which can be applied efficiently. In fact, in the setting of Section 6.2, it is possible to implement correction exploitation in linear time.

Proposition 6.9 *Given a concurrent program $\mathcal{P}$, a collection $\mathcal{C}$ of complex corrections $\overline{\alpha} = \alpha_1 \cdots \alpha_q$, and a counterexample Π, the corrections in $\mathcal{C}$ which are not exploited by Π can be discarded in linear time.*

6.4 Example for optimization

Let us consider an example which demonstrates that by the optimization techniques in Sections 6.2 and 6.3, quite some savings can be gained in finding a repair.

Example 6.1 Reconsider the concurrent program $\mathcal{P}$ in Figure 1, and let ϕ and F be as in Example 4.2, viz. $\phi = \phi_{flags} \wedge \phi_{crit}$, where $\phi_{flags} = \bigwedge_{i=1,2,\, V=A,B} \mathbf{AG}(flagiV \rightarrow \mathbf{AF}\neg flagiV)$ and $\phi_{crit} = \mathbf{AG}(\neg(b_1^{12} \wedge b_2^{10})) \wedge \mathbf{AG}(\neg(b_1^5 \wedge (b_2^5 \vee b_2^{11})))$, and $F = \{e_A, e_B\}$. As already mentioned, the program is not correct. Therefore, $M_F(\mathcal{P}) \not\models \phi$ must hold and a counterexample for ϕ must exist.

It can be verified that indeed $M_F(\mathcal{P}) \not\models \phi$: there is a path π from an initial state, leading to a state at which $flag1B = true$, and where another path π' starts along which $flag1B$ is always true. Thus, the formula $\mathbf{EF}(flag1B \wedge \mathbf{EG}flag1B)$ is true, which means that the formula $\mathbf{AG}(flag1B \rightarrow \mathbf{AF}\neg flag1B)$ is false; since the latter is a conjunct of ϕ, also ϕ is false. This should give rise to a counterexample.

Indeed, consider the following path π (we show of each state s_i, left to right, the break points for P_A and P_B, the program variables that are true, and the process lastly executed; in the initial state s_0, all variables are false):

$$
\begin{array}{llllll}
\pi(0) = s_0 = & 1 & 1 & & P_A \\
\pi(1) = s_1 = & 2 & 1 & flag1A & P_A \\
\pi(2) = s_2 = & 2 & 2 & flag1A, flag1B & P_B \\
\pi(3) = s_3 = & 3 & 2 & flag1A, flag1B & P_A \\
\pi(4) = s_4 = & 3 & 3 & flag1A, flag1B & P_B \\
\cdots
\end{array}
$$

At s_4, where $flag1B$ is true, the path π' starts:

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

where $flag1B$ is always true. The computation goes on such that P_B loops between break points 3 and 4, and P_A loops between break points 1–3, 5–7, and 14.

Notice that starting from s_1, not all computation paths are wrong. Indeed, for each state in which $flag1A$ is false, P_B could go beyond break point 3 into the critical section.

Suppose then an oracle for a counterexample (e.g., a call to a procedure) returns the following multi-path Π, which formally represents the intuitive computation from above:

$$[s_0, s_1, s_2, s_3, [s_4, s_5 \ldots, s_{14}, s_{15}, s_5, \ldots, s_{14}, \ldots], s_5, \ldots, s_{14}, s_{15}, s_5, \ldots, s_{14}, \ldots]$$

here, the branching path and the remaining main path are identical; i.e., the branching path is a subpath of the main path, and thus Π intuitively amounts to a single path. Note that Π can be represented, using the scheme in Section 5.2, by the expression $[s_0, s_1, s_2, s_3, s_4(s_5 \cdots s_{15})^\infty, (s_5, \ldots, s_{15},)^\infty]$.

Now let us look for a repair for this program, where we assume that a single error is present and thus focus on single correction repairs, with possible further restriction to particular correction types.

Then, the naive repair approach 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 first consider single statement repairs which change right hand sides of assignments. 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 $flag1A$,

$flag1B$, $turn1B$. Hence, even if the assignment $turn2B := true$ after break point 9 in P_A is executed by Π, no correction of it has to be considered, since $turn2B$ is not evaluated in Π. Thus, only 5 out of 12 candidate repairs remain. In case of arbitrary single assignment repairs, 15 out of 72 candidate repairs remain. Finally, if also assignment interchange is considered, then 17 out of 77 candidate repairs remain.

Apparently, a single correction repair for $\mathcal{P}$ is the correction $\alpha = \langle A, 2, \gamma \rangle$ where $\gamma =$ *r-change*, i.e., statement 2 in P_A is changed to $turn1B := true$. Indeed, the modified program $\mathcal{P}^\alpha$ does not enable P_B to loop forever between 3 and 4. $\square$

6.5 Further optimization

Of course, further and stronger optimization techniques are imaginable, but they may come at a higher computational price which must be paid for analyzing the structure of $\mathcal{P}$, ϕ, and the counterexample Π.

For example, the exploitation technique from Section 6.3 can be sharpened by incorporating that the counterexample must actually "see" the effect of a single correction α, rather than only referencing a variable x_i in $V(\alpha)$; it might well happen that the value of x_i is overwritten before it is referenced, and thus the correction is not useful, provided that it does not directly affect the program specification or fairness constraints.

For an illustrating, simple example, consider the following (part of a) process:

$$
\begin{array}{ll}
1: & x := true; \\
2: & y := false; \\
3: & x := y; \\
4: & z := \mathit{not}\ x; \\
\vdots &
\end{array}
$$

Here, a correction of *stmnt*(1) to *x := false* is not "seen" by the program; likewise, a correction of *stmnt*(1) to *y := true* is not seen by the program. Thus, these corrections of the program are useless and can be discarded (provided that they do not directly affect the specification or fairness constraints).

For a single right side modification, we can say that a computation path *sees* this correction, if

1. it evaluates at some point k the variable, say x, of the left hand side of the corrected statement,
2. at some point $j \leq k$ the corrected statement was executed, and,
3. between j and k no assignment to x was made.

Thus, it is guaranteed that at point k after the execution of the corrected statement, the computation path experiences the effect of the correction (either by evaluating x in an **if** condition or by using it in the right side of an assignment). If no path of a counterexample sees the correction, than it is useless (if it does not interfere with the specification or the fairness constraints).

Importantly, the test whether a counterexample sees a correction is efficiently possible, given the counterexample representation that we have outlined in Section 5.2. For assignment swaps and complex corrections, the formalization of strong exploitation is similar, but it is more involved and also may have higher evaluation cost.

Another important optimization issue is the investigation of more sophisticated techniques for pruning the search space by using counterexamples. Connected with this is an optimized procedure for counterexample construction. So far, we were satisfied by having any counterexample which proves that a formula fails on a structure. However, in order to locate an error and to find a suitable repair, some counterexamples are obviously more useful than others. For example, if a counterexample Π references a single variable during its execution and another counterexample Π' references all variables, then Π might be preferable since it allows to attribute the error to fewer statements than Π'. Thus, a formal, comparative notion of quality of counterexamples has to be developed, and an optimized counterexample procedure which finds as good counterexamples as possible. This may turn out to be difficult and computationally complex, which calls for appropriate heuristics.

7 Conclusion

Model checking, which is successfully used in verifying concurrent systems, appears to be an area which has high potential for applying AI principles. In this direction, we have presented an approach to combine model checking with repair, which has been considered in the area of automated diagnosis. In the course of the formalization of this approach, theory revision and abductive reasoning play an important role.

To our knowledge, our approach to integrate repair into model checking is novel. Notice that automated diagnosis and repair of programs was investigated e.g. in [15, 44, 45]. The framework and setting in [15] (logic programs) is quite different; the approach of [44, 45] is developed for functional programming languages and generates out of a (faulty) program an instance of a model-based diagnosis problem. This is quite different from our approach, which aims at using techniques and concepts from model checking and combining them with AI principles.

For a successful integration of repair into the model checking paradigm, we had to extend the notion of a counterexample as described in [11] formally such that it is technically available on the whole language of $ACTL$ formulas. For this purpose, we have introduced the concept of multi-paths as a suitable formal of counterexample trees.

We then have presented optimization techniques which, as demonstrated on an example, may allow for a considerable reduction of the search space for a program repair. These optimization techniques, correction execution and correction exploitation, can be implemented to run efficiently.

Naturally, not all interesting and relevant issues can be addressed in this paper which introduces our approach, and a number of them must be left for further work.

We have already discussed some interesting optimization issues in Section 6.5. Another important direction of research concerns the extension of the framework by further types of corrections. The current framework allows for right modifications and left modifications of assignments, as well as the interchange of assignments. Further corrections, e.g. more complex right hand side modifications, could be desirable. Connected with this, an analysis of common errors in concurrent programming or protocol specification would be acknowledged in order to identify relevant errors which our framework should be able to handle.

Furthermore, it remains to be analyzed how abstract principles for abductive reasoning [34] can be exploited in this specific application domain, as well as whether abductive algorithms and computational devices developed in AI (e.g., truth maintenance systems) are fruitfully applicable.

Acknowledgements. The authors would like to thank O. Grumberg, K. McMillan, and S. Campos for their clarifying comments on counterexamples, in particular on how counterexample paths are computed and used in the SMV procedure. Moreover, they are grateful to M. Stumptner and F. Wotawa for bibliographic hints on diagnosis of programs.

Appendix

Theorem 5.1 *Let M be a FC-Kripke structure, ϕ a formula on $A(M)$, and $s \in S(M)$. Then, $M, s \not\models \phi$ if and only if there exists an l-counterexample Π for ϕ such that $or(\Pi) = s$.*

Proof. The statement clearly holds for pure state formulas, as $\Pi = s$ is the desired counterexample. We prove that it holds also for non-pure state formulas ϕ by structural induction, starting from a pure state formula.
($\Longleftarrow$) Suppose then ϕ is a non-pure state formula, and that Π is an l-counterexample for ϕ such that $or(\Pi) = s$. We have to show that $M, s \not\models \phi$. Consider the following possible cases:

1. $\phi = \mathbf{A}(\phi_1 \mathbf{U} \phi_2)$. It holds that $M, s \not\models \mathbf{A}(\phi_1 \mathbf{U} \phi_2)$ if and only if either $M, s \models \mathbf{E}(\neg\phi_2 \mathbf{U}(\neg\phi_1 \wedge \neg\phi_2))$ or $M, s \models \mathbf{EG}(\neg\phi_2)$ (cf. Proposition 2.1). Π is an infinite multi-path and either

   1.1 there exists an integer $k \geq 0$ such that $\Pi(k)$ is an l-counterexample for $\phi_1 \vee \phi_2$, $\Pi(i)$ is an l-counterexample for ϕ_2, for each $0 \leq i \leq k$, and $\Pi(j)$ is a state, for $j > k$, or

   1.2 $\Pi(i)$ is an l-counterexample for ϕ_2, for each $i \geq 0$;

   Let $\pi = \mu(\Pi)$ be the main path of Π; π is a fair path. In the case 1.1, $M, \pi \models \neg\phi_2 \mathbf{U}(\neg\phi_1 \wedge \neg\phi_2)$. Indeed, for the state $\pi(k) = or(\Pi(k))$ in π, we have $M, \pi(k) \not\models \phi_1 \vee \phi_2$: either $\phi_1 \vee \phi_2$ is a pure state formula, which means $\Pi(k) = \pi(k)$, or $\phi_1 \vee \phi_2$ is not a pure state formula and $\Pi(k) = \Pi' * \Pi''$ is the merge of two counterexamples for ϕ_1 and ϕ_2, respectively. By the induction hypothesis, $M, \pi(k) \not\models \phi_i$, for $i = 1, 2$; it follows $M, \pi(k) \models \neg\phi_1 \wedge \neg\phi_2$. Moreover, for every i such that $0 \leq i \leq k$, the induction hypothesis implies $M, \pi(i) \models \neg\phi_2$. This implies the claim $M, \pi \models \neg\phi_2 \mathbf{U}(\neg\phi_1 \wedge \neg\phi_2)$. As a consequence, $M \models \mathbf{E}(\neg\phi_2 \mathbf{U}(\neg\phi_1 \wedge \neg\phi_2)$, which means $M, s \not\models \mathbf{A}(\phi_1 \mathbf{U} \phi_2)$.

In the case 1.2, we have $M, \pi \models \mathbf{G}\neg\phi_2$. Indeed, the induction hypothesis implies $M, \pi(i) \models \neg\phi_2$, for every $i \geq 0$, since $\pi(i)$ is the origin of an l-counterexample for ϕ_2.

2. $\phi = \mathbf{A}(\phi_1 \mathbf{V} \phi_2)$. Then, Π is infinite and there exists a $k \geq 0$ such that $\Pi(j)$ is a counterexample for ϕ_1, for every $0 \leq j < k$, and $\Pi(k)$ is a counterexample for ϕ_2. Hence, by the induction hypothesis, $M, or(\Pi(j)) \not\models \phi_1$, for $0 \leq j < k$ and $M, or(\Pi(k)) \not\models \phi_2$. Let $\pi = \mu(\Pi)$ be the main path of Π. Since $or(\Pi(i)) = \pi(i)$, for every $0 \leq i \leq k$, and $\neg\phi_1$, $\neg\phi_2$ are state formulas, it follows that $M, \mu^j \models \neg\phi_1$, for $0 \leq j < k$ and $M, \mu^k \models \neg\phi_2$. Consequently, $M, \mu \models \neg\phi_1 \mathbf{U} \neg\phi_2$, which implies $M, \pi(0) \models \mathbf{E}(\neg\phi_1 \mathbf{U} \neg\phi_2)$. Since $\pi(0) = s$ and $\mathbf{E}(\neg\phi_1 \mathbf{U} \neg\phi_2)$ is equivalent to $\neg\phi$, it follows $M, s \not\models \phi$.

3. $\phi = \mathbf{AX}\phi_1$. Π is an infinite multi-path, such that $\Pi(1)$ is an l-counterexample for ϕ_1. Let $\pi = \mu(\Pi)$ be the main path of Π. π is a fair path. By the induction hypothesis, $M, \pi(1) \not\models \phi_1$. Hence, $M, s \not\models \phi$.

4. $\phi = \phi_1 \vee \phi_2$. There exist two multi-paths Π_1 and Π_2 such that $\Pi = \Pi_1 * \Pi_2$, Π_1 is an l-counterexample for ϕ_1, Π_2 is an l-counterexample for ϕ_2, and $or(\Pi_1) = or(\Pi_2) = s$. Hence, by the induction hypothesis, $M, s \not\models \phi_1$ and $M, s \not\models \phi_2$. Therefore, $M, s \not\models \phi$.

5. $\phi = \phi_1 \wedge \phi_2$. W.l.o.g, Π is a counterexample for ϕ_1 with $or(\Pi) = s$. By the induction hypothesis, $M, s \not\models \phi_1$, hence $M, s \not\models \phi$.

($\Longrightarrow$) Consider a non-pure state formula ϕ, and suppose that $M, s \not\models \phi$. We have to show that an l-counterexample Π for ϕ exists such that $or(\Pi) = s$.

1. $\phi = \mathbf{A}(\phi_1 \mathbf{U} \phi_2)$. It holds that $M, s \not\models \phi$ if and only if either (i) $M, s \models \mathbf{E}(\neg\phi_2 \mathbf{U}(\neg\phi_1 \wedge \neg\phi_2))$ or (ii) $M, s \models \mathbf{EG}(\neg\phi_2)$ (cf. Proposition 2.1).

   In the case (i), there exist a fair path π and an integer $k \geq 0$, such that $\pi(0) = s$, $M, \pi(i) \not\models \phi_2$ (i.e., $M, \pi(i) \models \neg\phi_2$), for each $0 \leq i < k$, and $M, \pi(k) \not\models \phi_1 \vee \phi_2$ (i.e., $M, \pi(k) \models \neg\phi_1 \wedge \neg\phi_2$). Thus, by the induction hypothesis, for each $0 \leq i \leq k$, there exists an l-counterexample Π_i for ϕ_2 such that $or(\Pi_i) = \pi(i)$, and by the induction hypothesis and an argument as in case 4., there exists an l-counterexample Π_k for $\phi_1 \vee \phi_2$ such that $or(\Pi_k) = \pi(k)$. Consider now the multi-sequence $\Pi = [\Pi_0, \Pi_1, \cdots, \Pi_k, \pi(k+1), \pi(k+2), \cdots]$. Clearly, Π_1 is a fair multi-path and $or(\Pi_1) = s$. Indeed, $\mu(\Pi) = \pi$, and each element $\Pi(i)$ is a multi-path. It is easily seen that Π is a l-counterexample for ϕ such that $or(\Pi) = s$.

   In the case (ii), there exists a path π such that $\pi(0) = s$ and $M, \pi \models \mathbf{G}(\neg\phi_2)$; since ϕ_2 is a state formula, this means $M, \pi(i) \not\models \phi_2$, for every $i \geq 0$. Hence, by the induction hypothesis, there exists an l-counterexample Π_i for ϕ_2 such that $or(\Pi_i) = \pi(i)$, for every $i \geq 0$. Consider the multi-sequence $\Pi = [\Pi_0, \Pi_1, \Pi_2, \cdots]$. Clearly, Π is a fair multi-path with $or(\Pi) = s$, and is a l-counterexample for ϕ.

2. $\phi = \mathbf{A}(\phi_1 \mathbf{V} \phi_2)$. We have $M, s \models \mathbf{E}(\neg\phi_1 \mathbf{U} \neg\phi_2)$ by duality of $\mathbf{V}$ and $\mathbf{U}$. Hence, there exists a fair infinite path π such that $\pi(0) = s$ and $M, \pi \models \neg\phi_1 \mathbf{U} \neg\phi_2$. The latter means that there exists a k such that $M, \pi^j \models \neg\phi_1$, for every $0 \leq j < k$ and $M, \pi^k \models \neg\phi_2$. Since ϕ_1 and ϕ_2 are state formulas, it follows that $M, \pi(j) \not\models \phi_1$ for every $0 \leq j < k$ and $M, \pi(k) \not\models \phi_2$. By the induction hypothesis, there exist an l-counterexample Π_j for ϕ_1 at $\pi(j)$, $0 \leq j < k$ and a l-counterexample Π_k for ϕ_2 at $\pi(k)$. Let then Π be the multi-sequence $\Pi = [\Pi_0, \Pi_1, \ldots, \Pi_k, \pi(k+1), \pi(k+2), \ldots]$. Clearly, Π is an infinite fair multi-path and $or(\Pi) = s$. Hence, Π is an l-counterexample for ϕ at s.

3. $\phi = \mathbf{AX}\phi_1$. If $M, s \not\models \mathbf{AX}\phi$, then there exists a fair path π such that $\pi(0) = s$ and $M, \pi(1) \not\models \phi_1$. By the induction hypothesis, there exists an l-counterexample Π' for ϕ_1 such that $or(\Pi') = \pi(1)$. Hence, the multi-path $\Pi = [\pi(0), \Pi', \pi(2), \pi(3), \cdots]$ is an l-counterexample for ϕ such that $or(\Pi) = s$.

4. $\phi = \phi_1 \vee \phi_2$. $M, s \not\models \phi$ implies that $M, s \not\models \phi_i$, for $i = 1, 2$; therefore, by the induction hypothesis, there exists an l-counterexample Π_i for ϕ_i such that $or(\Pi_i) = s$, $i = 1, 2$.

   Consequently, $\Pi = \Pi_1 * \Pi_2$ is an l-counterexample for ϕ such that $or(\Pi) = s$.

5. $\phi = \phi_1 \wedge \phi_2$. W.l.o.g., $M, s \not\models \phi_1$; by the induction hypothesis, ϕ_1 has a l-counterexample Π such that $or(\Pi) = s$, which is a l-counterexample for ϕ such that $or(\Pi) = s$. □

Proposition 6.1 *Let M and M' be two FC-Kripke structures which coincide on S, S_0, L, and F. Let Π be an l-counterexample in M for a formula ϕ on $ap(\phi) \subseteq A(M)$. If Π' is a fair multi-path in M' such that $\Pi \approx_{ap(\phi)} \Pi'$, then Π' is an l-counterexample for ϕ in M'.*

Proof. We proceed by structural induction on ϕ, where at the basis are pure state formulas.

Basis. ϕ is a pure state formula. Then $\Pi = s$ for some state s and $M, s \not\models \phi$. Since $\Pi' \approx_{A'} \Pi$, by Definition 6.2, $\Pi' = s'$ such that $s \approx_{A'} s'$. Hence, $L(s) \cap A' = L'(s') \cap A'$. Thus, $M, s \not\models \phi$ implies $M', s' \not\models \phi$ since ϕ is on A. Consequently, Π' is an l-counterexample for ϕ in M'.

Induction. Let ϕ be a non-pure state formula, let Π be a l-counterexample for ϕ, and let Π' be a fair multi-path such that $\Pi \approx_{A'} \Pi'$. We consider all possible cases for ϕ:

1. $\phi = \mathbf{A}(\phi_1 \mathbf{U} \phi_2)$. Either (1) or (2) must hold:

   1. there exists an integer $k \geq 0$ such that:

      a) $\Pi'(k) \approx_{A'} \Pi(k)$, where $\Pi(k)$ is a l-counterexample for $\phi_1 \vee \phi_2$ in M. Hence, by Definition 5.5 $\Pi(k)$ is both an l-counterexample for ϕ_1 and for ϕ_2 in M. Thus, by the induction hypothesis, $\Pi'(k)$ is both an l-counterexample for ϕ_1 and for ϕ_2 in M', and hence also for $\phi_1 \vee \phi_2$ in M'.

b) $\Pi'(i)$ is fair and $\Pi'(i) \approx_{A'} \Pi(i)$, where $\Pi(i)$ is an l-counterexample for ϕ_2 in M, for each $0 \leq i \leq k$. By the induction hypothesis, $\Pi'(i)$ is an l-counterexample for ϕ_2 in M'.

c) $\Pi'(j)$, for each $j > k$, is a state in M' as $\Pi'(j) \approx_{A'} \Pi(j)$.

2. $\Pi'(i) \approx_{A'} \Pi(i)$, where $\Pi(i)$ is an l-counterexample for ϕ_2, for each $i \geq 0$. By induction hypothesis, $\Pi'(i)$ is an l-counterexample for ϕ_2 in M'.

Hence, by Definition 5.5, Π' is an l-counterexample for ϕ in M'.

2. $\phi = \mathbf{A}(\phi_1 \mathbf{V} \phi_2)$. Π' is infinite, and there exists a $k \geq 0$ such that every $\Pi(j)$, $0 \leq j < k$, is an l-counterexample for ϕ_1, $\Pi(k)$ is an l-counterexample for ϕ_2, and every $\Pi(\ell)$ is a state, for $\ell > k$. Since $\Pi \approx_{A'} \Pi'$ and ϕ is on A, by the induction hypothesis $\Pi'(j)$ is a l-counterexample for ϕ_1 in M', for every $0 \leq j < k$, and $\Pi'(k)$ is a l-counterexample for ϕ_2 in M'. Moreover, $\Pi'(\ell)$ must be a state in M' for every $\ell > k$. Hence, Π' is a l-counterexample for ϕ in M'.

3. $\phi = \mathbf{AX}\phi_1$. Hence, $\Pi'(1)$ is an l-counterexample for ϕ_1. By the induction hypothesis, $\Pi'(1)$ is an l-counterexample for ϕ_1 in M'. Moreover, due to $\Pi(i) \approx_{A'} \Pi'(i)$, Π' is infinite and $\Pi'(i)$ is a state, for $i \neq 1$. Therefore, by Definition 5.5, Π' is an l-counterexample for ϕ in M'.

4. $\phi = \phi_1 \vee \phi_2$. Suppose w.l.o.g. that ϕ_1 is a non-pure state formula. Since Π is an l-counterexample for ϕ in M, $\Pi = \Pi_1 * \Pi_2$ where Π_i is an l-counterexample for ϕ_i in M, $i = 1, 2$. $\Pi \approx_{A'} \Pi'$ implies that $\Pi' = \Pi'_1 * \Pi'_2$ such that $\Pi'_i \approx_{A'} \Pi_i$; hence, by the induction hypothesis, Π'_i is an l-counterexample for ϕ_i in M', for $i = 1, 2$. Consequently, Π' is an l-counterexample for ϕ in M'.

5. $\phi = \phi_1 \wedge \phi_2$. Π is an l-counterexample in M for either ϕ_1 or ϕ_2. Hence, by the induction hypothesis, Π' is an l-counterexample in M' for either ϕ_1 or ϕ_2. Thus, by Definition 5.5, Π' is an l-counterexample for ϕ in M'. □

Lemma 6.5 *Let $\mathcal{R} = \langle \mathcal{P}, F, \phi \rangle$ be a PRP for which some repair exists. Then, every counterexample Π for ϕ in $M_F(\mathcal{P})$ is infinite.*

Proof. Towards a contradiction, suppose there exists a single state counterexample $\Pi = s$ for ϕ in $M_F(\mathcal{P})$. By Definition 5.6, $or(\Pi) = s$, and $or(\Pi) \in S_0(M_F(\mathcal{P}))$. Since $M_F(\mathcal{P})$ and $M_F(\mathcal{P}^{\overline{\alpha}})$ for a repair $\overline{\alpha}$ coincide on S_0 and L, clearly Π is also a counterexample for ϕ in $M_F(\mathcal{P}^{\overline{\alpha}})$. Corollary 5.2 and Proposition 4.2 imply that $\overline{\alpha}$ is not a repair for $\mathcal{R}$, which is a contradiction. □

Theorem 6.6 *Let $\overline{\alpha}$ be a repair for the PRP $\mathcal{R} = \langle \mathcal{P}, F, \phi \rangle$. Then, every counterexample Π for ϕ in $M_F(\mathcal{P})$ is infinite and executes $\overline{\alpha}$.*

Proof. Let Π be a counterexample for ϕ in $M = M_F(\mathcal{P})$. By Lemma 6.5, Π is an infinite fair multi-path. Towards a contradiction, suppose Π does not execute $\overline{\alpha}$. We prove that Π is also a fair infinite multi-path in M'. Hence, Π is a counterexample for ϕ in M', which contradicts that $\overline{\alpha}$ is a repair.

In order to establish this, it suffices to show by induction on $k \geq 0$ that every fair multi-path Π of M of depth $d(\Pi) \leq k$ which does not execute $\overline{\alpha}$ is a fair multi-path in M. (Notice that from the definition, counterexample multi-paths have finite depth.)

Basis. $k = 0$. In this case, $\Pi = s$ is a single state, and Π is by Definition 5.3 a fair multi-path in M'.

Induction. $k > 0$. Let Π be such that $d(\Pi) = k$. We have to show that (i) the main path $\mu(\Pi)$ is also a fair path in M', and (ii) $\Pi(i)$ is a fair multi-path in M', for every $i \geq 0$. For (i), we note that $\mu(\Pi)$ is a fair path in M which does not execute $\overline{\alpha}$. Therefore, by Lemma 6.4, $\mu(\Pi)$ is also a path in M'. As $F(M) = F(M')$, clearly $\mu(\Pi)$ is also fair in M'. For (ii), we note that each $\Pi(i)$ is a fair multi-path in M which does not execute $\overline{\alpha}$ such that $d(\Pi(i)) < k$, and apply the induction hypothesis. $\Box$

Lemma 7.1 *Let $\alpha = \langle k, m, \gamma\rangle$ be a correction for $\mathcal{P}$ such that $V(\alpha) \cap ap(F) = \emptyset$. Let π be a fair path in $M_F(\mathcal{P})$ which does not exploit α. Let s_0 be a state such that $s_0 \approx_{A'} \pi(0)$, where $A' = A \setminus V(\alpha)$. Let the sequence π' in $M_F(\mathcal{P}^\alpha)$ be defined by $\pi'(0) = s_0$, and $\pi'(i+1) = s_{i+1}$, for all $i \geq 0$, where s_{i+1} is the state reached from $\pi'(i)$ by executing $stmnt(k_i, \ell_i)$ in $\mathcal{P}^\alpha$, $k_i = \mathbf{ex}(\pi(i+1)))$, $\ell_i = \mathbf{b}(\pi(i+1)_{k_i})$. Then, π' is a fair path and $\pi' \approx_{A'} \pi$.*

Proof. By induction on i, we establish that $\pi(i) \approx_{A'} \pi'(i)$, for all $i \geq 0$, and $(\pi'(i-1), \pi'(i)) \in R^\alpha$, $i \geq 1$; this and the fact that $ap(F) \subseteq A'$ (as $ap(F) \cap V(\alpha) = \emptyset$) implies $\pi'(i) \models F$ iff $\pi(i) \models F$, $i \geq 0$. This proves the lemma.

Basis. For $i = 0$, this holds by hypothesis on $s_0 = \pi'(0)$.

Induction. Assume the statement holds for i, and consider $i + 1$. Since by the induction hypothesis $\pi(i) \approx_{A'} \pi'(i)$, we have $\mathbf{b}(\pi(i)) = \mathbf{b}(\pi'(i))$, i.e., the executions corresponding to π' and π up to $\pi'(i)$ resp. $\pi(i)$ have reached the same break points in all processes. By definition of R, $\pi(i+1)$ is reached from $\pi(i)$ by executing $stmnt(k_i, \ell_i)$ (for short, *stmnt*) in process P_k, where $k_i = \mathbf{ex}(\pi(i+1))$ and $\ell_i = \mathbf{b}(\pi(i))_{k_i}$. Consider the corresponding statement $stmnt(k_i, \ell_i)$ in $\mathcal{P}^\alpha$, denoted by *stmnt*$'$, and let s_{i+1} be the state reached from $\pi'(i)$ by its execution.[8] There are two cases.

(I) *stmnt*$'$ = *stmnt*. If *stmnt* is an assignment, then, since $\pi'(i) \approx_{A'} \pi(i)$ and *stmnt* may only evaluate variables in A', the transitions $(\pi(i), \pi(i+1))$ and $(\pi'(i), \pi'(i+1))$ amount to assigning the same value to the same variable x_i; hence, clearly $s_{i+1} \approx_{A'} \pi(i+1)$; otherwise, if *stmnt* is a conditional statement, for the same reason the expression $c(\mathbf{x})$ evaluates to the same at $\pi(i)$ and $\pi'(i)$, and hence $(\pi(i), \pi(i+1))$ and $(\pi'(i), \pi'(i+1))$ amount to executing the branching to the same break point; again, $s_{i+1} \approx_{A'} \pi(i+1)$.

(II) *stmnt* $\neq$ *stmnt*$'$. Then, *stmnt*$'$ is a changed statement, which is either due to an assignment modification or an assignment swap. In any case, *stmnt* and *stmnt*$'$ are assign-

[8]Note that also if semaphores would be allowed, *stmnt*$'$ would be executable at $\pi'(i)$ as $\pi'(i) \approx_{A'} \pi(i)$.

ments $x_i := g'(\mathbf{x})$ and $x_j := g(\mathbf{x})$, respectively, where $x_i, x_j \in V(\alpha)$, and their executions in P_{k_i} and P^{α}_k, respectively, lead to the same break points. Hence, clearly $s_{i+1} \approx_{A'} \pi(i+1)$. $\Box$

Proposition 6.7 *Given a concurrent program $\mathcal{P}$, a collection $\mathcal{C}$ of complex corrections $\overline{\alpha} = \alpha_1 \cdots \alpha_q$, and a counterexample Π, the corrections in $\mathcal{C}$ which are not executed by Π can be discarded in linear time.*

Proof. We sketch a possible algorithm with this property. In the first step, all distinct pairs $\pi(i), \pi(i+1)$ are determined such that $\pi(i)$ and $\pi(i+1)$ are successive states in the computation tree represented by Π. For each such pair, the statement $stmnt(k,b)$ which is executed by the transition from $\pi(i)$ to $\pi(i+1)$ can be determined easily. In step two, all these statements $stmnt(k,b)$ from step one are marked in $\mathcal{P}$. From the marked program, it is for a single correction $\alpha = \langle k, m, \gamma \rangle$ easy to tell by looking up at most two statements in $\mathcal{P}$ whether α is executed by Π or not; for a complex correction $\overline{\alpha} = \alpha_1 \cdots \alpha_q$, this is possible by a sequence of lookups for the single α_i's. This way, in step three all corrections $\overline{\alpha}$ in $\mathcal{C}$ are examined.

It is not hard to see that computing all pairs $\pi(i), \pi(i+1)$ is possible by a recursive procedure in linear time. Moreover, the lookup and marking of a statement $stmnt(b,k)$ in $\mathcal{P}$ can be done in constant time, provided the representation of $\mathcal{P}$ allows for random access to its statements. (A random access structure can be built in linear time; alternatively, a two-dimensional Boolean array T can be used where $T(k,b)$ tells whether statement $stmnt(k,b)$ was executed by some transition in Π.) Thus, also step two can be done in linear time. Since determining whether $\overline{\alpha} = \alpha_1 \cdots \alpha_q$ is executed by Π takes at most $2{\cdot}q$ lookups in $\mathcal{P}$, it follows that all complex corrections not executed by Π can be discarded from $\mathcal{C}$ in linear time.

Note that in the case where $\mathcal{C}$ has few and small corrections, i.e., the total number of simple component corrections α_i occurring in $\mathcal{C}$ is bounded by a constant, a natural variant of the algorithm is the following. Steps two and three are replaced by a test for each $stmnt(k,b)$ from step one and correction $\overline{\alpha}$ in CTL whether $\overline{\alpha}$ is executed by Π by virtue of $stmnt(k,b)$. Each such test is possible in constant time. Observe that the resulting algorithm does not access $\mathcal{P}$ at all.

We finally remark that for inputs where $\mathcal{P}$ is not available, $O(n \log n)$ algorithms for correction execution are possible by using sorting techniques. $\Box$

Theorem 6.8 *Let $\overline{\alpha}$ be a repair for the PRP $\mathcal{R} = \langle \mathcal{P}, F, \phi \rangle$ such that $V(\overline{\alpha}) \cap (ap(\phi) \cup ap(F)) = \emptyset$. Then, every counterexample Π for ϕ in $M_F(\mathcal{P})$ exploits $\overline{\alpha}$.*

Proof. Let Π be a counterexample for ϕ in $M = M_F(\mathcal{P})$. By way of contradiction, suppose that Π does not exploit $\overline{\alpha}$. We prove that $\overline{\alpha}$ is a weak correction w.r.t. Π, i.e., there exists a fair multi-path Π' in $M' = M_F(\mathcal{P}^{\overline{\alpha}})$ such that Π and Π' are equivalent on $A' = A \setminus V(\overline{\alpha})$. Thus, a contradiction arises by Proposition 6.2.

We show by induction on the depth $d(\Pi)$ of Π that if Π is a fair multi-path in M which does not exploit $\overline{\alpha}$ and s_0 is a state such that $s_0 \approx_{A'} or(\Pi)$, then there exists a fair multi-path Π' in M' such that $\Pi \approx_{A'} \Pi'$ and $or(\Pi') = s_0$. (Notice that, from the definition, every counterexample has finite depth.) In case of the counterexample, choose $s_0 = or(\Pi)$.

Basis. $d(\Pi) = 0$. Then, $\Pi = s$ is a state. Clearly, $\Pi' = s_0$ is the desired fair multi-path in M'.
Induction. Consider $d(\Pi) > 0$. Then, $\Pi = [\Pi_0, \Pi_1, \ldots]$ is an infinite multi-path such that $\mu(\Pi)$ is a fair path which does not exploit $\overline{\alpha}$ and every Π_i is a fair multi-path in M which does not exploit $\overline{\alpha}$. By Lemma 7.1, there exists a path $\pi^{(1)}$ for $\alpha = \alpha_1$ such that $\pi^{(1)} \approx_{A'_1} \pi^{(0)} = \mu(\Pi)$, where $A'_1 = A \setminus V(\alpha_1)$, $\pi^{(1)}(0) = s_0$, and $\pi^{(1)}$ is fair in $M_F(\mathcal{P}^{\alpha_1})$. By repeated application of Lemma 7.1, we obtain that for every $i = 1, \ldots, q$ there exists a fair path $\pi^{(i)}$ in $M_F(\mathcal{P}^{\alpha_1 \cdots \alpha_i})$ such that $\pi^{(i)} \approx_{A'_i} \pi^{(i-1)}$, where $A'_i = A \setminus V(\alpha_i)$ and $\pi^{(i)}(0) = s_0$. It follows that $\pi^{(q)} \approx_B \pi^{(0)}$ for $B = A \setminus \bigcup_i V(\alpha_i)$. Since $B = A'$ and $\pi^{(0)} = \mu(\Pi)$, it follows that the states $\pi^{(q)}(i)$ and $\mu(\Pi)(i)$ satisfy $\pi^{(q)}(i) \approx_{A'} \mu(\Pi)(i)$, for every $i \geq 0$. Hence, by the induction hypothesis, for every $i \geq 0$ there exists a fair multi-path Π'_i in M' such that $or(\Pi'_i) = \pi^{(q)}(i)$ and $\Pi_i \approx_{A'} \Pi'_i$. Let $\Pi' = [\Pi'_0, \Pi'_1, \ldots]$. It is easily verified that Π' is a fair multi-path in M' such that $\Pi \approx_{A'} \Pi'$ and $or(\Pi') = \pi^{(q)}(0) = s_0$. This concludes the induction and the proof of the theorem. □

Proposition 6.9 *Given a concurrent program $\mathcal{P}$, a collection $\mathcal{C}$ of complex corrections $\overline{\alpha} = \alpha_1 \cdots \alpha_q$, and a counterexample Π, the corrections in $\mathcal{C}$ which are not exploited by Π can be discarded in linear time.*

Proof. The algorithm is similar to the one sketched in the proof of Proposition 6.7. As there, in the first step all distinct pairs $\pi(i), \pi(i+1)$ are determined such that $\pi(i)$ and $\pi(i+1)$ are successive states in the computation tree represented by Π. In the second step, by a scan through these pairs, the set of variables $EV(\Pi)$ which are evaluated in at least one transition $\pi(i), \pi(i+1)$ can be easily determined by referring to the program $\mathcal{P}$. From $EV(\Pi)$, it is easy to verify whether Π exploits a single correction α by checking whether some $x_i \in V(\alpha)$ occurs in $EV(\Pi)$. For a complex correction $\overline{\alpha} = \alpha_1 \cdots \alpha_q$, this can be done analogously by taking $V(\overline{\alpha})$ in place of $V(\alpha)$.

Step one is possible in linear time, and also step two can be done in linear time if $\mathcal{P}$ is suitably represented, such that random access to its statements $stmnt(k, b)$ is possible; here, multiple examination of a $stmnt(k, b)$ in constructing $EV(\Pi)$ can be avoided by marking $stmnt(k, b)$ in $\mathcal{P}$ after the first access to it. Step three is also possible in linear time, by considering the corrections $\overline{\alpha}$ in $\mathcal{C}$ one by one. The result follows. □

References

[1] Aiello, L.C., Nardi, D., Perspectives in Knowledge Representation. *Applied Artificial Intelligence*, 5(1):29–44, 1991.

[2] Ben-Ari, M., Manna, Z., Pnueli, A., The Temporal Logic of Branching Time, *Acta Informatica,* 20:207–226, 1983.

[3] Brewka, G., Konolige, K., An Abductive Framework for Generalized Logic Programs and Other Nonmonotonic Systems. In *Proc. IJCAI '93*, pp. 9–17, 1993.

[4] Brewka, G., Dix, J., Konolige, K., Nonmonotonic Reasoning – An Overview, chapter 5: Abduction, pp. 65–86. CSLI Lecture Notes 73, CSLI Publications, Stanford University, 1997.

[5] Boutilier, C., Becher, V., Abduction as Belief Revision, *Artificial Intelligence*, 77:43–94, 1997.

[6] Browne, M.C., Clarke E. M., Dill, D., Checking the Correctness of Sequential Circuits. In *Proc. 1985 International Conference on Computer Design*, Port Chester, New York, October 1985, IEEE.

[7] Buccafurri, F., Eiter, T., Gottlob, G., Leone, N., On Witnesses of CTL Formulas, manuscript, in preparation, 1997.

[8] Burch, J. R., Clarke E. M., McMillan, K. L., Dill, D. L., Hwang, J., Symbolic Model Checking: 10^{120} States and Beyond, *Information and Computation*, 98(2):142–170,1992.

[9] Clarke E. M., Emerson E. A., Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981,* LNCS 131, 1981.

[10] Clarke E. M., Emerson E. A., Sistla, A. P., Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, *ACM Transactions on Programming Languages and Systems,* 8(2):244–263, 1986.

[11] Clarke, E.M., Grumberg, O., Long, D.E., Verification Tools for Finite-State Concurrent Systems. In: *A Decade of Concurrency - Reflections and Perspectives.* LNCS 803, Bakker, J.W. de, Roever, W.P. de, Rozenberg, G. (eds), pp. 124–175, 1994.

[12] Clarke, E.M., Grumberg, O., Long, D.E., Model Checking. In: *Deductive Program Design, M. Broy (ed), Proc. NATO ASI Series F*, vol. 152, Springer, 1996.

[13] Clarke, E.M., Grumberg, O., Long, D.E., Model Checking and Abstraction, *ACM Transaction on Programming Languages and Systems* 16(5):1512–1542, 1994.

[14] Clarke, E.M., Grumberg, O., McMillan, K., Zhao, X., Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In: *Proc. 32nd ACM/SIGDA Design Automation Conference 1995 (DAC '95)*, ACM Press. Also Technical Report CMU-CS-94-204, Carnegie Mellon University, Pittsburgh, PA, 1994.

[15] Console, L., Theseider Dupré, D., Friedrich, G., Model-Based Diagnosis Meets Error Diagnosis in Logic Programs. In: *Proc. IJCAI '93*, pp. 1494–1499, 1993.

[16] Console L., Theseider Dupré, D., Torasso, P., On the Relationship Between Abduction and Deduction, *J. Logic & Computation*, 1(5):661–690, 1991.

[17] Coudert, O., Berthet, C., Madre, J.C., Verification of Synchronous Sequential Machines, Based on Symbolic Execution. In: *Proc. 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, LNCS 407, 1989.

[18] Eiter, T., Gottlob, G., The Complexity of Logic-Based Abduction, *Journal of the ACM*, 42(1):3–42, 1995.

[19] Eiter, T., Gottlob, G., Leone, N., Semantics and Complexity for Abduction from Default Theories, *Artificial Intelligence*, 90:177–222, 1997.

[20] Emerson, E.A., Temporal and Modal Logics. In: *Handbook of Theoretical Computer Science*, J. van Leeuwen (ed), ch. 16, 995–1072, Elsevier Science Publishers, 1990.

[21] Emerson, E.A., Halpern, J.Y., "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic, *Journal of the ACM*, 33(1):151–178, 1986.

[22] Emerson, E.A., Clarke, E.M., Using Branching-Time Logic to Synthesize Synchronization Skeletons, *Science of Computer Programming*, 2:241–266, 1982.

[23] Friedrich, G., Gottlob, G., and Nejdl, W., Hypothesis Classification, Abductive Diagnosis, and Therapy. In: *Proc. International Workshop on Expert Systems in Engineering*, LNCS/LNAI 462, Gottlob, G., Nejdl, W. (eds), pp. 69–78, 1990.

[24] Friedrich, G., Gottlob, G., and Nejdl, W., Formalizing the Repair Process – Extended Report, *Annals of Mathematics and Artificial Intelligence*, 11:187–201, 1994.

[25] Gärdenfors, P., *Knowledge in Flux*, Bradford Books, MIT Press, 1988.

[26] Grumberg, O., Long, D.E., Model Checking and Modular Verification, *ACM Transaction on Programming Languages and Systems* 16(3):843-871, May 1994.

[27] Hamscher, W., editor, *Readings in Model-Based Diagnosis*, Morgan Kaufman Publ., 1992.

[28] Immerman, N., Vardi, M., Model Checking and Transitive-Closure Logic, 1997. In: *Proceedings Computer Aided Verification (CAV '97)*, pp. 291–302, 1997.

[29] Inoue K., Sakama, Ch., Abductive Framework for Nonmonotonic Theory Change. In *Proc. IJCAI '95*, pp. 204–210, 1995.

[30] Katsuno, H., Mendelzon, A. O., Propositional Knowledge Base Revision and Minimal Change, *Artificial Intelligence*, 52:253–294, 1991.

[31] Konolige, K., Abduction versus Closure in Causal Theories, *Artificial Intelligence*, 53:255–272, 1992.

[32] Kupferman, O., Grumberg, O., Buy One, Get One Free!! *Journal of Logic and Computation*, 6(4):523–539, 1996.

[33] Lobo, J., Uzcátegui, C., Abductive Change Operators, *Fundamenta Informaticae*, 27:385–412, 1996.

[34] Lobo, J., Uzcátegui, C., Abductive Consequence Relations, *Artificial Intelligence*, 27:385–412, 1997.

[35] Long, D.E., *Model Checking, Abstraction and Compositional Reasoning* PhD thesis, Carnegie Mellon University, 1993.

[36] McMillan, K.L., *Symbolic Model Checking: An Approach to the State Explosion Problem* PhD thesis, Carnegie Mellon University, 1992.

[37] Mishra, B., Clarke, E.M., Hierarchical Verification of Asynchronous Circuits Using Temporal Logic, *Theoretical Computer Science*, 38:269–291, 1985.

[38] Peterson, G., Myths About the Mutual Exclusion Problem, *Information Processing Letters*, 12(3):115–116, 1981.

[39] Pnueli, A., The Temporal Semantics of Concurrent Programs, *Theoretical Computer Science*, 13:45–60, 1981.

[40] Poole D., A Logical Framework for Default Reasoning, *Artificial Intelligence*, 36:27–47, 1988.

[41] Provan, G.M., Poole, D., The Utility of Consistency-Based Diagnostic Techniques. In: *Proc. KR '91*, pp. 461–472, Morgan Kaufmann, 1991.

[42] Selman, B., and Levesque, H., Support Set Selection for Abductive and Default Reasoning, *Artificial Intelligence*, 82:259–272, 1996.

[43] Sistla, A., P., Clarke, E.M., The Complexity of Propositional Linear Temporal Logics, *Journal of the ACM*, 32(3):733–749, 1985.

[44] Stumptner, M., Wotawa, F., Model-Based Program Debugging and Repair. In: *Proc. Ninth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE '96)*, Fukuoka, Japan, pp. 155–160, 1996.

[45] Stumptner, M., Wotawa, F., A Model-Based Approach to Software Debugging. In: *Proceedings Seventh International Workshop on Diagnosis (DX '96)*, Val Morin, Canada, Suhayya Abu-Hakima (ed), pp. 214–223, 1996.

[46] Vardi, M.Y., Wolper, P., An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the First Annual Symposium on Logic in Computer Science (LICS '86)*, pp. 332–344, IEEE Computer Society Press, 1986.