Form Methods Syst Des (2015) 47:26–50
DOI 10.1007/s10703-015-0223-6

Program repair without regret

Christian von Essen · Barbara Jobstmann

Published online: 24 February 2015

Abstract We present a new and flexible approach to repair reactive programs with respect to a specification. The specification is given in linear-temporal logic. Like in previous approaches, we aim for a repaired program that satisfies the specification and is syntactically close to the faulty program. The novelty of our approach is that it produces a program that is also semantically close to the original program by enforcing that a subset of the original traces is preserved. Intuitively, the faulty program is considered to be a part of the specification, which enables us to synthesize meaningful repairs, even for incomplete specifications. Our approach is based on synthesizing a program with a set of behaviors that stay within a lower and an upper bound. We provide an algorithm to decide if a program is repairable with respect to our new notion, and synthesize a repair if one exists. We analyze several ways to choose the set of traces to leave intact and show the boundaries they impose on repairability. We also discuss alternative notions based on reward models to obtain repair systems that behave similar to the original system. We have evaluated the approach on several examples.

Keywords Program repair · Reactive systems · Synthesis · Games · Linear temporal logic

1 Introduction

Writing a program that satisfies a given specification usually involves several rounds of debugging. Debugging a program is often a difficult and tedious task: the programmer has to find the bug, localize the cause, and repair it. Model checking [12,36] has been successfully used to expose bugs in a program. There are several approaches [9,13,20,23,26,39,40,48] to automatically find the possible location of an error. We are interested in automatically repairing a program. Automatic program repair takes a program and a specification and searches for a cor-

C. von Essen
Google Zurich, Zurich, Switzerland
e-mail: vonEssen.christian@gmail.com

B. Jobstmann (✉)
EPFL, Lausanne, Switzerland
e-mail: barbara.jobstmann@epfl.ch

rect program that satisfies the specification and is syntactically close to the original program (cf. [3,16,17,21,24,25,43,45]). Existing approaches follow the same idea: first, introduce freedom into the program (e.g., by describing valid edits to the program), and then search for a way of resolving this freedom such that the modified program satisfies the specification or the given test cases. While these approaches have been shown very effective, they suffer from a common weakness: they give little or no guarantees on preserving correct behaviors (i.e., program behaviors that do not violate the specification). Therefore, a user of a repair procedure may later *regret* having applied a fix to a program because it introduced new bugs by modifying behaviors that are not explicitly specified or for which no test case is available. The approach presented by Chandra et al. [17] provides some guarantees by requiring that a valid repair needs to pass a set of positive test cases. Correct behaviors outside these test cases are left unconstrained and the repair can thus change them unpredictably. Ebnenasir et al. [4,19] also observed the importance of maintaining existing behaviors. They present an approach to repair program with respect to UNITY properties that guarantees that any arbitrary UNITY properties is preserved. This is achieved by ensuring that the revised program does not exhibit new behaviors. For reactive programs, it is in many case unavoidable to introduce new behaviors, as the program needs to respond to every possible input sequence correctly. E.g., if the program violates a desired property on a particular input sequence, the revised version needs to respond differently on the exact same sequence and therefore exhibit a new behavior. More recently, Bonakdarpour et al. showed how to lift this restriction in the context of fault-tolerant distributed systems [7] and authentication protocols [5] using domain knowledge. None of the approaches allows the user to specify the behaviors that need to be preserved.

We present the first repair approach for reactive systems that constructs repairs that are guaranteed to satisfy the specification and that are not only syntactically, but also semantically close to the original program. We achieve this by allowing the user to defined a set of (correct) traces that any valid repair needs to include. The key benefits of our approach are: (i) it maintains correct program behavior, (ii) it is robust w.r.t. generous program modifications, i.e., it does not produce degenerated programs if given too much freedom in modifying the program, (iii) it works well with incomplete specifications, because it considers the faulty program as part of the specification and preserves its core behavior, and finally (iv) it is easy to implement on top of existing technology. We believe that our framework will prove useful because it does not require a complete specification by taking the program as part of the specification. It therefore makes writing specifications for programs easier. Furthermore, specifications are often given as conjunctions of smaller specifications that are verified individually. In order to keep desired behaviors, classical repair approaches repair a program with respect to the entire specification. Our approach can provide meaningful repair suggestions while focusing only on parts of the specification.

1.1 Outline

In Sect. 2 we introduce the used notation and present known results related to repair and synthesis. Section 3 presents an example motivating the need for a new definition of program repair. In Sect. 4, we define a new notion of repair for reactive programs and present an algorithm to compute such repairs. The algorithm is based on synthesizing repairs with respect to a lower and an upper bound on the set of generated traces. We present choices for possible lower bounds in Sect. 5 and discuss limitations of any repair approach that is based on preserving part of the program's behavior in Sect. 6. Section 7 is dedicated to alternative notions of "semantically close". Finally, we present experimental results based on a prototype employing the NuSMV model checker in Sect. 8. Our implementation is based on the idea of syntax-guided synthesis [1], which syntactically restricts the possible repair candidates.

This article is based on [44]. It adds further analysis with respect to optimal repair, suggests alternative approaches to define "close" repairs, provides more details about the experimental evaluation, and some new insides to improve the implementation.

2 Preliminaries

2.1 Words, languages, alphabet restriction and extension

Let AP be the finite set of *atomic propositions*. We define the *alphabet* over AP (denoted Σ_{AP}) as the set of all evaluations of AP, i.e., $\Sigma_{AP} = 2^{AP}$. If AP is clear from the context or not relevant, then we omit the subscript in Σ_{AP}. A *word* w is an infinite sequence of letters from Σ. We use Σ^ω to denote the set of all words. A *language* L is a set of words, i.e., $\mathrm{L} \subseteq \Sigma^\omega$. Given a word $w \in \Sigma^\omega$, we denote the letter at position i by w_i, where w_0 is the first letter. We use $w_{..i}$ to denote the prefix of w up to position i, and $w_{i..}$ to denote the suffix of w starting at position i. Given a set of propositions $\mathrm{I} \subseteq AP$, we define the I-*restriction* of a word $w \in \Sigma_{AP}^\omega$, denoted by $w{\downarrow_\mathrm{I}}$, as $w{\downarrow_\mathrm{I}} = l_0 l_1 \cdots \in \Sigma_\mathrm{I}^\omega$ with $l_i = (w_i \cap \mathrm{I})$ for all $i \geq 0$. Given a language $\mathrm{L} \subseteq \Sigma_{AP}^\omega$ and a set $\mathrm{I} \subseteq AP$, we define the I-*restriction* of L, denoted by $\mathrm{L}\downarrow_\mathrm{I}$, as the set of I-restrictions of all the words in L, i.e., $\mathrm{L}\downarrow_\mathrm{I} = \{w{\downarrow_\mathrm{I}} \mid w \in \mathrm{L}\}$. Given a word $w \in \Sigma_\mathrm{I}^\omega$ over a set of propositions $\mathrm{I} \subseteq AP$, we use $w{\uparrow_{AP}}$ to denote the *extension* of w to the alphabet Σ_{AP}, i.e., $w{\uparrow_{AP}} = \{w' \in \Sigma_{AP}^\omega \mid w'{\downarrow_I} = w\}$. Extension of a language $\mathrm{L} \subseteq \Sigma_\mathrm{I}^\omega$ is defined analogously, i.e., $\mathrm{L}\uparrow_{AP} = \{w{\uparrow_{AP}} \mid w \in \mathrm{L}\}$. A language $\mathrm{L} \subseteq \Sigma_{AP}^\omega$ is called I-*deterministic* for some set $\mathrm{I} \subseteq AP$ if for each word $v \in \Sigma_\mathrm{I}^\omega$ there is at most one word $w \in \mathrm{L}$ such that $w{\downarrow_\mathrm{I}} = v$. A language L is called I-*complete* if for each input word $v \in \Sigma_\mathrm{I}^\omega$ there exists at least one word $w \in \mathrm{L}$ such that $w{\downarrow_I} = v$.

2.2 Machines, automata, and formulas

A *(finite state) machine* is a tuple $\mathcal{M} = (M, \Sigma_\mathrm{I}, \Sigma_\mathrm{O}, m_0, \delta, \Omega)$, where M is a finite set of *states*, $\Sigma_\mathrm{I}(= 2^I)$ and $\Sigma_\mathrm{O}(= 2^O)$ are the *input* and the *output alphabet*, respectively, $m_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma_\mathrm{I} \to Q$ is the *transition function*, and $\Omega : Q \times \Sigma_\mathrm{I} \to \Sigma_\mathrm{O}$ is the *output function*. The *input signals* I and the *output signals* O *of* $\mathcal{M}$ are required to be distinct, i.e., $I \cap O = \emptyset$. A *run* ρ of $\mathcal{M}$ on an input word $w \in {\Sigma_\mathrm{I}}^\omega$ is the sequence of states that the machine visits while reading the input word, i.e., $\rho = q_0 q_1 \cdots \in M^\omega$ such that $\delta(q_i, w_i) = q_{i+1}$ for all $i \geq 0$. The *output word* $\mathcal{M}$ produces on w (denoted by $\mathcal{M}_\mathrm{O}(w)$) is the sequence of output letters that the machine produces while reading the input word, i.e., for the run $q_0 q_1 \ldots$ of $\mathcal{M}$ on w, the output word is $\mathcal{M}_\mathrm{O}(w) = l_0 l_1 \cdots \in \Sigma_\mathrm{O}^\omega$ with $l_i = \Omega(q_i, w_i)$ for all $i \geq 0$. The *combined input output word* $\mathcal{M}$ produces on w is defined as $\mathcal{M}(w) := (i_0 \cup o_0)(i_1 \cup o_1) \ldots \in \Sigma_{AP}^\omega$, where $w = i_0 i_1 \ldots$ and $\mathcal{M}_\mathrm{O}(w) = o_0 o_1 \ldots$. We denote by $\mathrm{L}(\mathcal{M})$ the *language* of $\mathcal{M}$, i.e., the set of combined input/output words $\mathrm{L}(\mathcal{M}) = \{\mathcal{M}(w) \mid w \in \Sigma_I^\omega\}$.

A *Büchi automaton* is a tuple $\mathcal{A} = (\mathrm{S}, \Sigma, \mathrm{s}_0, \Delta, \mathrm{F})$ where S is a finite set of *states*, Σ is the *alphabet*, $\mathrm{s}_0 \in \mathrm{S}$ is the *initial state*, $\Delta \subseteq \mathrm{S} \times \Sigma \times \mathrm{S}$ is the *transition relation*, and $\mathrm{F} \subseteq \mathrm{S}$ is the set of *accepting states*. A run of $\mathcal{A}$ on a word $w \in \Sigma^\omega$ is a sequence of states $\mathrm{s}_0\, \mathrm{s}_1\, \mathrm{s}_2 \ldots \in \mathrm{S}^\omega$ that respects the transition relation, i.e., $(\mathrm{s}_i, w_i, \mathrm{s}_{i+1}) \in \Delta$ for all $i \geq 0$. A word is accepted by a Büchi automaton $\mathcal{A}$ if there exists a run $s_0\, s_1 \ldots$ that visits infinitely often one of the accepting states, i.e., $s_i \in F$ for infinitely many i. We denote by $L(\mathcal{A})$ the language of the Büchi automaton, i.e., the set of words accepted by $\mathcal{A}$. A language that is accepted by a Büchi automaton is called ω-regular. A Büchi automaton $\mathcal{A} = (S, \Sigma, s_0, \Delta, F)$ is called *deterministic* is the transition relation Δ maps every state and letter pair to at most one successor state, i.e, Δ can be seen as a (partial) function $\Delta : S \times \Sigma \to S$. A *safety automaton* is a Büchi automaton $\mathcal{A} = (S, \Sigma, s_0, \Delta, F)$ without transitions from non-accepting to accepting states, i.e., $\forall (s, w, s') \in \Delta : s \notin F \to s' \notin F$. Note that in order for a run to be accepting is has to visiting only accepting states. A *parity automaton* is like a Büchi automaton $\mathcal{A} = (S, \Sigma, s_0, \Delta, \lambda)$ but with a different acceptance condition: the set of accepting states is replaced by a *parity function* $\lambda : S \to \{0, \ldots, d\}$ that maps every state to a number from finite set of natural numbers $\{0, \ldots, d\}$. A word is accepted by a parity automaton $\mathcal{A}$ if there exists a run $s_0\, s_1 \ldots$ such that the maximal number assigned (by the parity function) to a state that is visited infintely often is even, i.e., $\max_{\{s \mid \forall i \geq 0 \exists j \geq i : s = s_j\}} \lambda(s)$ is even.

We use linear temporal logic (LTL) [33] over a set of atomic propositions AP to specify the desired behavior of a machine. An LTL formula may refer to atomic propositions, Boolean operators, and the temporal operators *next* X and *until* U. Formally, an LTL formula φ is defined inductively as $\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$ with $p \in AP$. The semantics of an LTL formula φ is given with respect to words $w \in \Sigma_{AP}^{\omega}$ using the satisfaction relation $\models$. As usual, we define it inductively over the structure of the formula as follows: (i)$w \models p$ iff $p \in w_0$, (ii)$w \models \neg\varphi$ iff $w \not\models \varphi$, (iii)$w \models \varphi_1 \wedge \varphi_2$ iff $w \models \varphi_1$ and $w \models \varphi_2$, (iv)$w \models \mathsf{X}\,\varphi$ iff $w_{1..} \models \varphi$, and (v)$w \models \varphi_1\,\mathsf{U}\,\varphi_2$ iff $\exists i \geq 0 : w_{i..} \models \varphi_2$ and $\forall j, 0 \leq j < i : w_{j..} \models \varphi_1$. The Boolean operators $\vee$, $\to$, and $\leftrightarrow$ are derived as usual. We use the common abbreviations for false, true, F, and G, i.e., $\mathsf{false} := p \wedge \neg p$, $\mathsf{true} := \neg\mathsf{false}$, $\mathsf{F}\,\varphi := \mathsf{true}\,\mathsf{U}\,\varphi$, and $\mathsf{G}\,\varphi := \neg\,\mathsf{F}\,\neg\varphi$. For instance, every word w with $p \in w_i$ for some $i \geq 0$ satisfies $\mathsf{F}\,p$. Dually, every word with $p \notin w_i$ for all $i \geq 0$ satisfies $\mathsf{G}\,\neg p$. The language of φ, denoted $L(\varphi)$, is the set of words satisfying formula φ. For every LTL formula φ one can construct a Büchi automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L(\varphi)$ [30,47]. For every LTL formula φ one can construct a deterministic parity automaton such that $L(\mathcal{A}) = L(\varphi)$ [31,41]. We will use the following lemma in Sect. 4.

Lemma 1 (Machine languages) *The language* $L(\mathcal{M})$ *of any machine* $\mathcal{M} = (M, \Sigma_I, \Sigma_O, m_0, \delta, \Omega)$ *is I-deterministic (input deterministic) and I-complete (input complete).*

Proof It follows directly from the definition (i.e., from the fact that δ is a complete function). □

2.3 Realizability and synthesis problem

The synthesis problem [15] asks to construct a system that satisfies a given formal specification.

Definition 1 (*Realizability and Synthesis*) Given a language language L over the atomic propositions AP partitioned into input and output propositions, i.e., $AP = I \cup O$, and a finite state machine M with input alphabet Σ_I and output alphabet Σ_O, we say that $\mathcal{M}$ *implements* (*realizes, or satisfies*) L, denoted by $\mathcal{M} \models L$, if $L(\mathcal{M}) \subseteq L$. We say language L is *realizable* if there exists a machine M that implements L. An LTL-formula φ is realizable if $L(\varphi)$ is realizable.

Theorem 1 (Synthesis Algorithms [8,34,37]) *There exists a deterministic algorithm that checks whether a given LTL-formula (or an ω-regular language) φ is* realizable*, i.e., there exists a machine $\mathcal{M}$ such that $\mathcal{M} \models \varphi$. If φ is realizable, then the algorithm constructs $\mathcal{M}$.*

The synthesis problem can be solved by computing winning strategies in turn-based games.

2.4 Games and winning objectives

Two-player turn-based games are played by two player (Player 0 and Player 1) that push a pebble along the edges of a directed graph. The graph is partitioned into player-0 and player-1 states. Depending on the position of the pebble either Player 0 (in player-0 states) or Player 1 (in player-1 states) can decide along with outgoing edge the pebble is moved to the next state. A play of the game is a sequence of states that the pebble visits. Formally, a *game* is a tuple $\mathcal{G} = (\mathrm{S}, \mathrm{S}_0, \mathrm{S}_1, \mathrm{s}_0, \Delta)$, where S is a finite set of *states* partitioned into a set S_0 of *player-0 states* and a set S_1 of *player-1 states*, s_0 is an (optional) *initial state*, and $\Delta : \mathrm{S} \times \mathrm{S}$ is a *transition relation* indicating the edges between the states. Without loss of generality, we can assume that every state has at least one outgoing edge, i.e., $\forall\, \mathrm{s} \in \mathrm{S}\, \exists\, \mathrm{s}' \in \mathrm{S} : (\mathrm{s}, \mathrm{s}') \in \Delta$. We consider several types of games. They have the same structure but differ in the used winning objective and the defined behaviors of Player 1.

A *play* is an infinite sequence $\mathrm{s}_0\, \mathrm{s}_1 \ldots$ of states that respect the transition relation, i.e., $(\mathrm{s}_i, \mathrm{s}_{i+1}) \in \Delta$ for all $i \geq 0$. A *strategy* for Player i is function $\pi : \mathrm{S}^* \mathrm{S}_i \to \mathrm{S}$ that chooses a successor state for all finite sequences of states that end in a player-i state. A strategy must prescribe only available moves, i.e., for all sequences $p \in \mathrm{S}^*$ and player-i states $\mathrm{s} \in \mathrm{S}_i$, $(\mathrm{s}, \pi(p\,\mathrm{s})) \in \Delta$. A state s_0 together with two strategies π_0 and π_1 for Player 0 and Player 1, respectively, describes a unique play $\mathrm{s}_0\, \mathrm{s}_1 \ldots$, denoted by $\mathcal{G}(\mathrm{s}_0, \pi_0, \pi_i)$, in which the strategies are respected in all positions, i.e., $\forall i \geq 0 : \mathrm{s}_i \in \mathrm{S}_0 \to \mathrm{s}_{i+1} = \pi_0(\mathrm{s}_0 \ldots \mathrm{s}_i) \wedge \mathrm{s}_i \in \mathrm{S}_1 \to \mathrm{s}_{i+1} = \pi_1(\mathrm{s}_0 \ldots \mathrm{s}_i)$.

A *winning objective* for a player is a function $f : \mathrm{S}^\omega \to \mathbb{R}$ that maps every play to real value. We call an objective *qualitative* if it maps each play only to value 0 or 1; value 1 indicates that Player 0 has *won* the play; on plays with value 0, Player 0 loses and Player 1 wins. Analogously to the acceptance conditions of automata, we use safety and parity conditions to define wining objectives. Given a set $F \subseteq \mathrm{S}$ of states, the *safety objectives* requires that only states in F are visited, i.e., $f(\mathrm{s}_0\, \mathrm{s}_1 \ldots) = 1$ if $\forall i \geq 0 : \mathrm{s}_i \in F$, otherwise $f(\mathrm{s}_0\, \mathrm{s}_1 \ldots) = 0$. Given a function $\lambda : \mathrm{S} \to \{0, \ldots, d\}$ that maps every state to a *priority*, the *parity objective* requires that of the states that are visited infinitely often, the greatest priority is even, i.e., $f(\mathrm{s}_0\, \mathrm{s}_1 \ldots) = 1$ if $\max_{\{\mathrm{s} \mid \forall i \geq 0 \exists j \geq i : \mathrm{s} = \mathrm{s}_j\}} \lambda(\mathrm{s})$ is even, otherwise $f(\mathrm{s}_0\, \mathrm{s}_1 \ldots) = 0$. We also consider the following non-qualitative objective. Given a reward function $r : \mathrm{S} \to \{0, \ldots, d\}$ that maps every state to a *reward*, the *mean-payoff objective* maps every play to the average reward seen along the play, i.e., $f(\mathrm{s}_0\, \mathrm{s}_1 \ldots) = \liminf_{n \geq 0} \frac{1}{n} \sum_{i=0}^{n-1} r(\mathrm{s}_i)$.

Given a game $\mathcal{G} = (\mathrm{S}, \mathrm{S}_0, \mathrm{S}_1, \mathrm{s}_0, \Delta)$ and a qualitative objective f, a strategy π_0 is *winning* for Player 0, if for all player-1 strategies π_1: $f(\mathcal{G}(\mathrm{s}_0, \pi_0, \pi_1)) = 1$. We say that Player 0 *wins* a game, if she has a winning strategy. Winning and winning strategy for Player 1 are defined analogously. Given a game $\mathcal{G} = (\mathrm{S}, \mathrm{S}_0, \mathrm{S}_1, \mathrm{s}_0, \Delta)$ and a non-qualitative objective f, a strategy π_0 is *optimal* for Player 0, if for all player-0 strategies π_0', $\min_{\pi_1} f(\mathcal{G}(\mathrm{s}_0, \pi_0, \pi_1)) \geq \min_{\pi_1} f(\mathcal{G}(\mathrm{s}_0, \pi_0', \pi_1))$. Optimality for Player 1 is defined analogously.

Given a deterministic (parity) automaton $\mathcal{A} = (\mathrm{S}, \Sigma_{I \cup O}, \mathrm{s}_0, \Delta, \lambda)$, there is a simple way to construct a game $\mathcal{G} = (\mathrm{S}', \mathrm{S}_0, \mathrm{S}_1, \mathrm{s}_0', \Delta')$ (as follows) with parity objectives λ' such that if Player 0 wins the game, then there exists a machine $\mathcal{M} = (M, \Sigma_I, \Sigma_O, m_0, \delta, \Omega)$ with $L(\mathcal{M}) \subseteq L(\mathcal{A})$:

$$\begin{aligned}
S' &= S_0 \cup S_1 \\
S_0 &= S \times \Sigma_I \\
S_1 &= S \times \Sigma_I \times \Sigma_O \\
\Delta' &= \{(s'_0, (s_0, w_i)) \mid w_i \in \Sigma_I\} \\
&\quad \cup \{((s, w_i), (s', w_i, w_o)) \mid \Delta(s, w_i \cup w_o, s')\} \\
&\quad \cup \{((s, w_i, w_o), (s, w'_i)) \mid w'_i \in \Sigma_I\}
\end{aligned}$$

and $\lambda'((s, w_i)) = \lambda'((s, w_i, w_o)) = \lambda(s)$. We call such a game a *synthesis game*. In practical applications, this construction is often optimized or even avoided by going directly from LTL to a game (cf. [6]).

A *Markov decision process (MDP)* is a tuple $\mathcal{M} = (S, s_0, \Sigma_O, \overline{A}, p)$, where S is a finite set of states, s_0 is an initial state, Σ is a set of *actions*, $\overline{A} : S \to \Sigma$ is the *action or enable function*, and $p : S \times \Sigma \times S \to [0; 1]$ is a *probability function*, where $p(s, a, s')$ defines the probability of moving from a state s to a successor state s' using the action a. Intuitively, an MDP can be seen as two-player games, in which the Player strictly alternate and Player 1 is replaced by a random player. In every joint step, Player 0 first chooses an action (leading to a player-1 state) and then a successor from the player-1 state is chosen according to the probability function. For more details about MDP, please see [35].

3 Example

In this section we give a simple example to motivate our definitions and highlight the differences to previous approaches such as [24].

Example 1 (Traffic Light) Assume we want to develop a sensor-driven traffic light system for a crossing of two streets. For each street entering the crossing, the system has two sets

```
typedef enum {RED, YELLOW, GREEN} traffic_light;
module Traffic (clock, sensor1, sensor2, light1, light2);
  input clock, sensor1, sensor2;
  output light1, light2;
  traffic_light reg light1, light2;
  initial begin
    light1 = RED;
    light2 = RED;
  end
  always @(posedge clock) begin
    case (light1)
      RED: if (sensor1) // Repair: if (sensor1 & !(light2==RED & sensor2))
                light1 = YELLOW;
      YELLOW: light1 = GREEN;
      GREEN: light1 = RED;
    endcase // case (light1)
    case (light2)
      RED: if (sensor2)
                   light2 = YELLOW;
      YELLOW: light2 = GREEN;
      GREEN: light2 = RED;
    endcase // case (light1)
  end // always (@posedge clock)
endmodule // traffic
```

Fig. 1 Implementation of a traffic light system and a repair

of lights (called `light1` and `light2`) and two sensors (called `sensor1` and `sensor2`). By default both lights are red. If a sensor detects a car, then the corresponding lights should change from red to yellow to green and back to red. We are given the implementation shown in Fig. 1 as starting point. It behaves as follows: for each red light, the system checks if the sensor is activated (Line 12 and 18). If yes, this light becomes yellow in the next step, followed by a green phase and a subsequent red phase. Assume we require that our implementation is safe, i.e., the two lights are never green at the same time. In LTL, this specification is written as $\varphi = \mathsf{G}(\texttt{light1} \neq \texttt{GREEN} \vee \texttt{light2} \neq \texttt{GREEN})$. The current implementation clearly does not satisfy this requirement: if both sensors detect a car initially, then the lights will simultaneously move from red to yellow and then to green, thus violating the specification.

Following the approach in [24] we introduce a non-deterministic choice into the program and then use a synthesis procedure to select among these options in order to satisfy the specification. For instance, we replace Line 12 (in Fig. 1) by `if(?)` and ask the synthesizer to construct a new expression for `?` using the input and state variables. The synthesizer aims to find a simple expression s.t. φ is satisfied. In this case one simple admissible expression is `false`. It ensures that the modified program satisfies specification φ. While this repair is correct, it is very unlikely to please the programmer because it repairs "too much": it modifies the behavior of the system on input traces on which the initial implementation was correct. We believe it is more desirable to follow the idea of Chandra et al. [17] saying that a repair is only allowed to change the behavior of incorrect executions. In our case, the repair suggested above would not be allowed because it changes the behavior on correct traces, as we will show in the next section.

4 Repair

In this section we first give a repair definition for reactive systems which follows the intuition that a repair can only change the behavior of incorrect executions. Then, we provide an algorithm to compute such repairs.

4.1 Definitions

Given a machine $\mathcal{M}$ and a specification φ, we say a machine $\mathcal{M}'$ is an exact repair of $\mathcal{M}$ if (i) $\mathcal{M}'$ behaves like $\mathcal{M}$ on traces satisfying φ and (ii) if $\mathcal{M}'$ implements φ. Intuitively, the correct traces of $\mathcal{M}$ act as a *lower bound* for $\mathcal{M}'$ because they must be included in $\mathrm{L}(\mathcal{M}')$. Analogously, $\mathrm{L}(\varphi)$ acts as an *upper bound* for $\mathcal{M}'$, i.e., it specifies the allowed traces.

Definition 2 (*Exact Repair*) A machine $\mathcal{M}'$ is an exact repair of a machine $\mathcal{M}$ for a specification φ, if (i) all the correct traces of $\mathcal{M}$ are included in the language of $\mathcal{M}'$, and (ii) if the language of $\mathcal{M}'$ is included in the language of the specification φ, i.e.,

$$\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\varphi) \subseteq \mathrm{L}(\mathcal{M}') \subseteq \mathrm{L}(\varphi)$$

Note that the first inclusion defines the behavior of $\mathcal{M}'$ on all input words to which $\mathcal{M}$ responds correctly according to φ. In other terms, $\mathcal{M}'$ has only one choice for inputs which $\mathcal{M}$ treat correctly. Figure 2 illustrates Definition 2: the two circles depict $\mathrm{L}(\mathcal{M})$ and $\mathrm{L}(\varphi)$. A repair has to (i) cover their intersection (first inclusion in Definition 2), which we depict with the striped area in the picture, and (ii) lie within $\mathrm{L}(\varphi)$ (second inclusion in Definition 2). One such repair is depicted by the dotted area on the right. Note that it covers the complete intersection of $\mathrm{L}(\psi)$ and $\mathrm{L}(\mathcal{M})$.

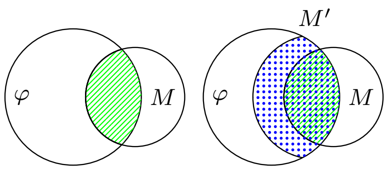

Fig. 2 Graphical representation of Def. 2

Example 2 (Traffic Light, cont.) The repair suggested in Example 1 (i.e., to replace `if (sensor1)` by `if (false)`) is not a valid repair according to Definition 2. The original implementation responds correctly, e.g., to the input trace in which `sensor1` is always high and `sensor2` is always low, but the repair produces different outputs. The initial implementation behaves correctly on any input trace on which `sensor1` and `sensor2` are never high simultaneously. Any correct repair should include these input/output traces. An exact repair (i.e, a repair according to Definition 2) replaces `if (sensor1)` by `if (sensor1 & !(light2 == RED & sensor2))`. This repair retains all correct traces while avoiding the mutual exclusion problem.

While Definition 2 excludes the undesired repair in our example, it is sometimes too restrictive and can make repair impossible, as the following example shows.

Example 3 (Definition 2 is too restrictive) Assume a machine $\mathcal{M}$ with input r and output g that always copies r to g, i.e., $\mathcal{M}$ satisfies $\mathsf{G}(r \leftrightarrow g)$. The specification requires that g is eventually high, i.e., $\varphi = \mathsf{F}\, g$. Definition 2 requires the repaired machine $\mathcal{M}'$ to behave like $\mathcal{M}$ on all traces on which $\mathcal{M}$ behaves correctly. $\mathcal{M}$ responds correctly to all input traces containing at least one r, i.e., $L(M) \cap L(\varphi) = \mathsf{F}(r \wedge g)$. Intuitively, $\mathcal{M}'$ has to mimic $\mathcal{M}$ as long as $\mathcal{M}$ still has a chance to satisfy φ (i.e., to produce a trace satisfying $\mathsf{F}(r \wedge g)$). Since $\mathcal{M}$ always has a chance to satisfy φ, $\mathcal{M}'$ has to behave like $\mathcal{M}$ in every step, therefore $\mathcal{M}'$ also violates φ, and cannot be repaired in this case.

In order to allow more repairs, we *relax* the restriction requiring that all correct traces are included. There are many possible choices for the set of traces $\mathcal{M}'$ has to mimic, therefore we leave it up to the user to choose the specific restriction.

Definition 3 (*Relaxed Repair*) Let ψ define a language (by an LTL-formula or a Büchi automaton). We say $\mathcal{M}'$ is a *repair of* $\mathcal{M}$ *with respect to* ψ *and* φ if $\mathcal{M}'$ behaves like $\mathcal{M}$ on all traces satisfying ψ and $\mathcal{M}'$ implements φ. That is, $\mathcal{M}'$ is a repair constructed from $\mathcal{M}$ iff

$$\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi) \subseteq \mathrm{L}(\mathcal{M}') \subseteq \mathrm{L}(\varphi) \tag{1}$$

In Fig. 3 we give a graphical representation of this definition. The two concentric circles depict φ and ψ. (The definition does not require that $\mathrm{L}(\psi) \subseteq \mathrm{L}(\varphi)$, but for simplicity we

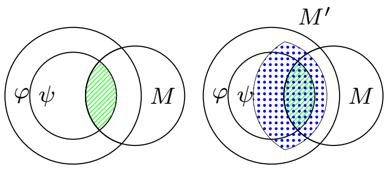

Fig. 3 Graphical representation of Def. 3

depict it like that.) The overlapping circle on the right represents $\mathcal{M}$. The intersection between ψ and $\mathcal{M}$ (the striped area in Fig. 3) is the set of traces $\mathcal{M}'$ has to mimic. On the right of Fig. 3, we show one possible repair (represented by the dotted area). The repair covers the intersection of $\mathrm{L}(\mathcal{M})$ and $\mathrm{L}(\psi)$, but not the intersection of $\mathrm{L}(\varphi)$ and $\mathrm{L}(\mathcal{M})$. The repair lies completely in $\mathrm{L}(\varphi)$. The choice of ψ influences the existence of a repair. In Sect. 5 we discuss several choices for ψ.

Example 4 (Example 3 continued) Example 3 shows that setting ψ to φ, i.e., $\mathsf{F}\, g$ in our example, can be too restrictive. If we relax ψ and require it only to include all traces in which g is true within the first n steps for some given n (i.e., $\psi = \bigvee_{0 \le i \le n} \mathsf{X}^n\, g$), then we can find a repair. A possible repair is a machine $\mathcal{M}'$ that copies r to g in the first n steps and keeps track if g has been high within these steps. In this case, $\mathcal{M}'$ continues mimicking $\mathcal{M}$, otherwise it sets g to high in step $n+1$, independent of the behavior of $\mathcal{M}$. This way $\mathcal{M}'$ satisfies the specification ($\mathsf{F}\, g$) and mimics $\mathcal{M}$ for all traces satisfying ψ.

4.2 Reduction to classical synthesis

The following theorem shows that our repair problem can be reduced to the classical synthesis problem.

Theorem 2 *Let φ, ψ be two specifications and $\mathcal{M}$, $\mathcal{M}'$ be two machines with input signals I and output signal O. Machine $\mathcal{M}'$ satisfies Formula 1* $(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi) \overset{(a)}{\subseteq} \mathrm{L}(\mathcal{M}') \overset{(b)}{\subseteq} \mathrm{L}(\varphi))$ *if and only if $\mathcal{M}'$ satisfies the following formula:*

$$\mathrm{L}(\mathcal{M}') \subseteq \underbrace{\big((\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi)){\downarrow_I}{\uparrow_{AP}} \rightarrow \mathrm{L}(\mathcal{M})\big)}_{(i)} \cap \underbrace{\mathrm{L}(\varphi)}_{(ii)} \tag{2}$$

For two languages A and B, $A \rightarrow B$ is an abbreviation for $(\Sigma^\omega \setminus A) \cup B$. Intuitively, Equation 2 requires that (i) $\mathcal{M}'$ behaves like $\mathcal{M}$ on all input words that $\mathcal{M}$ answers conforming to ψ and (ii) $\mathcal{M}$ satisfies specification φ.

Proof From left to right: We have to show that $\mathrm{L}(\mathcal{M}')$ is included in (i) and (ii). Inclusion in (ii) follows trivially from (b). It remains to show $\mathrm{L}(\mathcal{M}') \subseteq \big(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi)\big){\downarrow_I}{\uparrow_{AP}} \rightarrow \mathrm{L}(\mathcal{M})$. Let $w \in \mathrm{L}(\mathcal{M}')$. If $w \notin \big(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi)\big){\downarrow_I}{\uparrow_{AP}}$, then the implication follows trivially. Otherwise we have to show that $w \in \mathrm{L}(\mathcal{M})$. Since $w \in \big(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi)\big){\downarrow_I}{\uparrow_{AP}}$, it follows that $w{\downarrow_I} \in \big(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi)\big){\downarrow_I}$. From $w{\downarrow_I} \in \big(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi)\big){\downarrow_I}$ and the fact that $L(M)$ is input deterministic, we know that $M(w{\downarrow_I}) \in \mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi) \subseteq \mathrm{L}(\mathcal{M}')$ (due to (a)). Together with $\mathrm{L}(\mathcal{M}')$ being input deterministic, it follows that $M(w{\downarrow_I}) = M'(w{\downarrow_I}) = w$, and so $w \in \mathrm{L}(\mathcal{M})$ holds.

From right to left: We have to show (a) and (b). (b) follows trivially from $\mathrm{L}(\mathcal{M}') \subseteq (ii)$. It remains to show (a), i.e., that $\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi) \subseteq \mathrm{L}(\mathcal{M}')$. Assume a word $w \in \mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi)$, we have to show that $w \in \mathrm{L}(\mathcal{M}')$. Let $w' \in \mathrm{L}(\mathcal{M}')$ be a word such that $w{\downarrow_I} = w'{\downarrow_I}$. Note that w' exists because $\mathrm{L}(\mathcal{M}')$ is input complete. We now show that $w = w'$, which implies that $w \in \mathrm{L}(\mathcal{M}')$. Since $w \in \mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi)$, it follows that $w{\downarrow_I} (= w'{\downarrow_I}) \in (\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi)){\downarrow_I}$. Therefore, we know that $w' \in \big(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi)\big){\downarrow_I}{\uparrow_{AP}}$. From $\mathrm{L}(\mathcal{M}') \subseteq (i)$ and from $w' \in \mathrm{L}(\mathcal{M}')$, it follows that $w' \in \mathrm{L}(\mathcal{M})$. Since $\mathrm{L}(\mathcal{M})$ is input deterministic, $w \in \mathrm{L}(\mathcal{M})$, $w' \in \mathrm{L}(\mathcal{M})$, and $w{\downarrow_I} = w'{\downarrow_I}$, it follows that $w = w'$. □

This theorem leads together with [34] to the following corollary, which allows us to use classical synthesis algorithms to compute repairs.

Corollary 1 (Existence of repair) *A repair can be constructed from a machine $\mathcal{M}$ with respect to specifications ψ and φ if and only if the language*

$$\big((\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi))\downarrow_I\uparrow_{AP} \rightarrow \mathrm{L}(\mathcal{M})\big) \cap \mathrm{L}(\varphi) \tag{3}$$

is realizable.

4.3 Algorithm

Corollary 1 gives an algorithm to construct repairs based on synthesis techniques (cf. [24]). In order to compute the language defined by Formula 3, we can use standard automata-theoretic operations. More precisely, we construct a Büchi automaton $\mathcal{A}_\varphi$ recognizing φ and a Büchi automaton $\mathcal{A}_\psi$ recognizing ψ. Note that $\mathcal{M}$ is a Büchi automaton in which all states are accepting. Since Büchi automata are closed under conjunction, disjunction, projection, and complementation, we can construct an automaton for $\big(\overline{(\mathcal{M} \times \mathcal{A}_\psi)|_I} + \mathcal{M}\big) \times \mathcal{A}_\varphi$, where $A \times B$ denotes the conjunction, $A + B$ denotes the disjunction of automata A and B, $\bar{A}$ denotes the complementation of A, and $A|_I$ the projection of automaton A with respect to a set of proposition I. Once we have a Büchi automaton for the language in Formula 3, we can use Theorem 1 to synthesize a repair.

This algorithm in unlikely to scale because the complementation of a Büchi automaton induces an exponential blow-up in the worst case [18]. Furthermore, the projection operator can introduce non-determinism that can complicate the application of a synthesis procedure due to the need of an additional determinization step, leading to another exponential blow-up [32,42]. In the following we show how to obtain an efficient algorithm by avoiding complementation (Lemma 2) and projection (Lemma 3).

Lemma 2 *Given a machine $\mathcal{M}$ with input signals I and output signals O and an LTL-formula φ over the atomic propositions $AP = I \cup O$, the following equalities hold:*

$$\Sigma_{[I]}^\omega \setminus \big(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\varphi)\big)\downarrow_I = \big(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\neg\varphi)\big)\downarrow_I \tag{4}$$
$$\Sigma_{[AP]}^\omega \setminus \big(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\varphi)\big)\downarrow_I\uparrow_{AP} = \big(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\neg\varphi)\big)\downarrow_I\uparrow_{AP} \tag{5}$$

Proof Intuitively, Equation 4 means that the set of input words on which $\mathcal{M}$ behaves correctly, i.e., satisfies φ, is the complement of the set of inputs on which $\mathcal{M}$ behaves incorrectly, i.e., violates φ and therefore satisfies $\neg\varphi$. Formally, we know from the semantics of LTL that $\mathrm{L}(\neg\varphi) = \Sigma^\omega \setminus \mathrm{L}(\varphi)$, which implies that

$$\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\neg\varphi) \overset{(a)}{=} \mathrm{L}(\mathcal{M}) \cap \big(\Sigma^\omega \setminus \mathrm{L}(\varphi)\big) \overset{(b)}{=} \mathrm{L}(\mathcal{M}) \setminus \mathrm{L}(\varphi). \tag{6}$$

Equality 6.b follows from simple set theory. Furthermore, since $\mathrm{L}(\mathcal{M})$ is input deterministic and input complete, we know that

$$\forall\, w, w' \in \mathrm{L}(\mathcal{M}) : (w\downarrow_I = w'\downarrow_I) \rightarrow w = w' \tag{7}$$
$$\forall\, w \in \Sigma_{[AP]}^\omega : \exists\, w' \in \mathrm{L}(\mathcal{M}) : w\downarrow_I = w'\downarrow_I \tag{8}$$

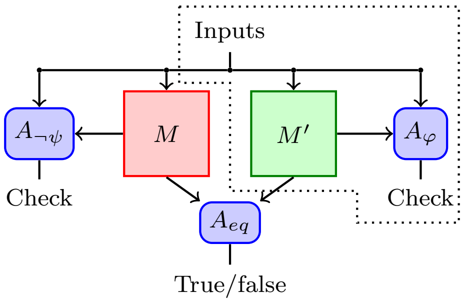

Fig. 4 Efficient implementation

We use these facts to show that for all $A \subseteq \Sigma^\omega$, $\Sigma_{[}I]^\omega \setminus \big(\mathrm{L}(\mathcal{M}) \cap A\big)\downarrow_I = \big(\mathrm{L}(\mathcal{M}) \setminus A\big)\downarrow_I$ holds, which proves together with Equation 6 that Equation 4 is true:

$$\begin{aligned}
v \in \big(\mathrm{L}(\mathcal{M}) \setminus A\big)\downarrow_I &\iff \exists w \in \mathrm{L}(\mathcal{M}) \setminus A : (w\downarrow_I = v)\\
&\iff \exists w \in \mathrm{L}(\mathcal{M}) : (w\downarrow_I = v) \wedge w \notin A\\
&\underset{\text{Eq.7}}{\overset{\text{Eq.8}}{\iff}} \forall w \in \mathrm{L}(\mathcal{M}) : (w\downarrow_I = v) \rightarrow w \notin A\\
&\iff \forall w \in \mathrm{L}(\mathcal{M}) : w \in A \rightarrow (w\downarrow_I \neq v)\\
&\iff \forall w \in \mathrm{L}(\mathcal{M}) \cap A : (w\downarrow_I \neq v)\\
&\iff \nexists w \in \mathrm{L}(\mathcal{M}) \cap A : (w\downarrow_I = v) \iff v \notin \big(\mathrm{L}(\mathcal{M}) \cap A\big)\downarrow_I
\end{aligned}$$

Equation 5 is a simple extension of Equation 4 to the alphabet $\Sigma_{[}AP]$. It follows from the fact that for any language $\mathrm{L} \subseteq \Sigma_{[}I]^\omega : (\Sigma_I^\omega \setminus \mathrm{L})\uparrow_{AP} = \Sigma_{[}I]^\omega\uparrow_{AP} \setminus \mathrm{L}\uparrow_{AP}$ holds. □

With the help of Lemma 2 we can simplify Formula 3 to

$$\big((\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\neg\psi))\downarrow_I\uparrow_{AP} \cup \mathrm{L}(\mathcal{M})\big) \cap \mathrm{L}(\varphi) \tag{9}$$

This allows us to compute a repair using a synthesis procedure for the automaton $\big((\mathcal{M} \times \mathcal{A}_{\neg\psi})|_I + \mathcal{M}\big) \times \mathcal{A}_\varphi$, which is much simpler to construct.

Lemma 3 (Avoiding input projection) *Given a machine $\mathcal{M}$ and an LTL-formula φ, for every word $w \in \Sigma^\omega$, $w \in (\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\varphi))\downarrow_I\uparrow_{AP} \iff \mathcal{M}(w\downarrow_I) \in \mathrm{L}(\varphi)$ holds.*

Proof

$$\begin{aligned}
w \in (\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\varphi))\downarrow_I\uparrow_{AP} &\iff w\downarrow_I \in (\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\varphi))\downarrow_I\\
&\iff \exists w' \in \mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\varphi) : w'\downarrow_I = w\downarrow_I\\
&\iff \exists w' \in \mathrm{L}(\mathcal{M}) : w'\downarrow_I = w\downarrow_I \wedge w' \in \mathrm{L}(\varphi)\\
&\iff \mathcal{M}(w\downarrow_I) \in \mathrm{L}(\varphi)
\end{aligned}$$

□

Due to Lemma 3 we can check if a word produced by $\mathcal{M}'$ lies in $(\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\varphi))\downarrow_I\uparrow_{AP}$ by checking whether $\mathcal{M}$ treats the input projection of that word correctly. A synthesizer looking for a solution to Equation 9 can simulate $\mathcal{M}$ and check its output against $\neg\psi$ to decide whether $\mathcal{M}'$ is allowed to deviate from $\mathcal{M}$. This allows us to solve our repair problem using the simple setup we depict in Fig. 4. It shows five automata running in parallel:

(1) The original machine $\mathcal{M}$.
(2) The repair candidate $\mathcal{M}'$, a copy of $\mathcal{M}$ that includes multiple options to modify $\mathcal{M}$.
(3) A specification automaton A_{φ} to check if the new machine $\mathcal{M}'$ satisfies its objective.
(4) A specification automaton $A_{\neg\psi}$ to check if the original machine $\mathcal{M}$ violates ψ.
(5) A specification automaton A_{eq} that checks if the outputs of $\mathcal{M}$ and $\mathcal{M}'$ coincide, i.e., $eq = \mathsf{G}(\bigwedge_{o\in O} o \leftrightarrow o')$, where O is the set of outputs of $\mathcal{M}$ and o' is the copy of output $o \in O$ in machine $\mathcal{M}'$.

Theorem 3 *Given the setup depicted in Fig. 4, a repair option in $\mathcal{M}'$ is a valid repair according to Definition 3, if it satisfies the formula*

$$\varphi \wedge (\neg\psi \vee eq). \tag{10}$$

Proof Follows from Lemma 2 and Lemma 3. □

Formula 10 forces $\mathcal{M}'$ to (1) behave according to φ and (2) mimic the behavior of $\mathcal{M}$, if $\mathcal{M}$ satisfies ψ. Note that all automata can be constructed separately because they can be connected through the winning (or acceptance) condition. We avoid the monolithic construction of a specification automaton and obtain the same complexity as for classical repair. E.g., if φ, $\neg\psi$, and eq are represented by Büchi automata, then we can check for $\varphi \wedge (\neg\psi \vee eq)$ by first merging the acceptance states of $\neg\psi$ and eq, and then solving for a generalized Büchi condition, which is quadratic in the size of the state space $(|A_{\neg\psi}| \times |M| \times |M'| \times |A_{\varphi}| \times 2)$.

4.4 Implementation

Our prototype implementation is based on the following two ideas:

(1) If a synthesis problem can be decided by looking at a finite set of possible repairs[1] (combinations of choices), then the choice of repair can be encoded using multiple initial states.
(2) An initial state that does not lead to a counter example represents a correct repair. Any model checker can be adapted to return such an initial state, if one exists. By default a model checker returns the opposite, i.e., an initial state that leads to a counter-example but it is not difficult to change it. E.g., in BDD-based model-checkers some simple set operations suffice and in SAT-based checkers one can make use of unsat-core to eliminate failing initial states.

The main drawback of this approach is that the state space is multiplied by the number of considered repairs. However, the approach has several benefits which make it particularly interesting for program repair. First, it is easy to restrict the set of repairs to those that are simple and readable. In our prototype implementation we adapt the idea of *Syntax-Guided Synthesis* [1] and search for a repair within a given set of user-defined expressions. In the examples, we derive these expressions manually from the operators used in the program (see Sect. 8 for more details). Furthermore, we assume a given fault location that will be replaced by one of the user-defined expressions (cf. [24,27]). Expression generation and fault localization are interesting and active research directions (cf. Sect. 1) but are not addressed in this chapter. We focus on the problem of deciding what constitutes a good repair. The second main benefit is that we can adapt an arbitrary model checker to solve our repair problem. We believe (based on initial experiments) that at the current state, model checkers

[1] Note that any synthesis problem with memoryless winning strategies satisfies this condition.

are significantly more mature than synthesis frameworks. In our implementation we used a version of NuSMV [11] that we slightly modified to return an initial state that does not lead to a counter example.

Note that using the syntax-guided approach (i.e., using a set of expressions to choose from) is an implementation choice not a restriction of our approach. An implementation generating arbitrary expression as in [24,27] is possible.

5 Discussion about choosing in the lower bound ψ

We present several different choices for ψ and analyze their strengths and weaknesses:

(1) $\psi = \varphi$
(2) If $\varphi = f \rightarrow g$, then $\psi = f \wedge g$
(3) $\psi = \emptyset$

5.1 Exact

Choosing $\psi = \varphi$ is the most restrictive choice. It requires that $\mathcal{M}'$ behaves like $\mathcal{M}$ on all words that are correct in $\mathcal{M}$. While this is in general desirable, this choice can be too restrictive as Example 3 in Sect. 4 shows. One might think that the problem in Example 3 is that φ is a liveness specification. The following example shows that choosing $\psi = \varphi$ can also be too restrictive for safety specifications.

Example 5 Let $\mathcal{M}$ be a machine with input r and output g; $\mathcal{M}$ always outputs $\neg g$, i.e., $\mathcal{M}$ implements $\mathsf{G}(\neg g)$. Assume $\varphi = \mathsf{F}(\neg r) \rightarrow \mathsf{G}(g) = \mathsf{G}(r) \vee \mathsf{G}(g)$. Applying Formula 9, we obtain $(\mathsf{G}(\neg g) \wedge \neg(\mathsf{G}(r) \vee \mathsf{G}(g)))\downarrow_I\uparrow_{AP}$[2] $\wedge\, (\mathsf{G}(r) \vee \mathsf{G}(g)) = (\mathsf{F}(\neg r) \wedge \mathsf{G}(g)) \vee (\mathsf{G}(r) \wedge \mathsf{G}(\neg g))$. This formula is not realizable because a machine does not know if the environment will always send a request ($\mathsf{G}(r)$) or if the environment will eventually stop sending a request ($\mathsf{F}(\neg r)$). A correct machine has to respond differently in these two cases. So, $\mathcal{M}$ cannot be repaired if $\psi = \varphi$.

5.2 Assume-guarantee

It is very common that the specification is of the form $f \rightarrow g$ (as in the previous example). Usually, f is an assumption on the environment and g is the guarantee the machine has to satisfy if the environment meets the assumption. Since we are only interested in the behavior of $\mathcal{M}$ if the assumption is satisfied, it is reasonable to ask the repair to mimic only traces on which the assumption and the guarantee is satisfied, i.e., choosing $\psi = f \wedge g$.

Example 6 (Example 5 continued) Recall Example 5, we decompose φ into assumption $\mathsf{F}\,\neg r$ and guarantee $\mathsf{G}\,g$. Now, we can see that $\mathcal{M}$ is only correct on words on which the assumption is violated, so the repair should not be required to mimic the behavior of $\mathcal{M}$. If we set $\psi = \mathsf{F}\,\neg r \wedge \mathsf{G}\,g$, then $L(M) \cap L(\psi) = \emptyset$ and $\mathcal{M}'$ is unrestricted on all input traces.

5.3 Unrestricted

If we choose $\psi = \emptyset$ the repair is unrestricted and the approach coincides with the work presented in [24].

[2] LTL is not closed under projection. We use LTL only to describe the corresponding automata computations.

6 Reasons for repair failure

In the following we discuss why a repair attempt can fail. The first and simplest reason is that the specification is not realizable. In this case, there is no correct system implementing the specification and therefore also no repair. However, a machine can be unrepairable even with respect to a realizable specification. The existence of a repair is closely related to the question of realizability (Corollary 1). Rosner [38] identified two reasons for a specification φ to be unrealizable.

(1) *Input-completeness* if φ is not input-complete, then φ is not realizable. For instance, consider specification $\mathsf{G}(r)$ requiring that r is always true. If r is an input to the system, the system cannot choose the value of r and therefore also not guarantee satisfaction of φ.
(2) *Causality/clairvoyance* certain input-complete specifications can only be implemented by a clairvoyant system, i.e., a system that has knowledge about future inputs (a system that is non-causal). For instance, if the specification requires that the current output is equal to the next input, written as $\mathsf{G}(o \leftrightarrow \mathsf{X}\, i)$, then a correct system needs a look-ahead of size one to produce a correct output.

The following lemma shows that given an input-complete specification φ, input-completeness will not cause our repair algorithm to fail.

Lemma 4 (Input-completeness) *If φ is input-complete, then $\big((\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi))\!\downarrow_I \rightarrow \mathrm{L}(\mathcal{M})\big) \cap \mathrm{L}(\varphi)$ is input-complete.*

Proof Let $w_I \in \Sigma_{\mathrm{I}}{}^{\omega}$. If $w_I \in (\mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi))\!\downarrow_I$, then there is a word $w \in \mathrm{L}(\mathcal{M}) \cap \mathrm{L}(\psi)$ such that $w\!\downarrow_I = w_I$. Therefore we have found a word for w_I. If not, then a word for w_I exists because φ is input complete. □

A failure due to missing causality can be split into two cases: the case in which the repair needs finite look-ahead (see Example 7 below) and the case in which it needs infinite look-ahead (see Example 8 below). The examples show that even if the specification is realizable (meaning implementable by a causal system), the repair might not be implementable by a causal system.

Example 7 Consider the realizable specification $\varphi = g \vee \mathsf{X}\, r$ and a machine $\mathcal{M}$ that keeps g low all the time, i.e., $\mathcal{M}$ satisfies $\mathsf{G}(\neg g)$. If input r is high in the second step, $\mathcal{M}$ satisfies φ. An exact repair (according to Definition 2) needs to set g to low in the first step if the input in the second step is $high$, because it has to mimic $\mathcal{M}$ in this case. On the other hand, it the input in the second step is low, g needs to be set to high in the first step. So, any exact repair has to have a look-ahead of at least one, in order to react correctly.

The following example shows a faulty machine and a (realizable) specification for which a correct repair needs infinite look-ahead.

Example 8 Consider a machine $\mathcal{M}$ with input r and output g that copies the input to the output. Assume we search for a repair such that the modified machine satisfies the specification $\varphi = \mathsf{G}\,\mathsf{F}\, g$ requiring that g is high infinitely often. Machine $\mathcal{M}$ violates the specification on all input sequences that keep r low from some point onwards, i.e., on all words fulfilling $\mathsf{F}(\mathsf{G}\, r)$. Recall that a repair $\mathcal{M}'$ has to behave like $\mathcal{M}$ on all correct inputs. In this example, $\mathcal{M}'$ has to behave like $\mathcal{M}$ on all finite inputs, because it does not know whether or not the input word lies in $\mathsf{F}(\mathsf{G}\, r)$ without seeing the word completely, i.e., without infinite look-ahead.

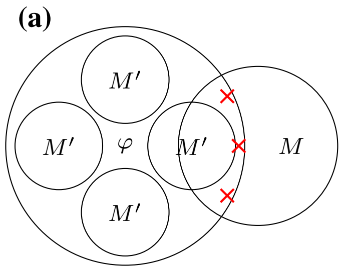

Fig. 5 Two reasons for unrepairability. **a** $\mathcal{M}$ includes bad traces, **b** $\mathcal{M}$ cuts two valid machines

Theorem 4 (Possibility of repair) *Assume that we cannot repair machine $\mathcal{M}$ with respect to a realizable specification φ. Then a repairing machine needs either finite or infinite look-ahead.*

Proof Follows from [38], Corollary 1, and Lemma 4. □

6.1 Characterization based on possible machines

Another way to look at a failed repair attempt is from the perspective of possible machines. Recall, in Fig. 3 we depict a correct repair $\mathcal{M}'$ as a circle covering the set of words in the intersection of $\mathcal{M}$ and ψ. In Fig. 5 we use the same graphical representations to explain two reasons for failure. Figure 5a depicts several machines $\mathcal{M}'$ realizing φ. A repair of $\mathcal{M}$ has to be one of the machines realizing φ. As observed in [22], there are words satisfying φ that cannot be produced by any correct machine (depicted as red crosses in Fig. 5a). E.g, recall the specification $\varphi = g \vee \mathsf{X}(r)$ in Example 7. The word in which g is low initially and r high in the second step satisfies φ but will not be produced by any correct (causal) machine because the machine cannot rely on the environment to raise r in the second step. If the machine we are aiming to repair includes such a trace, a repair attempt with $\psi = \varphi$ will fail. In this case, we can replace φ (or ψ) by the strongest formula that is open-equivalent[3] to φ in order to obtain a solvable repair problem. However, even if φ is replaced by its strongest open-equivalent formula, the repair attempt might fail for the reason depicted in Fig. 5b. We again depict several machines $\mathcal{M}'$ realizing φ. $\mathcal{M}$ shares traces with several of these machines, but no machine covers the whole intersection of φ and $\mathcal{M}$. In other words, an implementing machine would have to share the characteristics of two machines.

7 Alternative notions of "semantically close"

The first inclusion in Definition 3 strictly defines the set of traces of a machine $\mathcal{M}$ a repaired machine $\mathcal{M}'$ has to include. We can relax or strengthen this requirement using rewards. Rewards allows us to ask for the machine that *agrees most of the time* with $\mathcal{M}$ or for the machine that *agrees on the most traces* with $\mathcal{M}$. We will first present an example showing where alternative notions make sense.

[3] Two formulas φ and φ' are open-equivalent if any machine M implementing φ also implements φ' and vice-versa [22].

7.1 Letter-optimal solutions

One intuitive solution to the repair problem is a machine that "modifies the least number of output letters". If φ is a safety condition, then we search for a repair that stays within the safe region. In order to find a valid repair that minimized the number of different output letters between $\mathcal{M}$ and $\mathcal{M}'$, our task is twofold: (1) stay in the safe region and (2) minimize the number of times $\mathcal{M}'$ chooses an output that differs from the output of $\mathcal{M}$. Since we consider infinite traces, we average the difference between $\mathcal{M}$ and $\mathcal{M}'$ over the length of the trace. This can be achieved using a mean payoff objective, as the following example illustrates.

Example 9 Let $\varphi = \mathsf{G}(r \to g)$, and let $\mathcal{M}$ fulfill $\mathsf{G}((g \leftrightarrow \mathsf{X}\,\neg g))$, i.e., the machine signals g in every second step. Even if we choose ψ to be equal to φ, then a valid repair $\mathcal{M}'$ could, e.g., set g always high on every input trace for which $\mathcal{M}$ violates the specification, thus the output of $\mathcal{M}'$ would differs from that of $\mathcal{M}$ in every second step.

In order to guide $\mathcal{M}'$ to choose output values that are similar to the ones from $\mathcal{M}$, we can reward $\mathcal{M}'$ whenever it copies the behavior of $\mathcal{M}$. To calculate such a machine $\mathcal{M}'$, we augment the synthesis game for φ[4] and $\mathcal{M}$ (cf. Sect. 2) with rewards. Recall that Player 0 wins the synthesis game if she wins the safety game. The game is played in rounds. First player 1 picks an input from Σ_I, and then player 0 picks an output form Σ_O. We extend the game by granting a reward whenever the output chosen at a state is equal to the corresponding output of $\mathcal{M}$ at this state and ask Player 0 to maximize the average reward long each play. Recall that such an objectives is called mean-payoff objective.

In this example, an optimal strategy (i.e., the strategy that maximizes the grant) will give a grant in every second step and whenever r is signaled.

In the following we generalize this idea to liveness specifications by adding a parity condition to the mean-payoff game. For a discussion and implementation of two-player games with mean-payoff and parity objectives, see e.g., [2].

Definition 4 (*Letter-optimal repair*) Let $\mathcal{G} = ((\mathrm{S} \times \Sigma_{AP}), \mathrm{S}_0, \mathrm{S}_1, \mathrm{s}_0, \Delta)$ be a two player game with a parity objective $\lambda : \mathrm{S} \times \Sigma_{AP} \to \mathbb{N}$ such that (1) player 0 controls the output symbols, (2) player 1 controls the input symbols, and (3) the game is played in turns, i.e.,

(1) $\forall(\mathrm{s}, i \cup o) \in \mathrm{S}_0 : ((\mathrm{s}, i \cup o), (\mathrm{s}', i' \cup o')) \in \Delta \to i = i'$,
(2) $\forall(\mathrm{s}, i \cup o) \in \mathrm{S}_1 : ((\mathrm{s}, i \cup o), (\mathrm{s}', i' \cup o')) \in \Delta \to o = o'$,
(3) $\forall(\mathrm{s}, \mathrm{s}') \in \Delta \forall i \in 0, 1 : \mathrm{s} \in \mathrm{S}_i \to \mathrm{s}' \notin \mathrm{S}_i$.

Such a game is generated, for example, to synthesize an LTL-formula. Let further $\mathcal{M} = (M, \Sigma_\mathrm{I}, \Sigma_\mathrm{O}, m_0, \delta, \Omega)$ be a machine. Then we define the *Letter-Optimal parity game* $\mathcal{G}'$ by combining $\mathcal{G}$ with $\mathcal{M}$ as follows. Let $\mathcal{G}' = (M \times (\mathrm{S} \times \Sigma_{AP}), M \times \mathrm{S}_0, M \times \mathrm{S}_1, (m_0, \mathrm{s}_0), \Delta')$ be a two player game, where $((m, (\mathrm{s}, i \cup o)), (m', (\mathrm{s}', i' \cup o'))) \in \Delta'$ if and only if $((\mathrm{s}, i \cup o), (\mathrm{s}', i' \cup o')) \in \Delta$ and

$$s' = \begin{cases} m & \text{if } (b, i \cup o) \in \mathrm{S}_1 \\ \delta(m, i) & \text{if } (b, i \cup o) \in \mathrm{S}_0 \end{cases}$$

As reward function, we define $r((m, (\mathrm{s}, i \cup o))) = 1$ if $\Omega(m, i) = o$ and $(\mathrm{s}, i) \in \mathrm{S}_1$, and 0 otherwise. That is, a reward is assigned if the output of the machine and the output of the last player-0 transition agree. In addition, we define $\lambda' : M \times (\mathrm{S} \times \Sigma_{AP}) \to \mathbb{N}$ by $\lambda'(m, (\mathrm{s}, i \cup o)) = \lambda((\mathrm{s}, i \cup o))$, i.e., we copy the parity condition.

[4] Note that this extension can also be applied in addition to ψ, since our new repair notion is reducible to the classical synthesis problem (cf. Sect. 4).

A letter-optimal repair is a winning and mean-payoff optimal strategy for the defined game and vice versa. The parity condition ensures that the repair is qualitatively correct (e.g., fulfills an LTL-formula), while the quantitative (mean-payoff) condition ensures that as few letters as possible are modified. Note that such games might require infinite memory strategies in general, but ϵ-optimal finite memory strategies exist [2].

7.2 Trace-optimal solutions

Another intuitive solution to the repair problem is a machine that "modifies the least number of traces on average". We model the "on average" part using uniformly distributed inputs. This requires us to combine adversarial and probabilistic opponents because we need to assume that Player 1 is adversarial to ensure the correctness of the specification; on the other hand, to optimize the "on average" modifications, we assume a probabilistic Player 1. In the following we will focus on safety specifications because for those specifications one can take a unified view by translating the adversarial behavior into probabilistic behavior. For parity specification, a unified view is not possible: one has to keep them separately and find a strategy that satisfies the parity condition against an adversarial player and optimizes the mean-payoff condition against a probabilistic player. Note that one can compute such a strategy by adapting the algorithm for mean-payoff-parity game [14] to handle a probabilistic opponent.

Example 10 Consider the following formula.

$$\varphi = (i \wedge \mathsf{X}\, i \rightarrow ((o \wedge \mathsf{X}\, o) \vee (\neg o \wedge \mathsf{X} \neg o))) \wedge (i \wedge \mathsf{X} \neg i \rightarrow (\neg o \wedge \mathsf{X} \neg o))$$

as specification over input alphabet $\Sigma_\mathrm{I} = \{i\}$ and output alphabet $\Sigma_\mathrm{O} = \{o\}$. It requires on input $ii \ldots$ either output $oo \ldots$ or output $\neg o \neg o \ldots$. On input $i \neg i$ it requires output $\neg o \neg o$. Consider further a faulty machine $\mathcal{M}$ that writes o constantly. The machine is correct on input words starting with ii and words starting with $\neg i$. It is incorrect on all words starting with $i \neg i$.

Our goal is to minimize the number of traces that are modified. We therefore assign to each trace a payoff, either 1 or 0. An unchanged trace gets payoff 1, while a changed trace gets payoff 0. To "count" the number of changed traces in a set of traces, we take the average payoff of all traces in that set. If all traces are changed, then the payoff is 0; if none of the traces was changed, then the payoff is 1. If half of the traces have changed, then the payoff is 0.5.

The minimal number of traces a repaired system has to change is all traces that start with i, i.e., 50 % of all possible traces.

The following definition provides an MDP whose optimal strategy provides a machine that is correct for a safety specification and has a minimal number of changed traces.

Definition 5 (*Trace-optimal MDP*) Let $\mathcal{A} = (\mathrm{S}, \mathrm{s}_0, \Sigma, \Delta, \mathrm{F})$ be a realizable, deterministic safety automaton, and let $\mathcal{M} = (M, m_0, \Sigma_\mathrm{I}, \Sigma_\mathrm{O}, \delta, \Omega)$ be a machine not fulfilling $\mathcal{A}$.

We define the *Word-Optimal MDP* $\mathcal{M}$ as follows. Let $\mathcal{M} = ((M \cup \{\bot, \top\}) \times \mathrm{S} \times \Sigma_\mathrm{I}, (\top, \mathrm{s}_0, i_0), \Sigma_\mathrm{O}, \overline{A}, p)$ be an MDP, where i_0 is some letter in Σ_I, $\overline{A}(m, \mathrm{s}) = \{o \in \Sigma_\mathrm{O} \mid \exists\, \mathrm{s}' : (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta\}$, i.e., the set of outputs allowed by the safety automaton in state s for input i. Further, the probability function is defined as

$$p((m, \mathrm{s}, i), o)(m', \mathrm{s}', i') = \begin{cases} \frac{1}{|\Sigma_I|} & m = \top \wedge m' = m_0 \wedge \mathrm{s}' = \mathrm{s}_0 \\ \frac{1}{|\Sigma_I|} & m \neq \bot \wedge \Omega(m, i) = o \wedge \\ & m' = \delta(m, i) \wedge (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta \\ \frac{1}{|\Sigma_I|} & m \neq \bot \wedge \Omega(m, i) \neq o \wedge \\ & m' = \bot \wedge (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta \\ \frac{1}{|\Sigma_I|} & m = \bot \wedge m' = \bot \wedge (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta \\ 0 & \text{otherwise} \end{cases}$$

As reward function, we define $r((m, \mathrm{s}, i), o) = 1$ if $m \neq \bot$, and 0 otherwise.

The initial state (modeled by $\top$) defines the distribution of the first letter. Subsequently, the machine and the safety automaton move synchronously, depending on the random input letter. If the strategy for an MDP makes a choice that differs from the choice of $\mathcal{M}$, then the first component of the state of the MDP goes to $\bot$ immediately, signaling that we "left" the machine. On the other hand, if the strategy for an MDP never differs from the choice of $\mathcal{M}$, then the first component of the states of a trace will always be an element of M. Therefore, the reward we chose will provide an average payoff of 1 if the behavior of $\mathcal{M}$ is never left, and a payoff of 0 if the behavior differs at least once. In that sense, the reward function "counts" changed and unchanged traces. An optimal, i.e., maximizing strategy for $\mathcal{M}$ therefore changes the minimal number of traces.

8 Empirical results

In this section we first describe the repair we synthesized for the traffic light example from Sect. 3. Then, we summarize the results on a set of example we analyzed. All experiments were run on a 2.4GHz Intel(R) Core(TM)2 Duo laptop with 4 GB of RAM.

8.1 Traffic light example

In the traffic light example, we gave the synthesizer the option to choose from 2^{50} expressions (all possible logical expression over combinations of light colors and signal states). NuSMV returns the expression $(s_2 \wedge s_1 \wedge (l_2 \neq \text{RED})) \vee (\neg s_2 \wedge s_1 \wedge l_2 \neq \text{GREEN})$, which is equivalent to $s_1 \wedge ((s_2 \wedge l_2 \neq \text{RED}) \vee (\neg s_2 \wedge l_2 \neq \text{GREEN}))$ in 0.2 seconds. The repair forbids the first light from turning yellow if the second light is already green. This is not the repair we suggested in Sect. 3 because the synthesizer has freedom to choose between the expressions that satisfy the new notion. Our new approach avoids the obvious but undesired repair of leaving the first light red, irrespective of an arriving car. This is the solution NuSMV provides (within 0.16s) if we use the previous repair notion [24].

8.2 Experiment description

In order to empirically test the viability of our approach and to confirm our improved repair suggestions, we applied our approach to several examples. We will first describe the examples we considered, and then we will analyze the results.

```
binary_search(array, needle) {
  lower := 1
  upper := len(array);
  i := (upper − lower)/2;
  while( array[i] != needle
         && array[upper] > needle
         && array[lower] < needle) {
    if (array[i] < needle) {
      lower = i;
    } else {
      upper = i;
    }
    i = (upper − lower)/2;
    return i;
  }
}
```

Fig. 6 Faulty implementation of a binary search algorithm

8.2.1 Binary search

Figure 6 shows an implementation of a binary search algorithm that includes a common mistake related to the assignment of `lower`. The implementation loops forever if the array has even length and the target value (`needle`) is in the rightmost element. We can check the implementation against the following property to reveal the mistake.

$$\varphi = \texttt{sorted(array)} \wedge \texttt{needle} \in \texttt{array} \implies \mathsf{F}\,\texttt{array[i] = needle}.$$

In order to repair the implemenation we allow the synthesizer to replace the assignment to `lower` with `i`, `lower + 1`, `lower - 1`, `upper`, `i - 1`, or `i + 1`. Synthesizing with $\psi = \varphi$ will not allow us to find a solution to this example. In order to see that, consider the array as input symbol and the returned value as output symbol. We then demand that the synthesizer finds a solution such that

(1) The two implementations return the same result when the input was invalid
(2) The two implementations return the same result when the input was valid and the broken implementation returns the correct result

It makes more sense to demand that the two implementations return the same result if the input is valid and the original implementation returns the correct result (Sect. 5). In fact, given this specification, the synthesizer returns the correct result `lower + 1`.

8.2.2 PCI

This example models the PCI Bus protocol, and is taken from the NuSMV distribution. The arbiter has to give bus access to six elements, all of which can demand access at any time. The solution is to have three smaller two-input arbiters that decide for priority between a pair of elements each, and one three-input arbiter that takes the decision of the two-input arbiters as input. They can run either in a fixed-priority or in a round-robin mode. For each bus element, there is a specification demanding that the element eventually can access the bus, if it demands it.

We introduced a bug in the round-robin mode of the two-input arbiter, which gave access to a element 1 if element 2 requested it (a simple off-by-one error). This meant that, for example, the processor would never receive access to the bus, i.e., one of the specifications is violated. We freed the behavior of the offending round-robin scheduler, thus allowing the synthesizer a lot of choice.

Using the classical synthesis approach, we can guarantee access to the processor by always granting access to this processor. We have repaired the arbiter according to the violated property, but significantly restricted its behavior and introduced new bugs, that lead to violations for other properties. With our approach, the synthesizer finds an implementation that guarantees access for all bus elements, although the specification refers only to one of clients.

8.2.3 Read–write lock example

A read–write lock can be implemented using a semaphore and a lock. In read–write locks there can be arbitrarily many readers to some data-structure. However, if a thread wants to write to the data-structure, then it tries to acquire a write-lock. Once it tries to acquire a write-lock, an implemenation can stop granting access to new readers. It then waits until all readers have left the data-structure, grants the write-lock, and only starts granting read- or write-locks, once the write-lock is released.

Consider the implementation shown in Fig. 7. Our specification demands that if whatever happens in ... is bounded, then there is no deadlock. The implementation violates this specification. Consider a run in which two threads simultaneously try to acquire the write-lock. The system can grant one half of the locks to one thread, the rest of the locks to the other thread. Since no thread has all locks, none can proceed and none of them releases all locks. Therefore the system is in a deadlock.

We now add an additional mutex, because we suspect that we have to lock the writer locking function, but we do not know how and when exactly. Figure 8 shows the extended implementation. We leave the actual condition when to acquire and release the lock free. Our repair method will only activate the condition in function `write_lock`. If any other lock is added, the change modifies runs that were implementing the specification before. Only the traces where two or more threads try to acquire the writer-lock are affected. This is not the case for the original repair approach, which can enable locks everywhere.

8.2.4 Processor

Here we take a model processor from the VIS distribution. We introduce a bug that shows up when executing the `XOR` opcode, i.e., where the processor has to store the bit-wise xor of two words.

We have several different repair models for this example, varying in the number and structure of candidates. In the first model, we restrict repair to the faulty component. Here, classical synthesis and our new approach provide the same result.

In the other model, we have more freedom in repairs. In this case, classical synthesis will allow changes that are not relevant to the specification, thereby introducing new bugs. Our approach, on the other hand, forbids such repairs and finds the only repair not inducing new bugs.

```
struct rw_lock {semaphore sem(N_THREADS)};

write_lock(rw_lock) {
  for i from 1 to N_THREADS {
      sem−−;
    }
}

read_lock(rw_lock) {
  sem−−;
}

release_read_lock(rw_lock) {
  sem++;
}

release_write_lock(rw_lock) {
  for i from 1 to N THREADS {
      sem++;
    }
}

THREAD_i {
  while (∗) {
    if (∗) {
      read_lock(lock);
      ....;
      release_read_lock(lock);
    } else {
      write_lock(lock);
      ....;
      release_write_lock(lock);
    }
  }
}
```

Fig. 7 Faulty implementation of a read–write lock

8.3 Results

We report the results in Table 1; For each example, we report the number of choices for the synthesizer (Column #Repairs), the time and number of BDD variables to (1) verify the correctness of the repair that we obtain (Column Verification), (2) find a repair with our new approach (Column Repair), and (3) solve the classical repair problem (Column Classical Repair).

In order to synthesize a repair, we followed the approach described in Sect. 4.4 (Fig. 4), i.e., we manually added freedom to the model and wrote formula for $\neg\psi$ and equality checking. For all but one of the examples (Processor (1)), the previous approach synthesizes degenerated repairs, while our approach leads to a correct program repair.

AG ($\rightarrow$) is Example 5 from Sect. 5. It uses the original specification for ψ, i.e., $\psi = \mathsf{F}(\neg r) \rightarrow \mathsf{G}(g)$. We let the synthesizer choose between all possible Boolean combinations of g, r and a memory bit containing the previous value of g. Our approach fails to find a repair. AG (&) is Example 6 from Sect. 5 with $\psi = \mathsf{F}(\neg r) \wedge \mathsf{G}(g)$, using the same potential repairs. In this case, a valid repair is found. As in the Assume-Guarantee examples, we have two different choices for ψ in the Binary Search (BS) example. In the case that $\psi = sorted \rightarrow correct$, there is no repair available, while for $\psi = sorted \wedge correct$ we find the correct repair.

```
struct rw_lock {semaphore sem(N_THREADS)};
mutex m;

write_lock(rw_lock) {
  if (?) lock(m);
  for i from 1 to N_THREADS {
      sem−−;
    }
  if (?) unlock(m);
}

read_lock(rw_lock) {
  if (?) lock(m);
  sem−−;
  if (?) unlock(m);
}

release_read_lock(rw_lock) {
  if (?) lock(m);
  sem++;
  if (?) unlock(m);
}

release_write_lock(rw_lock) {
  if (?) lock(m);
  for i from 1 to N_THREADS {
      sem++;
    }
  if (?) unlock(m);
}
```

Fig. 8 Read–write lock implementation extended with mutex option

Table 1 Experimental results

	#Rep.	Verification		Repair		Class. repair	
		Time	#Vars	Time	#Vars	Time	#Vars
AG (→)	2^{12}	n/a	n/a	0.038	16	0.012	14
AG (&)	2^{12}	0.015	14	0.025	14	0.012	12
BS (→)	5	n/a	n/a	0.78	27	0.1	21
BS (&)	5	0.232	27	0.56	27	0.1	21
RW-lock	16	0.222	34	0.232	34	0.228	22
Traffic	2^{55}	0.183	68	0.8	68	0.155	63
PCI	27	0.3	56	0.8	56	0.5	53
Processor (1)	2	2 m 02 s	135	2 m 41 s	135	0.5	69
Processor (2)	4	4 m 28 s	138	5 m 07 s	138	0.5	69
Processor (3)	25	5 m 23 s	140	18 m 05 s	140	0.5	71

The RW-Lock example demonstrates that our approach can also be used to synthesize locks. The synthesizer can choose between 16 options (which represent release/acquire actions of different locks at different locations). Our notion of repair forbids the acquisition of other locks (the repair we obtained in 27 ms with the approach in [24]), because this would imply for example that two threads asking for a read-lock at the same time have to wait for each other. Our notion of repair encodes that runs that were unobstructed before remain unobstructed in the new implementation as well, as long as they do not lead to a dead-lock. Our experiments show that our notion of repair urges the synthesizer to find the intended solution by forcing it to leave correct program runs unchanged. We therefore believe that our approach makes synthesis as a development methodology more practical.

The Processor examples demonstrate what happens in complex models when increasing the amount of freedom in a model. They also show how repairing partial specifications may lead to the introduction of new bugs. In Processor (1), the minimal amount of non-determinism is introduced, i.e., only as much freedom as strictly necessary to repair. Here, the classical approach and our new approach give the same result. In Processor (2), we introduce more freedom, which leads to incorrect repairs with the classical approach. In particular, the fault is in the ALU of the processor, and the degenerated repairs incorrectly execute the `AND` instruction, which is handled correctly in the original model. We allow replacing the faulty and the a correct instruction by either a `XOR, AND, OR, SUB` or `ADD` instruction. Finally, Processor (3) shows that the time necessary for synthesis grows sub-linearly with the number of repair options.

On average, synthesizing a repair takes 2.3 times more time than checking its correctness. Our new approach seems to be one order of magnitude slower than the classical approach. This is expected because finding degenerated repairs is usually much simpler. (This is comparable to finding trivial counter examples.) In order to find correct repairs with the approach of [24], we would need to increase the size of the specification, which will significantly slow down the approach.

We also rerun the processor example using a commercial tool with an option to focus on sequential equivalence checking. We observed an 20 % performance increase with the option enabled, which is a preliminary confirmation of your hypothesis that tools specialized in solving the sequential equivalence checking problems are well suited for this task.

Finally, during our experiments we observed that we can speed-up the computation be requiring only a subset of the outputs to follow the fault machine $\mathcal{M}$, if $\mathcal{M}$ satisfies the lower bound ψ. E.g., only outputs that are in the cone-of-influence of the repair location need to be compared.

9 Future work and conclusions

9.1 Future work

Investigation of ways to increase the computational power of a repaired machine seems interesting. Every machine $\mathcal{M}'$ repairing $\mathcal{M}$ has to behave like $\mathcal{M}$ until it concludes that $\mathcal{M}$ does not respond to the remaining input word correctly. As shown in Example 7, $\mathcal{M}'$ might not know early enough if $\mathcal{M}$ will fail or succeed. Therefore, studying repairs with finite look-ahead is an interesting direction. To extend the applicability of our approach to infinite state programs, one could explore suitable program abstraction techniques (cf. [46]). We are also planing to investigate if simulation distances [10] can be used to find a semantically close repair. Finally, we are planing to continue our experiments with model checkers specialized in solving the sequential equivalence checking problem [28,29]. We believe that such solvers perform well on our problem, because $\mathcal{M}'$ and $\mathcal{M}$ have many similar structures.

9.2 Conclusion

When fixing programs, we usually fix bugs one by one; at the same time, we try to leave as many parts of the program unchanged as possible. In this chapter, we introduced a new notion of program repair that supports this method. The approach allows an automatic program repair tool to focus on the task at hand instead of having to look at the entire specification. It also facilitates finding repairs for programs with incomplete specifications, as they often show up in real word programs.

References

1. Alur R, Bodík R, Juniwal G, Martin MMK, Raghothaman M, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A (2013) Syntax-guided synthesis. Paper presented at the conference on FMCAD, IEEE, New York, pp 1–17, Oct 2013
2. Bohy A, Bruyère V, Filiot E, Raskin J-F (2013) Synthesis from LTL specifications with mean-payoff objectives. In: Piterman N, Smolka SA (eds) TACAS, vol 7795. Springer, Heidelberg, pp 169–184
3. Buccafurri F, Eiter T, Gottlob G, Leone N (1999) Enhancing model checking in verification by ai techniques. Artif Intell 112(1–2):57–104
4. Bonakdarpour B, Ebnenasir A, Kulkarni SS (2009) Complexity results in revising unity programs. TAAS 4(1):5
5. Bonakdarpour B, Hajisheykhi R, Kulkarni SS (2014) Knowledge-based automated repair of authentication protocols. In: Jones CB, Pihlajasaari P, Sun J (eds) 19th International symposium on FM: lecture notes in computer science. vol 8442. Springer, pp 132–147, 2014
6. Bodík R, Jobstmann B (2013) Algorithmic program synthesis: introduction. STTT 15(5–6):397–411
7. Bonakdarpour B, Kulkarni SS, Abujarad F (2012) Symbolic synthesis of masking fault-tolerant distributed programs. Distrib Comput 25(1):83–108
8. Büchi JR, Landweber LH (1969) Solving sequential conditions by finite-state strategies. Trans Am Math Soc 138:295–311
9. Ball T, Naik M, Rajamani SK (2003) From symptom to cause: localizing errors in counterexample traces. In: Aiken A, Morrisett G (eds) In: Conference on POPL, ACM, New Orleans, pp 97–105, Jully 2003
10. Černý P, Henzinger TA, Radhakrishna A (2010) Simulation distances. In: Proceedings of the 21st international conference on concurrency theory, CONCUR'10. Springer, Berlin, pp 253–268
11. Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV Version 2: an opensource tool for symbolic model checking. In: 14th International conference on CAV, Springer, New York, July 2002
12. Clarke EM, Emerson EA (1981) Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen D (ed) Logic of programs, vol 131. Springer, New York, pp 52–71
13. Clarke EM, Grumberg O, McMillan KL, Zhao X (1995) Efficient generation of counterexamples and witnesses in symbolic model checking. In proceedings of the 32nd annual ACM/IEEE Design Automation Conference, ACM, New York, pp 427–432, Jan 1995
14. Chatterjee K, Henzinger TA, Jurdzinski M (2005) Mean-payoff parity games. In proceedings of the 20th annual IEEE symposium on LICS, IEEE Computer Society, New York, pp 178–187, June 2005
15. Church A (1963) In: Clarke E, Emerson EA, Sistla AP (eds) Proceedings of the 5th international congress of mathematicians, pp 23–25, Aug 1962
16. Chang K-H, Markov IL, Bertacco V (2008) Fixing design errors with counterexamples and resynthesis. IEEE Trans CAD Integr Circuits Syst 27(1):184–188
17. Chandra S, Torlak E, Barman S, ZBodik R (2011) Angelic debugging. In: 33rd International conference on software engineering (ICSE), ACM, New York, pp 121–130, May 2011
18. Drusinsky D, Harel D (1994) On the power of bounded concurrency i: finite automata. J ACM 41(3):517–539
19. Ebnenasir A, Kulkarni SS, Bonakdarpour B (2005) Revising unity programs: possibilities and limitations. In: OPODIS. Springer, Heidelberg, pp 275–290, 2005
20. Edelkamp S, Lluch-Lafuente A, Leue S (2001) Trail-directed model checking. Electron Notes Theor Comput Sci 55(3):343–356
21. Griesmayer A, Bloem R, Cook B (2006) Repair of boolean programs with an application to c. In: Ball T, Jones R (eds) CAV, volume 4144 of LNCS, Springer, Heidelberg, pp 358–371, 2006

22. Greimel K, Bloem R, Jobstmann B, Vardi M (2008) Open implication. In: ICALP, LNCS 5126, Springer, Heidelberg, pp 361–372, 2008
23. Groce A, Visser W (2003) What went wrong: explaining counterexamples. In: Ball T, Rajamani SK (eds) SPIN. Springer, New York, pp 121–135
24. Jobstmann B, Griesmayer A, Bloem R (2005) Program repair as a game. In: Ball T, Jones RB (eds) CAV. Springer, Heidelberg, pp 226–238
25. Janjua MU, Mycroft A (2006) Automatic correction to safety violations in programs. Thread Verification (TV'06). Unpublished
26. Jin HS, Ravi K, Somenzi F (2004) Fate and free will in error traces. STTT 6(2):102–116
27. Jobstmann B, Staber S, Griesmayer A, Bloem R (2012) Finding and fixing faults. J Comput Syst Sci 78(2):441–460
28. Khasidashvili Z, Moondanos J, Kaiss D, Hanna Z (2001) An enhanced cut-points algorithm in formal equivalence verification. In: Sixth IEEE international proceedings of the HLDVT, pp 171–176, 2001
29. Kaiss D, Skaba M, Hanna Z, Khasidashvili Z (2007) Industrial strength sat-based alignability algorithm for hardware equivalence verification. In: Proceedings of the FMCAD, IEEE, pp 20–26, 2007
30. Lichtenstein O, Pnueli A (1985) Checking that finite state concurrent programs satisfy their linear specification. In: Proceedings of the POPL, ACM, New York, pp 97–107, Jan 1985
31. Piterman N (2006) From nondeterministic buchi and streett automata to deterministic parity automata. In: 21st annual IEEE symposium on LICS, IEEE, pp 255–264, 2006
32. Piterman N (2007) From nondeterministic büchi and streett automata to deterministic parity automata. Log Methods Comput Sci 3(3):1–21
33. Pnueli A (1977) The temporal logic of programs. In: IEEE Symposium on FOCS, pp 46–57, 1977
34. Pnueli A, Rosner R (1989) On the synthesis of a reactive module. In: proceedings of the 16th ACM Symposium on POPL, ACM, pp 179–190, Jan 1989
35. Puterman ML (1994) Markov decision processes: discrete stochastic dynamic programming. Wiley, New York
36. Queille J-P, Sifakis J (1982) Specification and verification of concurrent systems in CESAR. In: 5th international symposium on programming, pp 337–351, 1982
37. Rabin MO (1969) Decidability of second-order theories and automata on infinite trees. Trans Am Math Soc 141:1–35
38. Rosner R (1997) Modular synthesis of reactive systems. PhD thesis, Stanford University
39. Renieris M, Reiss SP (2003) Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE international conference on ASE, IEEE, pp 30–39, 2003
40. Ravi K, Somenzi F (2004) Minimal assignments for bounded model checking. In: Proceedings of TACAS, Springer, Heidelberg, pp 31–45, Apr 2004
41. Safra S (1988) On the complexity of omega-automata. In: Proceedings of the FOCS, IEEE, pp 319–327, Oct 1988
42. Schewe S (2009) Tighter bounds for the determinisation of büchi automata. In: 12th international conference on FOSSACS, Springer, New York, pp 167–181, 2009
43. Samanta R, Deshmukh JV, Emerson EA (2008) Automatic generation of local repairs for boolean programs. In: FMCAD, Springer, Heidelberg, pp 1–10, 2008
44. von Essen C, Jobstmann B (2013) Program repair without regret. In: Proceedings of CAV, Springer, Berlin, Heidelberg, pp 896–911, 2013
45. Vechev M, Yahav E, Yorsh G (2009) Inferring synchronization under limited observability. In: TACAS'09, vol. 5505 of LNCS, Springer, New York, pp 139–154, 2009
46. Vechev MT, Yahav E, Yorsh G (2010) Abstraction-guided synthesis of synchronization. In: Proceedings of POPL, pp 327–338, 2010
47. Wolper P, Vardi MY, Sistla AP (1983) Reasoning about infinite computation paths (extended abstract). In: Proceedings of the FOCS, Tucson, pp 185–194, 1983
48. Zeller A, Hildebrandt R (2002) Simplifying and isolating failure-inducing input. IEEE Trans Softw Eng 28(2):183–200