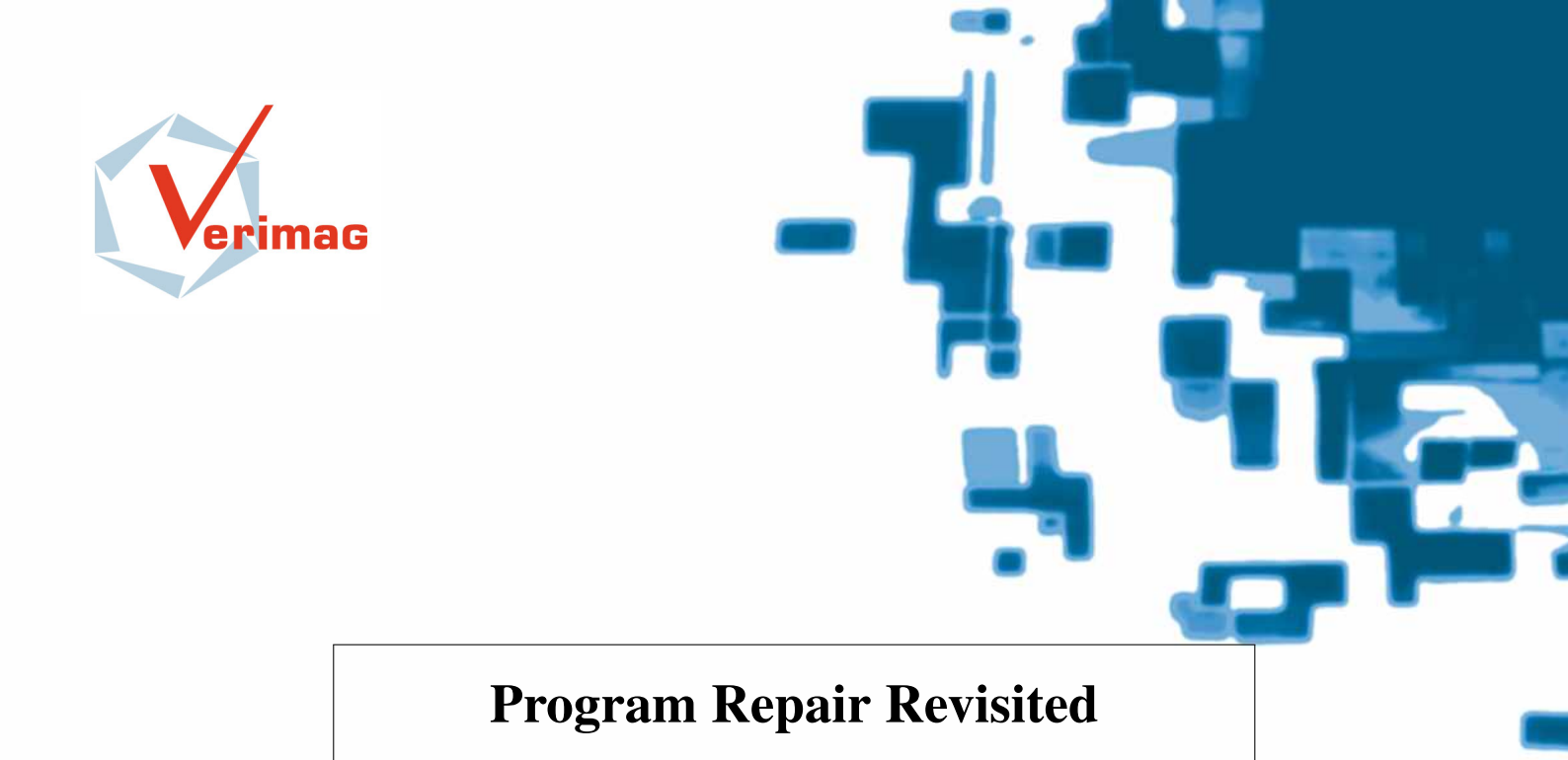


Program Repair Revisited

Christian von Essen and Barbara Jobstmann

Verimag Research Report n^o TR-2012-4

April 7, 2012

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Program Repair Revisited

Christian von Essen and Barbara Jobstmann

April 7, 2012

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 require that a repaired program satisfies the specification and is syntactically close to the faulty program. In addition our approach also allows the user to ask for a program that is semantically close by enforcing that a specific subset of the correct traces is preserved. Our approach is based on synthesizing a program producing a set of traces that stays 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.

Keywords:

Reviewers:

How to cite this report:

```
@techreport {TR-2012-4,
  title = {Program Repair Revisited},
  author = {Christian von Essen and Barbara Jobstmann},
  institution = {{Verimag} Research Report},
  number = {TR-2012-4},
  year = {}
}
```

1 Introduction

Debugging a program is often a difficult and tedious task: a programmer has to find the bug, localized the cause, and repair it. Model checking [8,23] has been successfully used to expose bugs in a program. There are several interesting approach [7,11,25,31,16,14,1,26] to explain the possible cause of an error. We are interested in addressing the last debugging step by automatically providing a bug-fix. Automatic program repair takes a program and a specification and searches for a correct program that satisfies the specification and is syntactically close to the original program (cf. [2,17,10,13,15,5,28,29,4]).

We are inspired by the work of Jobstmann et al. [17,18] and Chandra et al. [4]. Jobstmann et al. suggest to repair a program based on a game derived from the program and a formal specification given in Linear-Temporal Logic (LTL) [20]. The authors allow the repair procedure to replace an arbitrary component (e.g., an assignment or an if-condition) and require that the repaired program satisfies the formal specification. This approach has the benefit that no specific fault model needs to be defined and that expressive repair statements can be found. However, the automatic repair tool is free to change the original program substantially, especially when given too much freedom. Chandra et al. follow a different approach, they search in the space of all edits to a program for one edit that repairs failing tests without breaking any passing tests. We believe that the idea of not breaking correct behaviors is essential for automatically repairing programs.

We show how to generalize this idea to reactive programs with specifications and present the first repair approach that constructs repairs that are also semantically close to the original program. The key benefits of our approach are: (i) One can adjust how close the original program and the repair should be. (ii) The approach is less sensitive than previous approaches to the set of allowed program modifications. E.g., the approach in [17] constructs degenerated programs if given too much freedom in modifying the program. Our approach is less likely to construct degenerated programs because it preserves correct behaviors and their properties. (iii) Finally, since it preserves the core behavior of the program, the need for a complete specification is reduced.

Contributions. We present an example motivating the need for a new definition of program repair (Section 3). We define a new notion of repair for reactive programs and present an algorithm to compute such repairs (Section 4). The algorithm is based on synthesizing repairs with respect to a lower and an upper bound on the set of generated traces. We show the limitations of any repair approach that is based on preserving part of the program's behavior (Section 5).

2 Preliminaries

Words, languages, and projections. 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., $\Sigma_{\mathrm{AP}} = 2^{\mathrm{AP}}$. 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 $\mathrm{w} \in \Sigma^\omega$, we denote the letter at position i by w_i, where w_0 is the first letter. We use $\mathrm{w}_{..i}$ to denote the prefix of w up to position i, and $\mathrm{w}_{i..}$ to denote the suffix of w starting at position i. Given word $\mathrm{w} \in \Sigma_{\mathrm{AP}}^\omega$ and a set of propositions $\mathrm{I} \subseteq \mathrm{AP}$, we define the I-*projection* of w, denoted by $\mathrm{w}_{|\mathrm{I}}$, as $\mathrm{w}_{|\mathrm{I}} = l_0 l_1 \dots$ with $l_i = (\mathrm{w}_i \cap \mathrm{I})$ for all $i \geq 0$. Given a language $\mathrm{L} \subseteq \Sigma_{\mathrm{AP}}^\omega$ and a set $\mathrm{I} \subseteq \mathrm{AP}$, we define the I-*projection* of L, denoted by $\mathrm{L}_{|\mathrm{I}}$, as the set of words with the same I-projection, i.e., $\mathrm{L}_{|\mathrm{I}} = \{\mathrm{w} \in \Sigma_{\mathrm{AP}}^\omega \mid \exists\, \mathrm{w}' \in \mathrm{L} : \mathrm{w}_{|\mathrm{I}} = \mathrm{w}'_{|\mathrm{I}}\}$. Note that $\mathrm{w}_{|I} \in L_{|I}$ for every word $\mathrm{w} \in L$. A language $\mathrm{L} \subseteq \Sigma_{\mathrm{AP}}^\omega$ is called I-*deterministic* for some set $\mathrm{I} \subseteq \mathrm{AP}$ if for each word $v \in \Sigma_{\mathrm{I}}^\omega$ there is at most one word $w \in \mathrm{L}$ such that $w_{|\mathrm{I}} = v$. 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 $w_{|I} = v$.

Machines, automata, and formulas. A *finite state machine* is a tuple $\mathrm{M} = (\mathrm{Q}, \Sigma_{\mathrm{I}}, \Sigma_{\mathrm{O}}, \mathrm{q}_0, \delta, \gamma)$, where Q is a finite set of *states*, $\Sigma_{\mathrm{I}} \subseteq \Sigma$ and $\Sigma_{\mathrm{O}} \subseteq \Sigma$ are the *input* and the *output alphabet*, respectively, with $\Sigma_{\mathrm{I}} \cap \Sigma_{\mathrm{O}} = \emptyset$ and $\Sigma_{\mathrm{I}} \cup \Sigma_{\mathrm{O}} = \Sigma$, $\mathrm{q}_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma_{\mathrm{I}} \to Q$ is the *transition function*, and $\gamma : Q \times \Sigma_{\mathrm{I}} \to \Sigma_{\mathrm{O}}$ is the *output function*. A *run* ρ of M on an input word $\mathrm{w} \in {\Sigma_{\mathrm{I}}}^\omega$ is the sequence of states that the machine visits while reading the input word, i.e., $\rho = q_0 q_1 \dots \in \mathrm{Q}^\omega$ such that $\delta(q_i, w_i) = q_{i+1}$ for all $i \geq 0$. The *output word* M produces on w (denoted by $\mathrm{M}_{\mathrm{O}}(w)$) is the sequence of output letters that

the machine produces while reading the input word, i.e., for the run $q_0 q_1 \dots$ of M on w, the output word is $\mathrm{M_O}(w) = l_0 l_1 \dots \in \Sigma_\mathrm{O}^\omega$ with $l_i = \gamma(q_i, \mathrm{w}_i)$ for all $i \geq 0$. We denote by $\mathrm{L(M)}$ the *language* of M, i.e., the set of combined input/output words $\mathrm{L(M)} = \{(i_0 \cup o_0)(i_1 \cup o_1) \dots \in \Sigma_\mathrm{AP}^\omega \mid i \geq 0, i_j \in {\Sigma_\mathrm{I}}^\omega, o_j = \mathrm{M_O}(i)\}$.

A *Büchi automaton* is a tuple $\mathrm{A} = (\mathrm{S}, \Sigma, \mathrm{s}_0, \Delta, \mathrm{F})$ 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 A on a word $\mathrm{w} \in \Sigma^\omega$ is a sequence of states $\mathrm{s}_0\, \mathrm{s}_1\, \mathrm{s}_2 \dots \in \mathrm{S}^\omega$ such that $(\mathrm{s}_i, \mathrm{w}_i, \mathrm{s}_{i+1}) \in \Delta$ for all $i \geq 0$. A word is accepted by A if there exists a run $\mathrm{s}_0\, \mathrm{s}_1 \dots$ such that $\mathrm{s}_i \in \mathrm{F}$ for infinitely many i. We denote by $\mathrm{L(A)}$ the language of the Büchi automaton, i.e., the set of words accepted by A. A language that is accepted by a Büchi automaton is called ω-regular.

We use Linear Temporal Logic (LTL) [20] over a set of atomic proposition 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 $p \in \mathrm{AP}$. The semantics of an LTL formula φ is given with respect to words $\mathrm{w} \in \Sigma_\mathrm{AP}^\omega$ using the satisfaction relation $\models$. As usual, we define it inductively over the structure of the formula as follows: (i) $\mathrm{w} \models p$ iff $p \in \mathrm{w}_0$, (ii) $\mathrm{w} \models \neg\varphi$ iff $\mathrm{w} \not\models \varphi$, (iii) $\mathrm{w} \models \varphi_1 \wedge \varphi_2$ iff $\mathrm{w} \models \varphi_1$ and $\mathrm{w} \models \varphi_2$, (iv) $\mathrm{w} \models \mathsf{X}\varphi$ iff $\mathrm{w}_{1..} \models \varphi$, and (v) $\mathrm{w} \models \varphi_1\, \mathsf{U}\, \varphi_2$ iff $\exists i \geq 0 : \mathrm{w}_{i..} \models \varphi_2$ and $\forall j, 0 \leq j < i : \mathrm{w}_{j..} \models \varphi_1$. The Boolean operators $\vee$, $\rightarrow$, 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 $p \in \mathrm{w}_i$ for some $i \geq 0$ satisfies $\mathsf{F}\,p$. Dually, every word with $p \notin \mathrm{w}_i$ for all $i \geq 0$ satisfies $\mathsf{G} \neg p$. The language of φ, denoted $\mathrm{L}(\varphi)$, is the set of words satisfying formula φ. For every LTL formula φ one can construct a Büchi automaton A such that $\mathrm{L(A)} = \mathrm{L}(\varphi)$ [30,19]. Given a machine M and an LTL-formula φ, we say that M *satisfies* (or *implements*) φ, denoted by $\mathrm{M} \models \varphi$, if $\mathrm{L(M)} \subseteq \mathrm{L}(\varphi)$.

The following lemma follows directly from the definitions. We will use it in Section 4.

Lemma 1 (Machine languages). *The language of a machine is input deterministic and input complete.*

Synthesis problem. The synthesis problem [6] asks to construct a system that satisfies a given formal specification.

Definition 1 (Realizability and Synthesis). *Given an LTL formula (or an ω-regular language) φ over the atomic propositions* AP *partitioned into input and output propositions, i.e.,* $\mathrm{AP} = I \cup O$*, we say that φ is* realizable *if there exists a finite state machine M with input alphabet Σ_I and output alphabet Σ_O such that $M \models \varphi$.*

Theorem 1 (Synthesis Algorithms [3,24,21]). *There exists a deterministic algorithm that checks weather a given LTL-formula (or an ω-regular language) φ is* realizable*, i.e., there exists a machine* M *such that* $\mathrm{M} \models \varphi$*. If φ is realizable, then the algorithm constructs* M*.*

3 Example

In this section we give a simple example to motivate our definitions and highlight the differences to previous approaches such as [17].

Example 1 (Traffic Light). Assume we want to develop a sensor-driven traffic light system for a crossing of two streets. The system has two sets of lights (called `light1` and `light2`) and two sensors (called `sensor1` and `sensor2`), one set of lights and one sensor for each street entering the crossing. 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. As starting point we are given the implementation shown in Figure 1. 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 $\varphi = \mathsf{G}(\texttt{light1} \neq \texttt{GREEN} \vee \texttt{light1} \neq \texttt{GREEN})$. The current implementation clearly does not satisfy this requirement: if both sensors signal a car initially, then the lights will simultaneously move from red to yellow and to then to green, thus violating the specification.

```
1  typedef enum {RED, YELLOW, GREEN} traffic_light;
2  module Traffic (clock, sensor1, sensor2, light1, light2);
3    input clock, sensor1, sensor2;
4    output light1, light2;
5    traffic_light reg light1, light2;
6    initial begin
7      light1 = RED;
8      light2 = RED;
9    end
10   always @(posedge clock) begin
11     case (light1)
12       RED: if (sensor1) // Repair : if(sensor1 & !(light2 == RED & sensor2))
13                 light1 = YELLOW;
14       YELLOW: light1 = GREEN;
15       GREEN: light1 = RED;
16     endcase // case (light1)
17     case (light2)
18       RED: if (sensor2)
19                   light2 = YELLOW;
20       YELLOW: light2 = GREEN;
21       GREEN: light2 = RED;
22     endcase // case (light1)
23   end // always (@posedge clock)
24 endmodule // traffic
```

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

Let us try to repair the given implementation. Following the approach in [17] 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 Figure 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 because replacing Line 12 with `if (false)` ensures that the modified program satisfies specification φ. While this suggested repair is correct, it is very unlikely to please the programmer because it repairs "too much": it modifies the behavior of the system also on input traces on which the initial implementation was correct. We believe it is more desirable to follow the idea of Chandra et al. [4] saying that a repair is only allowed to change the behavior on incorrect execution. In this 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 that follows the intuition that a repair can only change the behavior of incorrect executions. Then, we provide an algorithm to compute such repairs and show repairs our new approach yields for the traffic light example.

4.1 Definitions

Given a machine M and a specification φ, we say a machine M' is an exact repair of M if (i) M' behaves like M on all traces satisfying φ and (ii) if M' fulfills φ. This leads to the following definition.

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

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

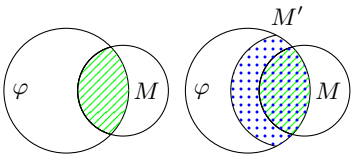


Fig. 2. Graphical representation of Def. 2

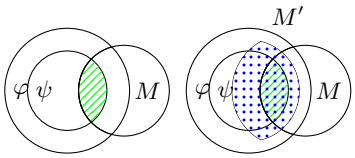


Fig. 3. Graphical representation of Def. 3

Note that the first inclusion defines the behavior of M' on all input words to which M responses correctly according to φ. Figure 2 illustrates Definition 2: the two circles depict $\mathrm{L}(\mathrm{M})$ 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.

Example 2 (Traffic Light, cont.). Following our definition the repair suggested in Example 1 (i.e., to replace `if (sensor1)` by `if (false)`) is not a valid repair anymore. The reason is that 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. In fact the initial implementation behaves correctly on any input trace on which `sensor1` and `sensor2` are never high simultaneously. So, any correct repair should include this input/output traces. An exact repair according to Definition 2 replaces `if (sensor2)` 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 too restrictive and makes repairing impossible in other cases, as the following example shows.

Example 3 (Definition 2 is too restrictive). Assume a machine M with one input r and one output g; M always copies r to g, i.e., M fulfills $\mathsf{G}(r \leftrightarrow g)$. The specification requires that g is eventually high, i.e., $\varphi = \mathsf{F}\, g$. We cannot repair M to satisfy $\mathsf{F}\, g$. The reason is that Definition 2 requires the repair M' to behave like M on all correct input traces. Therefore, as long as M still has a chance to satisfy φ, M' has to mimic the behavior of M. M is correct on all input traces that have a least one request, i.e., on traces satisfying $\mathsf{F}\, r$. So, M has always a chance to satisfy φ. Therefore, M' has to behave like M on every input and thus violates φ.

In order to find a repair, we have to relax the requirement that M' has to behave like M on all correct traces. Since there are many possible choices for the set of traces M' has to mimic, we leave the specific restriction free in the following definition.

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

$$\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi) \subseteq \mathrm{L}(\mathrm{M}') \subseteq \mathrm{L}(\varphi) \tag{2}$$

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

4.2 Algorithm

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* M, M′ *be two machines. Machine* M′ *satisfies Formula 2* $(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi) \overset{(a)}{\subseteq} \mathrm{L}(\mathrm{M}') \overset{(b)}{\subseteq} \mathrm{L}(\varphi))$ *if and only if* M′ *satisfies the following formula:*

$$\mathrm{L}(\mathrm{M}') \subseteq \underbrace{\big((\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi))_{|I} \rightarrow \mathrm{L}(\mathrm{M})\big)}_{(i)} \cap \underbrace{\mathrm{L}(\varphi)}_{(ii)} \tag{3}$$

For two languages A and B, $A \rightarrow B$ is an abbreviation for $(\Sigma^\omega \setminus A) \cup B$. Intuitively, Formula 3 requires that (i) M′ *behaves like* M *on all input words that* M *answers conforming to ψ and (ii)* M *fulfills specification φ.*

Proof. From left to right: We have to show that (i) and (ii) include $\mathrm{L}(\mathrm{M}')$. Inclusion in (ii) follows trivially from (b). It remains to show $\mathrm{L}(\mathrm{M}') \subseteq \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)\big)_{|I} \rightarrow \mathrm{L}(\mathrm{M})$. Let $\mathrm{w} \in \mathrm{L}(\mathrm{M}')$. If $\mathrm{w} \notin \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)\big)_{|I}$, then the implication follows trivially. Otherwise we have to show that $\mathrm{w} \in \mathrm{L}(\mathrm{M})$. From $\mathrm{w} \in \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)\big)_{|I}$ it follows that there is a word $\mathrm{w}' \in \mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)$ (and therefore $\mathrm{w}' \in \mathrm{L}(\mathrm{M})$) with $\mathrm{w}_{|I} = \mathrm{w}'_{|I}$. From this and from (a) $\mathrm{w}' \in \mathrm{L}(\mathrm{M}')$ follows. Since $\mathrm{L}(\mathrm{M}')$ is input deterministic and since $\mathrm{w} \in \mathrm{L}(\mathrm{M}')$, we have that $\mathrm{w}' = \mathrm{w}$. Therefore, $\mathrm{w} \in \mathrm{L}(\mathrm{M})$.

From right to left: We have to show (a) and (b) Again, (b) follows trivially from $\mathrm{L}(\mathrm{M}') \subseteq (ii)$. It remains to show (a), i.e., that $\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi) \subseteq \mathrm{L}(\mathrm{M}')$. Let $\mathrm{w} \in \mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)$. Let further $\mathrm{w}' \in \mathrm{L}(\mathrm{M}')$ such that $\mathrm{w}_{|I} = \mathrm{w}'_{|I}$. Such a word must exist because M′ is input complete. We now show $\mathrm{w} = \mathrm{w}'$, and therefore $\mathrm{w} \in \mathrm{L}(\mathrm{M}')$.

Since $\mathrm{w} \in \mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)$ and since w' and w have the same input projection, we know that $\mathrm{w}' \in \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)\big)_{|I}$. From $\mathrm{L}(\mathrm{M}') \subseteq (i)$ and from $\mathrm{w}' \in \mathrm{L}(\mathrm{M}')$, it follows that $\mathrm{w}' \in \mathrm{L}(\mathrm{M})$. Since w and w' have the same input projection, $\mathrm{w} \in \mathrm{L}(\mathrm{M})$, and $\mathrm{L}(\mathrm{M})$ is input deterministic, it follows that $\mathrm{w} = \mathrm{w}'$.

This theorem leads together with [21] to the following corollary, which allows us to use classical synthesis algorithm to compute repairs.

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

$$\big((\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi))_{|I} \rightarrow \mathrm{L}(\mathrm{M})\big) \cap \mathrm{L}(\varphi) \tag{4}$$

is realizable.

4.3 Implementation Details

Corollary 1 gives an algorithm to construct repairs based on synthesis techniques (cf. [17]). In order to compute the language defined by Formula 4, we can use standard automata-theoretic operations. More precisely, we construct a Büchi automaton A_φ recognizing φ and a Büchi automaton A_ψ recognizing ψ. Note that M is a Büchi automaton, in which all states are accepting. Since Büchi automata are closed under conjunction, disjunction, projection, and negation, we can construct an automaton for $\big(\overline{(\mathrm{M} \times \mathrm{A}_\psi)_{|I}} + \mathrm{M}\big) \times \mathrm{A}_\varphi$, where by $A \times B$ denotes the conjunction, $A + B$ denotes the disjunction of automata A and B, $\bar{A}$ denotes the negation, 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 4, we can use Theorem 1 to synthesize a repair.

Since negating on non-deterministic Büchi automata induces an exponential blow-up in the worst case [9], we show in Lemma 2 how to avoid the negation. Using Lemma 2 we can replace Formula 4 by

$$\big((\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\neg\psi))_{|I} \cup \mathrm{L}(\mathrm{M})\big) \cap \mathrm{L}(\varphi) \tag{5}$$

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

Lemma 2. *For any machine* M *and any LTL-formula φ, the following equality holds:*

$$\Sigma^\omega \setminus \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi)\big)_{|I} = \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\neg\varphi)\big)_{|I} \tag{6}$$

Proof. We first prove the following two equalities:

(i) $\mathrm{L(M)} \cap \big(\Sigma^\omega \setminus \mathrm{L}(\varphi)\big) = \mathrm{L(M)} \setminus \mathrm{L}(\varphi)$
(ii) For each $A \subseteq \Sigma^\omega$ we have $\big(\mathrm{L(M)} \setminus A\big)_{|I} = \neg\big(\mathrm{L(M)} \cap A\big)_{|I}$

For (i): $\mathrm{L(M)} \cap \big(\Sigma^\omega \setminus \mathrm{L}(\varphi)\big) = \{\mathrm{w} \in \Sigma^\omega \mid \mathrm{w} \in \mathrm{L(M)} \wedge \mathrm{w} \in \Sigma^\omega \setminus \mathrm{L(M)}\} =$
$\{\mathrm{w} \in \Sigma^\omega \mid \mathrm{w} \in \mathrm{L(M)} \wedge \mathrm{L} \notin \mathrm{L}(\varphi)\} = \{\mathrm{w} \in \mathrm{L(M)} \mid \mathrm{w} \notin \mathrm{L}(\varphi)\} = \mathrm{L(M)} \setminus \mathrm{L}(\varphi)$
For (ii): $\mathrm{w} \in \big(\mathrm{L(M)} \setminus A\big)_{|I} \iff \exists\, \mathrm{w}' \in \mathrm{L(M)} \setminus A : \mathrm{w}'_{|I} = \mathrm{w}_{|I} \iff \exists\, \mathrm{w}' \in \mathrm{L(M)} : \mathrm{w}'_{|I} = \mathrm{w}_{|I} \wedge \mathrm{w}' \notin A \overset{*}{\iff} \forall\, \mathrm{w}' \in \mathrm{L(M)} : \mathrm{w}'_{|I} = \mathrm{w}_{|I} \Rightarrow \mathrm{w}' \notin A \iff \forall\, \mathrm{w}' \in \mathrm{L(M)} : \mathrm{w} \in A \Rightarrow \mathrm{w}'_{|I} \neq \mathrm{w}_{|I} \iff \forall\, \mathrm{w}' \in \mathrm{L(M)} \cap A : \mathrm{w}'_{|I} \neq \mathrm{w}_{|I} \iff \nexists\, \mathrm{w} \in \mathrm{L(M)} \cap A : \mathrm{w}'_{|I} = \mathrm{w}_{|I} \iff \mathrm{w} \notin \big(\mathrm{L(M)} \cap A\big)_{|I}$
Note that the equivalence marked with $(*)$ holds because $\mathrm{L(M)}$ is input deterministic. Using these two equalities we prove the desired lemma as follows: $\big(\mathrm{L(M)} \cap \mathrm{L}(\neg\varphi)\big)_{|I} = \big(\mathrm{L(M)} \cap \Sigma^\omega \setminus \mathrm{L}(\varphi)\big)_{|I} \overset{\text{(i)}}{=} \big(\mathrm{L(M)} \setminus \mathrm{L}(\varphi)\big)_{|I} \overset{\text{(ii)}}{=} \neg\big(\mathrm{L(M)} \cap \mathrm{L}(\varphi)\big)_{|I}$

5 Discussion

In this section we discuss choices for ψ and analyze why a repair can fail.

5.1 Choices for ψ

We present different choices for ψ and analyze their strengths and weaknesses: (1) $\psi = \varphi$, (2) if $\varphi = f \to g$, then $\psi = f \wedge g$, and (3) $\psi = \emptyset$.

Exact. Choosing $\psi = \varphi$ is the most restrictive choice. It requires that M' behaves like M on all words that are correct in M. While this is in general desirable, this choice can be too restrictive as Example 3 in Section 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 4. Let M be a machine that has one input r and one output g; M always outputs $\neg g$, i.e., M fulfills $\mathsf{G}(\neg g)$. Assume $\varphi = \mathsf{F}(\neg r) \to \mathsf{G}(g) = \mathsf{G}(r) \vee \mathsf{G}(g)$. Applying Formula 5, we obtain $(\mathsf{G}(\neg g) \wedge \neg(\mathsf{G}(r) \vee \mathsf{G}(g)))|_I$[1] $\wedge$ $(\mathsf{G}(r) \vee \mathsf{G}(g)) = (\mathsf{F}(\neg r) \wedge \mathsf{G}(g)) \vee (\mathsf{G}(r) \wedge \mathsf{G}(\neg g))$. 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, M cannot be repaired if $\psi = \varphi$.

Assume-Guarantee. It is very common that the specification is of the form $f \to g$ (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 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 5 (Example 4 continued). Recall Example 4, we decompose φ into assumption $\mathsf{F}\,\neg r$ and guarantee $\mathsf{G}\,g$. Now, we can see that M is only correct on words on which the assumption is violated, so the repair should not be required to mimic the behavior of M. If we set $\psi = \mathsf{F}\,\neg r \wedge \mathsf{G}\,g$, then $L(M) \cap L(\psi) = \emptyset$ and M' is unrestricted on all input traces.

Unrestricted. If we choose $\psi = \emptyset$ the repair is unrestricted and the approach coincides with the work presented in [17]. In the appendix, we discuss another choice for ψ that guarantees that a repair can be found if there exists a repair with $\psi = \emptyset$. The notion is based on automata-based game theory.

We also show in the appendix how to relax the requirement on a repair using a probabilistic satisfaction. The idea is that instead of asking M' to mimic all traces in $\mathrm{L}(\psi) \cap L(\mathrm{M})$, we ask M' to maximize the number of traces that mimic M by maximizing the probability to produce the same traces as M.

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

5.2 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 [27] 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 3 (Input-completeness). *If φ is input-complete, then $\big((\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi))_{|I} \rightarrow \mathrm{L}(\mathrm{M})\big) \cap \mathrm{L}(\varphi)$ is input-complete.*

Proof. Let $\mathrm{w}_I \in {\Sigma_\mathrm{I}}^\omega$. If $\mathrm{w}_I \in (\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi))_{|I}$, then there is a word $\mathrm{w} \in \mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)$ such that $\mathrm{w}_{|I} = \mathrm{w}_I$. 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 6 below) and the case in which it needs infinite look-ahead (see Example 7 below). The example show that even if the specification is realizable (meaning implementable by a causal system), the repair might not by implementable by a causal system.

Example 6. Consider the realizable specification $\varphi = g \vee \mathsf{X}\, r$ and a machine M that keeps g low all the time, i.e., M satisfies $\mathsf{G}(\neg g)$. If input r is high in the second step, 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 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.

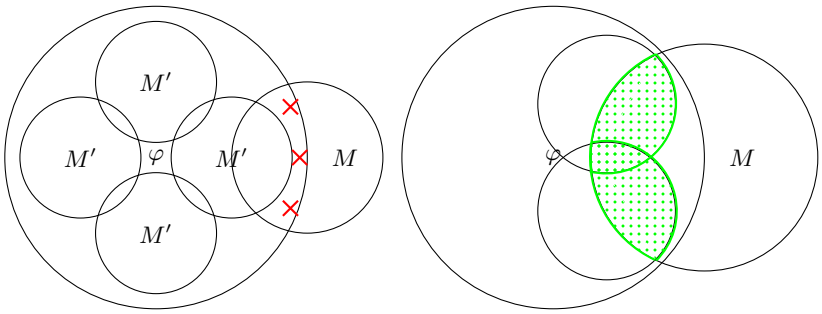
The following example shows a faulty machine and a (realizable) specification for which a correct repair needs infinite look-ahead.

Example 7. Consider a machine 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 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 M' has to behave like M on all correct inputs. In this example, M' has to behave like 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.

Theorem 3 (Possibility of repair). *Assume that we cannot repair machine M with respect to a realizable specification φ. Then, a repairing machine needs either finite or infinite look-ahead.*

Proof. Follows from [27], Corollary 1, and Lemma 3.

Characterization based on possible machines. Another way to look at a failed repair attempt is from the perspective of possible machines. Recall, in Figure 3 we depict a correct repair M' as a circle covering the set of words in the intersection of M and ψ. In Figure 4 we use the same graphical representations to explain two reasons for failure. Figure 4(a) depicts several machines M' realizing φ. A repair of M has to be



(a) M includes bad traces (b) M cuts several valid machines

Fig. 4. Two reasons for unrepairability

one of the machines realizing φ. As observed in [12], there are words satisfying φ that cannot be produced by any correct machine (depicted as red crosses in Figure 4(a)). E.g, recall the specification $\varphi = g \vee \mathsf{X}(r)$ in Example 6. 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[2] 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 depict in Figure 4(b). We again depicts several machines M' realizing φ. M shares traces with several of these machines, but no machine covers the whole intersection of φ and M. In other words, an implementing machine would have to share the characteristics of two machines.

6 Future Work and Conclusions

Future Work. We will follow two orthogonal directions to make it possible to repair more machines. The first one increases the computational power of a repair machine. Every machine M' repairing M has to behave like M until it concludes that M does not respond to the remaining input word correctly. As shown in Example 6, M' might not know early enough if M will fail or succeed. Therefore, studying repairs with finite look-ahead is an interesting direction. The second direction studies relaxed notion of set-inclusion in order to express how "close" two machines are. It is possible to measure the distance between a repair M' and a machine M quantitatively, e.g,. by measuring how often they differ on average, or when the last point in time is where a different output occurs. For instance, this allows to implement a "supverisor" that occasionally overwrites the output of M' to fulfill a liveness specification.

Conclusion. In this paper, we presented a new theoretical notion of repair that guarantees that a subset of the previously correct executions stays unchanged. We showed several ways to choose the traces that should stay unchanged by a repair. We also analyzed the limits of this repair definition in general. We proposed an algorithm to automatically (i) decide if a repair is possible and (ii) find a repair if one exists.

We believe that our framework will prove useful in practice because it implicitly includes liveness conditions. This makes writing specifications of programs easier and also allows to encode liveness specifications into the code using classic assertions. Furthermore, our notion is close to how repair is done in practice. This implies that the limits we identified in this paper are also relevant to how repair is done in practice. In particular it shows that the current program repair practice cannot be extended to liveness conditions in general.

[2] Two formulas φ and φ' are open-equivalent any machine M implementing φ also implements φ' and vice versa [12].

References

1. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL 2003*, pages 97–105, Jan. 2003. 1
2. F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by ai techniques. *Artif. Intell.*, 112(1-2):57–104, 1999. 1
3. J. R. Büchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969. 1
4. S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *ICSE 2011*, pages 121–130, New York, NY, USA, 2011. ACM. 1, 1
5. K.-H. Chang, I. L. Markov, and V. Bertacco. Fixing design errors with counterexamples and resynthesis. *IEEE Trans. on CAD*, 27(1):184–188, 2008. 1
6. A. Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, 1963. 2
7. E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *DAC*, 1995. 1
8. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, 1981. 1
9. D. Drusinsky and D. Harel. On the power of bounded concurrency i: Finite automata. *J. ACM*, 41(3):517–539, 1994. 4.3
10. A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising unity programs: Possibilities and limitations. In *OPODIS*, volume 3974 of *LNCS*, 2005. 1
11. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Trail-directed model checking. *ENTCS*, 5(3), Aug. 2001. Software Model Checking Workshop 2001. 1
12. K. Greimel, R. Bloem, B. Jobstmann, and M. Vardi. Open implication. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 361–372, 2008. LNCS 5126. 5.2, 2
13. A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to c. In *CAV*, volume 4144 of *LNCS*, pages 358–371. Springer, 2006. 1
14. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *10th International SPIN Workshop*, pages 121–135. Springer-Verlag, 2003. LNCS 2648. 1
15. M. U. Janjua and A. Mycroft. Automatic correction to safety violations in programs. Thread Verification (TV'06), 2006. Unpublished. 1
16. H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *TACAS'02*, pages 445–459, Grenoble, France, Apr. 2002. LNCS 2280. 1
17. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 226–238. Springer, 2005. 1, 3, 1, 4.3, 5.1
18. B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *Journal of Computer and System Sciences (JCSS)*, 78(2):441–460, 2012. 1
19. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL*, pages 97–107, 1985. 2
20. A. Pnueli. The temporal logic of programs. In *FOCS*. IEEE Comp.Soc., 1977. 1, 2
21. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989. 1, 4.2
22. M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, Apr. 1994. 6
23. J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, pages 337–351, 1982. 1
24. M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969. 1
25. K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *TACAS'04*, pages 31–45, Barcelona, Spain, Mar.-Apr. 2004. LNCS 2988. 1
26. M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ICASE*, pages 30–39, Montreal, Canada, Oct. 2003. 1
27. R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Stanford University, 1997. 5.2, 5.2
28. R. Samanta, J. V. Deshmukh, and E. A. Emerson. Automatic generation of local repairs for boolean programs. In A. Cimatti and R. B. Jones, editors, *FMCAD*, pages 1–10, 2008. 1
29. M. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. In *TACAS'09*, volume 5505 of *LNCS*, pages 139–154. Springer, 2009. 1
30. P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths (extended abstract). In *FOCS*, pages 185–194. IEEE, 1983. 2
31. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002. 1

Appendix

Alternative Choices for ψ

In the following we assume that the reader is familar with automata-based game theory.

Winning Region. Another sensible choice is the set of traces that will not force us into a situation where we can lose even if we modify the output. In other words, M' is able to stay in a position in which it can still win the game it plays against the environment to fulfill φ.

Example 8. 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 M writing o constantly as machine we need to repair. The machine is correct on input words starting with ii and words starting with $\neg i$. It is incorrect on all words starting with $i\neg i$. If we choose $\psi = \varphi$, then a repairing machine M' constructed from M has to behave like M on input words starting with ii, i.e., has to write oo. It follows that M' has to begin its output with o if the input begins with i. Therefore, M' has to start its answer to $i\neg i$ with o, violating φ.

The problem for M' stems from having to output o on input i. If the environment now chooses to generate $\neg i$, then M' loses. In other words, when writing o as answer to i, M' leaves its *winning region*. If ψ recognizes all input-output-words such that M stays in the winning region of φ, then this allows M' to answer $\neg o\neg o$ on input ii, therefore it can fulfill φ. On input $\neg i\ldots$, it still behaves like M.

Staying in the winning region is not always enough to allow a repair, as the following example shows.

Example 9. Let $\varphi = \mathsf{F}\, g$ and M copy its input to its output.

In this case, the input-output-words that allow M' to stay in the winning region are all words: at any point, a request signal might come, so M' still has to behave like M on all traces.

The last example is a liveness specification. Choosing ψ as the winning region allows us to always repair a machine with respect to a safety-specification, as the following theorem shows. This is of practical interest, because assertions inserted into a program always correspond to a safety-specification.

Theorem 4. *Let φ be a realizable safety specification and let ψ be the set of words that allow to stay in the winning region of a game constructed from φ. Then M can be repaired w.r.t. φ.*

Proof. Let S be a safety-automaton recognizing the set words input-output words that stay in the winning region of a game constructed from ψ. We now have that an input-output words fulfills φ if and only if it stays inside S, because φ is a safety specification.

Then $\big((\mathrm{L}(S) \cap \mathrm{L}(\mathrm{M}))_{|I} \rightarrow \mathrm{L}(\mathrm{M})\big) \cap \mathrm{L}(\varphi)$ is realizable by M' defined as follows. As states we choose the combined state space of $S \times \mathrm{M} \times \varphi$. M' behaves like M until M suggests to move outside of S. In that case, M' starts to play a strategy to win φ. If an input trace is such that its combined input-output word of M is in S, then the combined input-output word is also in φ; therefore the output of M' is correct. Otherwise the combined input-output word lies outside of $\mathrm{L}(S) \cap \mathrm{L}(\mathrm{M})$, and M' is therefore free to choose its output independently from M. The output changes once M suggests a move that would lead outside of S. Because M' chooses the new output while we are still in S, i.e., still in the winning region of φ, the combined input-output word lies in φ.

Probabilistic ψ

Markov Chains and Markov Decision Processes. The first inclusion in Definition 3 strictly defines the set of traces of a machine M a repaired machine M' has to include. We can relax this requirement using probabilities. Using Markov Decision Processes allows us to ask for the machine that *agrees most of the time* with M and for the machine that *agrees on the most traces* with M. We will first define Markov

Decision Processes. For extensive explanation and analysis of Markov Decision Processes, we refer the reader to [22]. We then continue with an example showing where relaxing ψ with probabilities makes sense. Afterwords we define the Markov Decision Processes necessary to gain the desired optimal repaired machines.

Let $\mathcal{D}(S) := \{p : S \to [0,1] \mid \sum_{s \in S} p(s) = 1\}$ be the *set of probability distributions* over a finite set S. A *Markov decision process (MDP)* is a tuple $\Lambda = (L, L_0, \Omega, \tilde{\Omega}, p)$, where L is a finite set of *states*, $L_0 \subseteq L \in \mathcal{D}(L)$ is the *initial distribution*, Ω is a finite set of *actions*, $\tilde{\Omega} : L \to 2^{\Omega}$ is the *enabled action function* defining for each state l the set of enabled actions in l, and $p : L \times \Omega \to \mathcal{D}(L)$ is a probabilistic *transition function*. For technical convenience we assume that every state has at least one enabled action.

Sample Runs, Strategies, and Rewards. A *(sample) run* ρ of Λ is an infinite sequence of tuples $(l_0, a_0)(l_1, a_1) \cdots \in (L \times \Omega)^{\omega}$ of states and actions such that for all $i \geq 0$, (i) $a_i \in \tilde{\Omega}(l_i)$ and (ii) $p(l_i, a_i)(l_{i+1}) > 0$. We write Ω for the set of all runs, and Ω_l for the set of runs starting at state l.

A *strategy (or policy)* is a function $d : (L \times \Omega)^* L \to \mathcal{D}(\Omega)$ that assigns a probability distribution to all finite sequences in $(L \times \Omega)^* L$. A strategy must refer only to enabled actions, i.e., for all sequences $\mathrm{w} \in (L \times \Omega)^*$, states $l \in L$, and actions $a \in \Omega$, if $d(\mathrm{w}\, l)(a) > 0$, then action a has to be enabled in l, i.e., $a \in \tilde{\Omega}(l)$. An MDP together with a state l and a strategy d defines a probability space $\mathcal{P}^d_{\Lambda,l}$ that uniquely defines the probability of every measurable set of runs starting in l.

Given a *reward function* $r : L \times \Omega \to \mathbb{R}$, we call the expected value $\mathbb{E}_{\Lambda,d}[r]$ of r under Λ and d the *average reward*.

Given an MDP Λ strategy d is called *optimal for reward* r if $\mathbb{E}_{\Lambda,d}[r] = \min_{d'} \mathbb{E}_{\Lambda,d}[r]$, where d' ranges over all possible strategies.

Letter-Optimal Solutions. One intuitive solution to the repair problem is a machine that "modifies the least possible output letters on average", where we assume that input is uniformly distributed. If φ is a safety condition, then we are certain that we can win by staying in the winning region. Therefore, our task is twofold: (1) stay in the winning region and (2) minimize the number of times M' chooses an output that differs from the output of M on average. This can be achieved using Markov Decision Processes, as the following example illustrates.

Example 10 (Probabilistic ψ). Let $\varphi = \mathsf{G}(r \to g)$, and let M fulfill $\mathsf{G}((g \leftrightarrow \mathsf{X} \neg g))$, i.e., the machine signals g in every second step. If we choose ψ to recognize the winning region of φ, then a repairing machine can output $\mathsf{G}(g)$, once M' violates its specification, thus choosing an output that differs from that of M infinitely often.

Another option is to "reward" M' whenever it copies the behavior of M. To calculate such a machine M', we build an MDP from the winning region of φ and M. In each state, the non-deterministic choice allows to choose one element from Σ_{O}. The transition occurs then after a uniform probabilistic choice over Σ_{I} happens. The MDP grants a reward whenever the output chosen at a state is equal to the output M chooses at this state.

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. The following definition provides such an MDP with rewards.

Definition 4 (Letter-Optimal MDP). *Let* $\mathrm{A} = (\mathrm{S}, \Sigma, \mathrm{s}_0, \Delta, \mathrm{F})$ *be a realizable, deterministic safety automaton, and let* $\mathrm{M} = (\mathrm{Q}, \Sigma_{\mathrm{I}}, \Sigma_{\mathrm{O}}, \mathrm{q}_0, \delta, \gamma)$ *be a machine not fulfilling an LTL-formula* φ.

Then we define the Letter-Optimal MDP Λ *as follows. Let* $\Lambda = (\mathrm{Q} \times \mathrm{S} \times \Sigma_{\mathrm{I}}, L_0, \Sigma_{\mathrm{O}}, \tilde{\Omega}, p)$ *be an MDP, where* $L_0(\mathrm{q}_0, \mathrm{s}_0, i) = 1/|\,\Sigma_{\mathrm{I}}\,|$ *for all* $i \in \Sigma_{\mathrm{I}}\,|$ *and 0 for all other states,* $\tilde{\Omega}(\mathrm{q}, \mathrm{s}) = \{o \in \Sigma_{\mathrm{O}} \mid \exists\, \mathrm{s}' : (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta\}$*, i.e., the set of outputs allowed by the safety automaton in state* s *for input* i*. Further, the probability function is defined as*

$$p((\mathrm{q}, \mathrm{s}, i), o)(\mathrm{q}', \mathrm{s}', i') = \begin{cases} \frac{1}{|\,\Sigma_{\mathrm{I}}\,|} & \delta(\mathrm{q}, i) = \mathrm{q}' \wedge (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta \\ 0 & \textit{otherwise} \end{cases}$$

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

We have to show that the $p(\mathrm{q}, \mathrm{s})$ is a probability distribution over $\mathrm{Q} \times \mathrm{S} \times \Sigma_{\mathrm{I}}$. To that end, note that q' is defined by $\delta(\mathrm{q}, i)$ and that s' is defined by $(\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta$ because A is deterministic.

Trace-Optimal Solutions. Another intuitive solution to the repair problem is a machine that "modifies the least traces on average", where we again assume that input is uniformly distributed.

Example 11. We assume the same setup as in Example 8, i.e.,

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

and the machine writes o constantly.

Our goal is to minimize the number of traces that receive a change. 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 no traces are 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 (Word-Optimal MDP). *Let* $\mathrm{A} = (\mathrm{S}, \Sigma, \mathrm{s}_0, \Delta, \mathrm{F})$ *be a realizable, deterministic safety automaton, and let* $\mathrm{M} = (\mathrm{Q}, \Sigma_\mathrm{I}, \Sigma_\mathrm{O}, \mathrm{q}_0, \delta, \gamma)$ *be a machine not fulfilling an LTL-formula* φ*.*

Then we define the Word-Optimal MDP Λ *as follows. Let* $\Lambda = ((\mathrm{Q} \cup \{\bot\} \times \mathrm{S} \times \Sigma_\mathrm{I}, \{(\mathrm{q}_0, \mathrm{s}_0, i) \mid i \in \Sigma_\mathrm{I}\}, \Sigma_\mathrm{O}, \tilde{\Omega}, p)$ *be an MDP, where* $L_0(\mathrm{q}_0, \mathrm{s}_0, i) = 1/|\,\Sigma_\mathrm{I}\,|$ *for all* $i \in \Sigma_\mathrm{I}\,|$ *and 0 for all other states,* $\tilde{\Omega}(\mathrm{q}, \mathrm{s}) = \{o \in \Sigma_\mathrm{O} \mid \exists\, \mathrm{s}' : (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta\}$*, i.e., the set of outputs allowed by the safety automaton in state* s *for input* i*. Further, the probability function is defined as*

$$p((\mathrm{q}, \mathrm{s}, i), o)(\mathrm{q}', \mathrm{s}', i') = \begin{cases} \frac{1}{|\,\Sigma_\mathrm{I}\,|} & \mathrm{q} \neq \bot \wedge \gamma(\mathrm{q}, i) = o \wedge \\ & \mathrm{q}' = \delta(\mathrm{q}, i) \wedge (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta \\ \frac{1}{|\,\Sigma_\mathrm{I}\,|} & \mathrm{q} \neq \bot \wedge \gamma(\mathrm{q}, i) \neq o \wedge \\ & \mathrm{q}' = \bot \wedge (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta \\ \frac{1}{|\,\Sigma_\mathrm{I}\,|} & \mathrm{q} = \bot \wedge \mathrm{q}' = \bot \wedge (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta \\ 0 & \textit{otherwise} \end{cases}$$

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

If the strategy for an MDP makes a choice that differs from the choice of M, then the first component of the state of the MDP goes to $\bot$ immediatly, signalling that we "left" the machine. On the other hand, if the strategy for an MDP never differs from the choice of M, then the first component of the states of a trace will always be an element of Q. Therefore, the reward we chose will provide an average payoff of 1 if the behavior of 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 Λ therefore changes the minimal number of traces.

In this section we provide more examples and solutions that our repair mechanism provides.

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 datastructure. But 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 releases.

Consider the following implementation attempt.

```
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_THREAD {
      sem++;
    }
}

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

Our specification demands that if whatever happens in ... is bounded, then there is no deadlock. The implementation fails this specification. Consider a run in which 2 threads simultaneously try to acquire the write-lock. The system can no grant one half of the locks to one thread, the rest of the locks to the other thread. Because neither thread has all locks, none can proceed. Because none can proceed. Because none can proceed, no thread is releasing a lock. Therefore we are in a deadlock.

We now add an additional mutex, because we suspect that we somehow have to lock the writer locking function, but we don't know how and when exactly. Therefore we modify the implementation to look as follows.

```
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_THREAD {
      sem++;
    }
  if (?) unlock(m);
}
```

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 fulfilling the specification before. Only the traces where two or more threads try to acquire the writer-lock are affected.