Program Repair without Regret

Christian von Essen[1] and Barbara Jobstmann[1,2,3]

[1] Verimag, CNRS and Universities of Grenoble, France
[2] Jasper Design Automation, CA
[3] École Polytechnique Fédérale de Lausanne, Switzerland

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 have evaluated the approach on several examples.

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 [9, 26] has been successfully used to expose bugs in a program. There are several approaches [1, 8, 12, 15, 17, 28, 29, 37] 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 correct program that satisfies the specification and is syntactically close to the original program (cf. [2, 4, 5, 11, 14, 16, 18, 31, 35]). 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. [4] 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.

We present the first repair approach that constructs repairs that are guaranteed to satisfy the specification and that are not only syntactically, but also semantically close to the original program. 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.

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). Finally, we present experimental results based on a prototype employing the NuSMV [7] model checker.

2 Preliminaries

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., $\Sigma_{\mathrm{AP}} = 2^{\mathrm{AP}}$. 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 a set of propositions $\mathrm{I} \subseteq \mathrm{AP}$, we define the I-*restriction* of a word $\mathrm{w} \in \Sigma_{\mathrm{AP}}^\omega$, denoted by $\mathrm{w}{\downarrow_\mathrm{I}}$, as $\mathrm{w}{\downarrow_\mathrm{I}} = l_0 l_1 \cdots \in \Sigma_{\mathrm{I}}^\omega$ 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-*restriction* of L, denoted by $\mathrm{L}{\downarrow_\mathrm{I}}$, as the set of I-restrictions of all the words in L, i.e., $\mathrm{L}{\downarrow_\mathrm{I}} = \{\mathrm{w}{\downarrow_\mathrm{I}} \mid \mathrm{w} \in \mathrm{L}\}$. Given a word $w \in \Sigma_{\mathrm{I}}^\omega$ over a set of propositions $\mathrm{I} \subseteq \mathrm{AP}$, we use $w{\uparrow_{AP}}$ to denote the *extension* of w to the alphabet Σ_{AP}, i.e., $w{\uparrow_{AP}} = \{w' \in \Sigma_{\mathrm{AP}}^\omega \mid w'{\downarrow_I} = w\}$. 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 $\mathrm{L} \subseteq \Sigma_{\mathrm{AP}}^\omega$ 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 $w{\downarrow_\mathrm{I}} = v$. 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 $w{\downarrow_I} = v$.

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*, $\Sigma_\mathrm{I} (= 2^I)$ and $\Sigma_\mathrm{O} (= 2^O)$ are the *input* and the *output alphabet*, respectively, $\mathrm{q}_0 \in Q$ 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*. The *input signals I*

and the *output signals O of* M are required to be distinct, i.e., $I \cap O = \emptyset$. 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., $\rho = q_0 q_1 \cdots \in \mathrm{Q}^\omega$ such that $\delta(q_i, w_i) = q_{i+1}$ for all $i \geq 0$. The *output word* M produces on w (denoted by $\mathrm{M_O}(w)$) 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 M on w, the output word is $\mathrm{M_O}(w) = l_0 l_1 \cdots \in \Sigma_\mathrm{O}^\omega$ with $l_i = \gamma(q_i, \mathrm{w}_i)$ for all $i \geq 0$. The *combined input output word* M produces on w is defined as $\mathrm{M(w)} := (i_0 \cup o_0)(i_1 \cup o_1) \ldots \in \Sigma_\mathrm{AP}^\omega$, where $\mathrm{w} = i_0 i_1 \ldots$ and $\mathrm{M_O(w)} = o_0 o_1 \ldots$. We denote by $\mathrm{L(M)}$ the *language* of M, i.e., the set of combined input/output words $\mathrm{L(M)} = \{\mathrm{M(w)} \mid \mathrm{w} \in \Sigma_I^\omega\}$.

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 $\mathrm{s}_0\, \mathrm{s}_1\, \mathrm{s}_2 \ldots \in \mathrm{S}^\omega$ 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 $\mathrm{s}_0\, \mathrm{s}_1 \ldots$ 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) [24] 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 $p \in \mathrm{AP}$. 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 $\mathrm{L(A)} = \mathrm{L}(\varphi)$ [22, 36].

We will use the following lemma in Section 4. It follows directly from the definition (i.e., from the fact that δ is a complete function).

Lemma 1 (Machine languages). *The language* $\mathrm{L(M)}$ *of any machine* $\mathrm{M} = (\mathrm{Q}, \Sigma_\mathrm{I}, \Sigma_\mathrm{O}, \mathrm{q}_0, \delta, \gamma)$ *is I-deterministic (input deterministic) and I-complete (input complete).*

Realizability and synthesis problem. The synthesis problem [6] asks to construct a system that satisfies a given formal specification. Given a language language L over the atomic propositions AP partitioned into input and output propositions, i.e., $\mathrm{AP} = I \cup O$, and a finite state machine M with input alphabet Σ_I and output alphabet Σ_O, we say that M *implements* (*realizes, or satisfies*) L, denoted by $\mathrm{M} \models L$, if $\mathrm{L(M)} \subseteq L$. We say language L is *realizable* if there exists a machine M that implements L. An LTL-formula φ is realizable if $\mathrm{L}(\varphi)$ is realizable.

Theorem 1 (Synthesis Algorithms [3, 25, 27]). *There exists a deterministic algorithm that checks whether a given LTL-formula (or an ω-regular language) φ is* realizable. *If φ is realizable, then the algorithm constructs* M.

```
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

3 Example

In this section we give a simple example to motivate our definitions and highlight the differences to previous approaches such as [18].

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 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 Figure 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 [18] 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(?)`

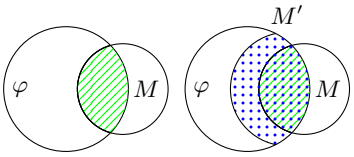


Fig. 2. Graphical representation of Def. 1

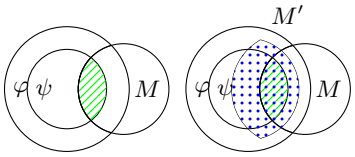


Fig. 3. Graphical representation of Def. 2

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

Definition 1 (Exact Repair). *A machine M' is an exact repair of a machine M for a specification φ, if (i) all the correct traces of M are included in the language of M', and (ii) if the language of M' is included in the language of the specification φ, i.e.,*

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

Note that the first inclusion defines the behavior of M' on all input words to which M responds correctly according to φ. In other terms, M' has only one choice forn inputs which M treat correctly. Figure 2 illustrates Definition 1: the two circles depict $\mathrm{L}(\mathrm{M})$ and $\mathrm{L}(\varphi)$. A repair has to (i) cover their intersection (first inclusion in Definition 1), which we depict with the striped area in the picture, and (ii) lie within $\mathrm{L}(\varphi)$ (second inclusion in Definition 1). One such repair is depicted by the dotted area on the right.

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 1. 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 according to Definition 1 replaces `if (sensor1)` by `if (sensor1 & !(light2 == RED & sensor2))`. This repair retains all correct traces while avoiding the mutual exclusion problem.

While Definition 1 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 1 is too restrictive). Assume a machine M with input r and output g that always copies r to g, i.e., M satisfies $\mathsf{G}(r \leftrightarrow g)$. The specification requires that g is eventually high, i.e., $\varphi = \mathsf{F}\, g$. Definition 1 requires the repaired machine M' to behave like M on all traces on which M behaves correctly. 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, M' has to mimic M as long as M still has a chance to satisfy φ (i.e., to produce a trace satisfying $\mathsf{F}(r \wedge g)$). Since M always has a chance to satisfy φ, M' has to behave like M in every step, therefore 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 in the following definition.

Definition 2 (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' implements φ. That is, M' is a repair constructed from M iff*

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

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 $\mathrm{L(M)}$ and $\mathrm{L}(\psi)$, but not the intersection of $\mathrm{L}(\varphi)$ and $\mathrm{L(M)}$. The repair lies completely in $\mathrm{L}(\varphi)$. The choice of ψ influences the existence of a repair. In Section 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., $\psi = \bigvee_{i=0..n} \mathsf{X}^n\, g$), then we can find a repair. A possible repair is a machine 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, M' continues mimicing M, otherwise it set g to high in step $n+1$ independent of the behavior of M. This way M' satisfies the specification ($\mathsf{F}\, g$) and mimics 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* M, M′ *be two machines with input signals I and output signal O. Machine* M′ *satisfies Formula 2* $(\mathrm{L(M)} \cap \mathrm{L}(\psi) \overset{(a)}{\subseteq} \mathrm{L(M')} \overset{(b)}{\subseteq} \mathrm{L}(\varphi))$ *if and only if* M′ *satisfies the following formula:*

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

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 *satisfies specification φ.*

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

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

This theorem leads together with [25] 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* M *with respect to specifications ψ and φ if and only if the language*

$$\big((\mathrm{L(M)} \cap \mathrm{L}(\psi))\downarrow_I\uparrow_{AP} \rightarrow \mathrm{L(M)}\big) \cap \mathrm{L}(\varphi) \tag{4}$$

is realizable.

4.3 Algorithm

Corollary 1 gives an algorithm to construct repairs based on synthesis techniques (cf. [18]). 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 complementation, 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 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 4, 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 [10]. 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 [23, 32]. 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* 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:*

$$\Sigma_I^\omega \setminus \big(\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\varphi)\big)\downarrow_I = \big(\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\neg\varphi)\big)\downarrow_I \tag{5}$$

$$\Sigma_{AP}^\omega \setminus \big(\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\varphi)\big)\downarrow_I\uparrow_{AP} = \big(\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\neg\varphi)\big)\downarrow_I\uparrow_{AP} \tag{6}$$

Proof. Intuitively, Equation 5 means that the set of input words on which M behaves correctly, i.e., satisfies φ, is the complement of the set of inputs on which 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

$$\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\neg\varphi) \overset{(a)}{=} \mathrm{L}(\mathrm{M})\cap\big(\Sigma^\omega\setminus\mathrm{L}(\varphi)\big) \overset{(b)}{=} \mathrm{L}(\mathrm{M})\setminus\mathrm{L}(\varphi). \tag{7}$$

Equality 7.b follows from simple set theory. Furthermore, since $\mathrm{L}(\mathrm{M})$ is input deterministic and input complete, we know that

$$\forall\, w, w' \in \mathrm{L}(\mathrm{M}) : (w\downarrow_I = w'\downarrow_I) \rightarrow w = w' \tag{8}$$

$$\forall\, w \in \Sigma_{AP}^\omega : \exists\, w' \in \mathrm{L}(\mathrm{M}) : w\downarrow_I = w'\downarrow_I \tag{9}$$

We use these facts to show that for all $A \subseteq \Sigma^\omega$, $\Sigma_I^\omega\setminus\big(\mathrm{L}(\mathrm{M})\cap A\big)\downarrow_I = \big(\mathrm{L}(\mathrm{M})\setminus A\big)\downarrow_I$ holds, which proves together with Equation 7 that Equation 5 is true.
$v \in \big(\mathrm{L}(\mathrm{M})\setminus A\big)\downarrow_I \iff \exists\,\mathrm{w}\in\mathrm{L}(\mathrm{M})\setminus A : (\mathrm{w}\downarrow_I = v) \iff \exists\,\mathrm{w}\in\mathrm{L}(\mathrm{M}) : (\mathrm{w}\downarrow_I = v)\wedge \mathrm{w}\notin A \overset{\text{Eq.8}}{\underset{\text{Eq.9}}{\iff}} \forall\,\mathrm{w}\in\mathrm{L}(\mathrm{M}) : (\mathrm{w}\downarrow_I = v)\rightarrow \mathrm{w}\notin A \iff \forall\,\mathrm{w}\in\mathrm{L}(\mathrm{M}) : \mathrm{w}\in A \rightarrow (\mathrm{w}\downarrow_I \neq v) \iff \forall\,\mathrm{w}\in\mathrm{L}(\mathrm{M})\cap A : (\mathrm{w}\downarrow_I \neq v) \iff \nexists\,\mathrm{w}\in\mathrm{L}(\mathrm{M})\cap A : (\mathrm{w}\downarrow_I = v) \iff v\notin\big(\mathrm{L}(\mathrm{M})\cap A\big)\downarrow_I$

Equation 6 is a simple extension of Equation 5 to the alphabet Σ_{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 4 to

$$\big((\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\neg\psi))\downarrow_I\uparrow_{AP}\cup\mathrm{L}(\mathrm{M})\big)\cap\mathrm{L}(\varphi) \tag{10}$$

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

Proof. $\mathrm{w} \in (\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi))\downarrow_I\uparrow_{AP} \iff \mathrm{w}\downarrow_I \in (\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi))\downarrow_I \iff \exists\, \mathrm{w}' \in \mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi) : \mathrm{w}'\downarrow_I = \mathrm{w}\downarrow_I \iff \exists\, \mathrm{w}' \in \mathrm{L}(\mathrm{M}) : \mathrm{w}'\downarrow_I = \mathrm{w}\downarrow_I \wedge \mathrm{w}' \in \mathrm{L}(\varphi) \iff \mathrm{M}(\mathrm{w}\downarrow_I) \in \mathrm{L}(\varphi)$

Due to Lemma 3 we can check if a word produced by M' lies in $(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi))\downarrow_I\uparrow_{AP}$ by checking whether M treats the input projection of that word correctly. A synthesizer looking for a solution to Equation 10 can simulate M and check its output against $\neg\psi$ to decide whether M' is allowed to deviate from M. This allows us to solve our repair problem using the simple setup depict in Figure 4. It shows five automata running in parallel:

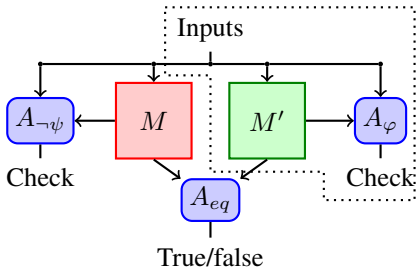


Fig. 4. Efficient implementation

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

Theorem 3. *Given the setup depicted in Figure 4, a repair option in* M' *is a valid repair according to Definition 2, if it satisfies the formula*

$$\varphi \wedge (\neg\psi \vee eq). \tag{11}$$

Proof. Follows from Lemma 2 and Lemma 3.

Formula 11 forces M' to (1) behave according to φ and (2) mimic the behavior of M, if 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 [4] (combinations of choices), then the choice of repair can be encoded using multiple initial states.

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

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 Solar-Lezama et al. [33] 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 Section 6 for more details). Furthermore, we assume a given fault location that will be replaced by one of the user-defined expressions (cf. [18, 19]). Expression generation and fault localization are interesting and active research directions (cf. Section 1) but are not addressed in this paper. 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 are significantly more mature than synthesis frameworks. In our implementation we used a version of NuSMV [7] that we slightly modified to return an initial state that does not lead to a counter example.

5 Discussion and Limitations

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

5.1 Choices for ψ

We present three different choices for ψ and analyze their strengths and weaknesse:
(1) $\psi = \varphi$, (2) if $\varphi = f \rightarrow g$, 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 5. Let M be a machine with input r and output g; M always outputs $\neg g$, i.e., 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 10, we obtain $(\mathsf{G}(\neg g) \wedge \neg(\mathsf{G}(r) \vee \mathsf{G}(g)))\downarrow_I\uparrow_{AP}$[5] $\wedge\ (\mathsf{G}(r) \vee \mathsf{G}(g)) = (\mathsf{F}(\neg r) \wedge \mathsf{G}(g)) \vee (\mathsf{G}(r) \wedge \mathsf{G}(\neg g))$. 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 $\psi = \varphi$.

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

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 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 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 [18].

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 [30] identified two reasons for a specification φ to be unrealizable.

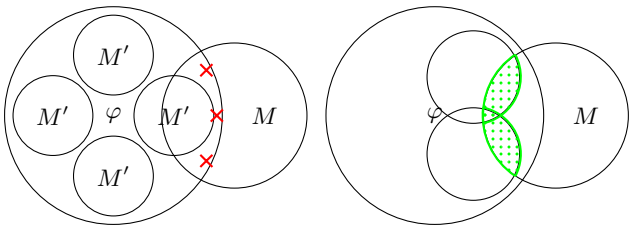
1. **Input-Completeness:** if φ is not input-complete, then φ is not realizable. For instance, consider specification $\mathsf{G}(r)$ requiring that r is always true. If r is an input to the system, the system cannot choose the value of r and therefore also not guarantee satisfaction of φ.
2. **Causality/Clairvoyance:** 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 $\big((\mathrm{L(M)} \cap \mathrm{L}(\psi))\downarrow_I \rightarrow \mathrm{L(M)}\big) \cap \mathrm{L}(\varphi)$ is input-complete.*

Proof. Let $\mathrm{w}_I \in {\Sigma_\mathrm{I}}^\omega$. If $\mathrm{w}_I \in (\mathrm{L(M)} \cap \mathrm{L}(\psi))\downarrow_I$, then there is a word $\mathrm{w} \in \mathrm{L(M)} \cap \mathrm{L}(\psi)$ such that $\mathrm{w}\downarrow_I = \mathrm{w}_I$. 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 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.



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

Fig. 5. Two reasons for unrepairability

Example 7. 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 1) 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 8. 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 4 (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 [30], Corollary 1, and Lemma 4.

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 5 we use the same graphical representations to explain two reasons for failure. Figure 5(a) depicts several machines M' realizing φ. A repair of M has to be one of the machines realizing φ. As observed in [13], there are words satisfying φ that cannot be produced by any correct machine (depicted as red crosses in Figure 5(a)). 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[6] 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 Figure 5(b). We again depict 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 Empirical Results

In this section we first describe the repair we synthesized for the traffic light example from Section 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.

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 Section 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 [18].

Experiments. 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 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 Section 4.4 (Figure 4), i.e., we manually added freedom to the model and wrote formula for $\neg\psi$ and equality checking. The examples are described in detail in the extended version[7] For all but one of the examples (Processor (1)), the previous approach synthesizes degenerated repairs, while our approach leads to a correct program repair.

Assume-Guarantee ($\rightarrow$) is Example 5 from Section 5.1. 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

[6] Two formulas φ and φ' are open-equivalent if any machine M implementing φ also implements φ' and vice-versa [13].

[7] The NuSMV models of the example and our implementation are available at `http://www-verimag.imag.fr/~vonessen/`.

of g. Our approach fails to find a repair. Assume-Guarantee ($\&$) is Example 6 from Section 5.1 with $\psi = \mathsf{F}(\neg r) \wedge \mathsf{G}(g)$, using the same potential repairs. In this case, a valid repair is found. The Binary Search examples model a binary search algorithm with a specification analogous to $sorted \rightarrow correct$, i.e., when the array is sorted, then the algorithm responds with the correct result. The bug is an incorrect assignment of the pointer into the array. The repais we allow are arithmetic combinations of the previous position, 1, -1, the lower bound and the upper bound. As in the Assume-Guarantee examples, we have two different choices for ψ here. In the case that $\psi = \varphi$, there is no repair available, while for $\psi = sorted \wedge correct$ we find the correct repair. The RW-Lock example demonstrates that our approach can also be used to synthesize locks. We require that only those program runs are changed that lead to a dead-lock, thereby synthesizing the minimum amount of locks. The potential repairs allow 16 different locking combinations, only one of which is optimal. The optimal solution is the only one admitted by our repair definition.

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 [18], we would need to increase the size of the specification, which will significantly slow down the approach.

7 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 repaired 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 7, 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 a relaxed notion of set-inclusion or equality in order to express how "close" two machines are. To extend the applicability of our approach to infinite-state programs, we will explore suitable program abstraction techniques (cf. [34]). Finally, we are planing to experiment with model checkers specialized in solving the sequential equivalence checking problem [20, 21]. We believe that such solvers perform well on our problem, because M' and M have many similar structures.

		Verification		Repair		Classical Repair	
	#Repairs	time	#Vars	time	#Vars	time	#Vars
Assume-Guarantee ($\rightarrow$)	2^{12}	n/a	n/a	0.038	16	0.012	14
Assume-Guarantee ($\&$)	2^{12}	0.015	14	0.025	14	0.012	12
Binary Search ($\rightarrow$)	5	n/a	n/a	0.78	27	0.1	21
Binary Search ($\&$)	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	2m02s	135	2m41s	135	0.5	69
Processor (2)	4	4m28s	138	5m07s	138	0.5	69
Processor (3)	25	5m23s	140	18m05s	140	0.5	71

Table 1. Experimental results

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 paper, 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. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: Localizing errors in counterexample traces. In: POPL 2003. (January 2003) 97–105
2. Buccafurri, F., Eiter, T., Gottlob, G., Leone, N.: Enhancing model checking in verification by ai techniques. Artif. Intell. **112**(1-2) (1999) 57–104
3. Büchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Transactions of the American Mathematical Society **138** (1969) 295–311
4. Chandra, S., Torlak, E., Barman, S., Bodik, R.: Angelic debugging. In: ICSE 2011, New York, NY, USA, ACM (2011) 121–130
5. Chang, K.H., Markov, I.L., Bertacco, V.: Fixing design errors with counterexamples and resynthesis. IEEE Trans. on CAD **27**(1) (2008) 184–188
6. Church, A.: Logic, arithmetic and automata. In: Proc. 1962 Int. Congr. Math. (1963)
7. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In: CAV. (2002)
8. Clarke, E., Grumberg, O., McMillan, K., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: DAC. (1995)
9. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Logic of Programs. (1981)
10. Drusinsky, D., Harel, D.: On the power of bounded concurrency i: Finite automata. J. ACM **41**(3) (1994) 517–539
11. Ebnenasir, A., Kulkarni, S.S., Bonakdarpour, B.: Revising unity programs: Possibilities and limitations. In: OPODIS. Volume 3974 of LNCS. (2005)

12. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Trail-directed model checking. ENTCS **5**(3) (August 2001) Software Model Checking Workshop 2001.
13. Greimel, K., Bloem, R., Jobstmann, B., Vardi, M.: Open implication. In: ICALP. (2008) 361–372 LNCS 5126.
14. Griesmayer, A., Bloem, R., Cook, B.: Repair of boolean programs with an application to c. In: CAV. Volume 4144 of LNCS. Springer (2006) 358–371
15. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: 10th International SPIN Workshop, Springer-Verlag (2003) 121–135 LNCS 2648.
16. Janjua, M.U., Mycroft, A.: Automatic correction to safety violations in programs. Thread Verification (TV'06) (2006) Unpublished.
17. Jin, H., Ravi, K., Somenzi, F.: Fate and free will in error traces. In: TACAS'02, Grenoble, France (April 2002) 445–459 LNCS 2280.
18. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: CAV. Volume 3576 of Lecture Notes in Computer Science., Springer (2005) 226–238
19. Jobstmann, B., Staber, S., Griesmayer, A., Bloem, R.: Finding and fixing faults. J. Comput. Syst. Sci. **78**(2) (2012) 441–460
20. Kaiss, D., Skaba, M., Hanna, Z., Khasidashvili, Z.: Industrial strength sat-based alignability algorithm for hardware equivalence verification. In: FMCAD. (2007) 20–26
21. Khasidashvili, Z., Moondanos, J., Kaiss, D., Hanna, Z.: An enhanced cut-points algorithm in formal equivalence verification. In: HLDVT. (2001) 171–176
22. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: POPL. (1985) 97–107
23. Piterman, N.: From nondeterministic buchi and streett automata to deterministic parity automata. Logical Methods in Computer Science **3**(3) (2007) 5
24. Pnueli, A.: The temporal logic of programs. In: FOCS, IEEE Comp.Soc. (1977)
25. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. (1989) 179–190
26. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Symposium on Programming. (1982) 337–351
27. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Transactions of the American Mathematical Society **141** (1969) 1–35
28. Ravi, K., Somenzi, F.: Minimal assignments for bounded model checking. In: TACAS'04, Barcelona, Spain (March-April 2004) 31–45 LNCS 2988.
29. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ICASE, Montreal, Canada (October 2003) 30–39
30. Rosner, R.: Modular Synthesis of Reactive Systems. PhD thesis, Stanford University (1997)
31. Samanta, R., Deshmukh, J.V., Emerson, E.A.: Automatic generation of local repairs for boolean programs. In Cimatti, A., Jones, R.B., eds.: FMCAD. (2008) 1–10
32. Schewe, S.: Tighter bounds for the determinisation of büchi automata. In: FOSSACS. (2009) 167–181
33. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: PLDI. (2005) 281–294
34. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL. (2010) 327–338
35. Vechev, M., Yahav, E., Yorsh, G.: Inferring synchronization under limited observability. In: TACAS'09. Volume 5505 of LNCS. Springer (2009) 139–154
36. Wolper, P., Vardi, M.Y., Sistla, A.P.: Reasoning about infinite computation paths (extended abstract). In: FOCS, IEEE (1983) 185–194
37. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering **28**(2) (2002) 183–200