Program Repair without Regret

Barbara Jobstmann
EPFL and Jasper DA
CNRS, Verimag

Joint Work with
Christian von Essen
UJF, Verimag
Google Zurich

Agenda

- Introduction
  - Motivation
  - Program Repair, related choices, our choices
  - Example
- Our Repair approach
  - Exact and relaxed repair problem
  - Reduction to classical synthesis
  - General and efficient algorithm
  - Implementation and results
- Conclusion and future work

Motivation

- Debugging can be tedious
  - Find the bug
  - Locate it
  - Fix it

Motivation

- Debugging can be tedious
  - Find the bug: model checking
  - Locate it: automatically analyze/modify/explain CEX/witness
  - Fix it: automatically repair

Automatic Repair

- Given faulty program + (explicit/implicit) specification
- Search for modification s. t. modified program is
  - “correct” and
  - “similar” to the original program
- Key choices in an repair approach:
  - Type of programs and specifications
  - Which modification do you allow?
  - How to you find and check corrections?
  - What do you mean by “similar”?

Key choices

1. Type of programs and specifications
   – Data or control-oriented
   – Specific properties (e.g., deadlock), general properties (e.g., given in a logic), explicit/implicit
2. Which modification do you allow?
   – Syntactic modifications, e.g., based on expression language, genetic algorithms, …
3. How to you find and check corrections?
   – “Smart” enumeration and verification
   – “Synthesize” (combine search and verification)
4. What do you mean by “similar”?
   – Focus on syntactic similarity (e.g., edit distance, …)

Our Choices [following CAV'05]

1. Type of programs and specifications
   - Reactive finite-state programs (Mealy machines)
   - General properties specified using LTL)
2. Which modification do you allow?
   - Theory: functions over state/input variables
   - Implementation: expression language
3. How to you find and check corrections?
   - Combine search and verification using game theory
4. What do you mean by "similar"?
   - Syntactic similarity (given by expression language)
   - **Semantic similarity (NEW in this work)**

Choice 1: Programs and specifications

```
mainLight = Red;
sideLight = Red;
always @(posedge clock)
begin
  case (mainLight)
    Red: if (mainSensor)
             mainLight = Yellow;
    Yellow: mainLight = Green;
    Green:  mainLight = Red;
  endcase // case (mainLight)
  case (sideLight)
    Red: if (sideSensor)
             sideLight = Yellow;
    Yellow: sideLight = Green;
    Green:  sideLight = Red;
  endcase // case (sideLight)
end
```

State variables:

mainLight in {Red, Yellow, Green}

sideLight in {Red, Yellow, Green}

Input variables:

mainSensor in {True, False}

sideSensor in {True, False}

Behavior represented as (infinite) sequence of evaluations of state and input variables: $w \in E(V)^{\omega}$

Program represented as set of behaviors: $L(P)$

mL	Red	Yellow	Green	Red	…
sL	Red	Red	Red	Yellow	…
mS	True	True	False	…	…
sS	False	False	True	…	…

Choice 1: Programs and specifications

```
mainLight = Red;
sideLight = Red;
always @(posedge clock)
begin
  case (mainLight)
    Red: if (mainSensor)
               mainLight = Yellow;
    Yellow: mainLight = Green;
    Green:  mainLight = Red;
  endcase // case (mainLight)
  case (sideLight)
    Red: if (sideSensor)
               sideLight = Yellow;
    Yellow: sideLight = Green;
    Green:  sideLight = Red;
  endcase // case (sideLight)
end
```

State variables:

mainLight in {Red, Yellow, Green}

sideLight in {Red, Yellow, Green}

Input variables:

mainSensor in {True, False}

sideSensor in {True, False}

Behavior represented as (infinite) sequence of evaluations of state and input variables: $w \in E(V)^{\omega}$

Program represented as set of behaviors: $L(P)$

Specification represented as set of behaviors: $L(\varphi)$

Specification:
never(mainLight == Green
and
sideLight == Green)

Choice 2: Modifications

```
mainLight = Red;
sideLight = Red;
always @(posedge clock)
begin
 case (mainLight)
  Red: if (   ???   )
          mainLight = Yellow;
  Yellow: mainLight = Green;
  Green:  mainLight = Red;
 endcase // case (mainLight)
 case (sideLight)
  Red: if (sideSensor)
          sideLight = Yellow;
  Yellow: sideLight = Green;
  Green:  sideLight = Red;
 endcase // case (sideLight)
end
```

Specification:
never(mainLight == Green
and
sideLight == Green)

State variables:

mainLight in {Red, Yellow, Green}

sideLight in {Red, Yellow, Green}

Input variables:

mainSensor in {True, False}

sideSensor in {True, False}

Behavior represented as (infinite) sequence of evaluations of state and input variables: $w \in E(V)^{\omega}$

Program represented as set of behaviors: $L(P)$

Specification represented as set of behaviors: $L(\varphi)$

Allowed modifications:
function over state and input variables

Choice 3: Repair Search using Games

```
mainLight = Red;
sideLight = Red;
always @(posedge clock)
begin
  case (mainLight)
    Red: if (    ???    )
              mainLight = Yellow;
    Yellow: mainLight = Green;
    Green:  mainLight = Red;
  endcase // case (mainLight)
  case (sideLight)
    Red: if (sideSensor)
              sideLight = Yellow;
    Yellow: sideLight = Green;
    Green:  sideLight = Red;
  endcase // case (sideLight)
end
```

Specification:
never(mainLight == Green
and
sideLight == Green)

State variables:

mainLight in {Red, Yellow, Green}

sideLight in {Red, Yellow, Green}

Input variables:

mainSensor in {True, False}

sideSensor in {True, False}

Behavior represented as (infinite) sequence of evaluations of state and input variables: $w \in E(V)^{\omega}$

Program represented as set of behaviors: $L(P)$

Specification represented as set of behaviors: $L(\varphi)$

Allowed modifications:
function over state and input variables

Winning objective: repaired program is correct, i.e.,
$L(P') \subseteq L(\varphi)$

Choice 4: Similarity

```
mainLight = Red;
sideLight = Red;
always @(posedge clock)
begin
  case (mainLight)
    Red: if (   ???   )
              mainLight = Yellow;
    Yellow: mainLight = Green;
    Green:  mainLight = Red;
  endcase // case (mainLight)
  case (sideLight)
    Red: if (sideSensor)
              sideLight = Yellow;
    Yellow: sideLight = Green;
    Green:  sideLight = Red;
  endcase // case (sideLight)
end
```

Specification:
never(mainLight == Green
and
sideLight == Green)

State variables:

mainLight in {Red, Yellow, Green}

sideLight in {Red, Yellow, Green}

Input variables:

mainSensor in {True, False}

sideSensor in {True, False}

Behavior represented as (infinite) sequence of evaluations of state and input variables: $w \in E(V)^{\omega}$

Program represented as set of behaviors: $L(P)$

Specification represented as set of behaviors: $L(\varphi)$

Allowed modifications:
"simple" function over state and input variables

Winning objective: repaired program is correct, i.e.,

$$L(P') \subseteq L(\varphi)$$

Simple Repair

```
mainLight = Red;
sideLight = Red;
always @(posedge clock)
begin
  case (mainLight)
    Red: if (mainSensor & !(sideLight == Red & sideSensor))
              mainLight = Yellow;
    Yellow: mainLight = Green;
    Green:  mainLight = Red;
  endcase // case (mainLight)
  case (sideLight)
    Red: if (sideSensor)
              sideLight = Yellow;
    Yellow: sideLight = Green;
    Green:  sideLight = Red;
  endcase // case (sideLight)
end
```

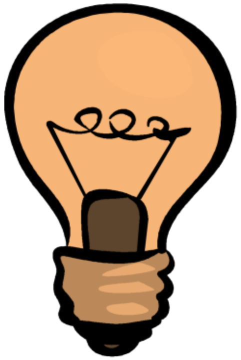
Specification:
never(mainLight == Green
and
sideLight == Green)

☺ No car crash: correct repair

☹ Main street blocked

What went wrong?

Lost intended behavior; changed
cessarily



Idea: semantic similarity

- Keep correct behaviors
- Modifications must not effect correct behaviors

[Angelic debugging, Chandra et al.]

Extend objective: repair keeps correct behaviors
$L(P) \cap L(\varphi) \subseteq L(P')$

Winning objective: repaired program is correct, i.e.,
$L(P') \subseteq L(\varphi)$

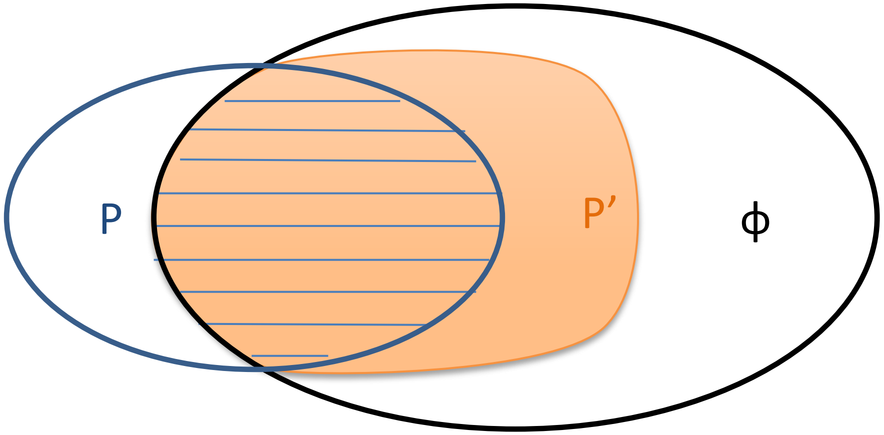
Agenda

- Introduction
  - Motivation
  - Program Repair, related choices, our choices
  - Example
- **Our Repair approach**
  - Exact and relaxed repair problem
  - Reduction to classical synthesis
  - General and efficient algorithm
  - Implementation and results
- Conclusion and future work

Exact Repair Problem

Program P' is an **"exact repair"** of program P for specification φ if
(i) all correct behaviors of P w.r.t. φ are part of P',
(ii) all behaviors of P' are correct w.r.t. φ, i.e.,

$$L(P) \cap L(\varphi) \subseteq L(P') \subseteq L(\varphi)$$



Ideal but sometimes too restrictive:

- exact repair might not exists
- exact repair might not be required

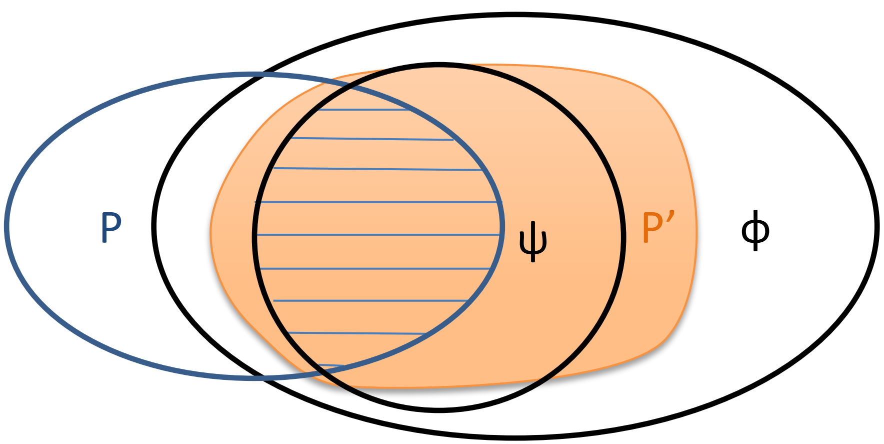
$$\varphi = \varphi_a \rightarrow \varphi_g$$

Behaviors that do not satisfy φ_a are correct but might not need to be preserved

Relaxed Repair Problem

Program P' is an **“relaxed repair”** of program P for specifications φ and ψ if
(i) all correct behaviors of P w.r.t. ψ are part of P',
(ii) all behaviors of P' are correct w.r.t. φ, i.e.,

$$L(P) \cap L(\psi) \subseteq L(P') \subseteq L(\varphi)$$



Some choices for ψ

- Exact repair $\psi = \varphi$
- Assume-Guarantee:
  if $\varphi = \ \varphi_a \rightarrow \varphi_g$, then $\psi = \varphi_a \land \varphi_g$
- Classical repair $\psi = \emptyset$

Reduction to Classical Synthesis

Given two specifications φ and ψ over variables $V = I \cup O$ and two programs P and P', then P' is a relaxed repair of P if and only if P' satisfies the following formula

$$L(P') \subseteq \alpha \text{ with}$$

$$\alpha = \underbrace{\left(E(V)^{\omega} \setminus \left(L(P) \cap L(\psi)\right) \downarrow_I \uparrow_V\right.}_{\text{(All behaviors with a sequence of inputs for which P violates the spec)}} \left.\cup\, L(P)\right) \cap L(\varphi)$$

There exists a relaxed repair for program P w.r.t. φ and ψ if and only if language α is realizable.

General Algorithm

1. Construct an (Buchi) automaton A_α for
   $\alpha = \left(E(V)^\omega \setminus \left(L(P) \cap L(\psi)\right) \downarrow_I \uparrow_V \cup L(P)\right) \cap L(\varphi)$
   A_α can be constructed because
   - P, ψ, φ can be represented as Buchi automata and
   - Buchi automata are closed under intersection, union, projection, and complementation
2. Synthesize P' using A_α as specification

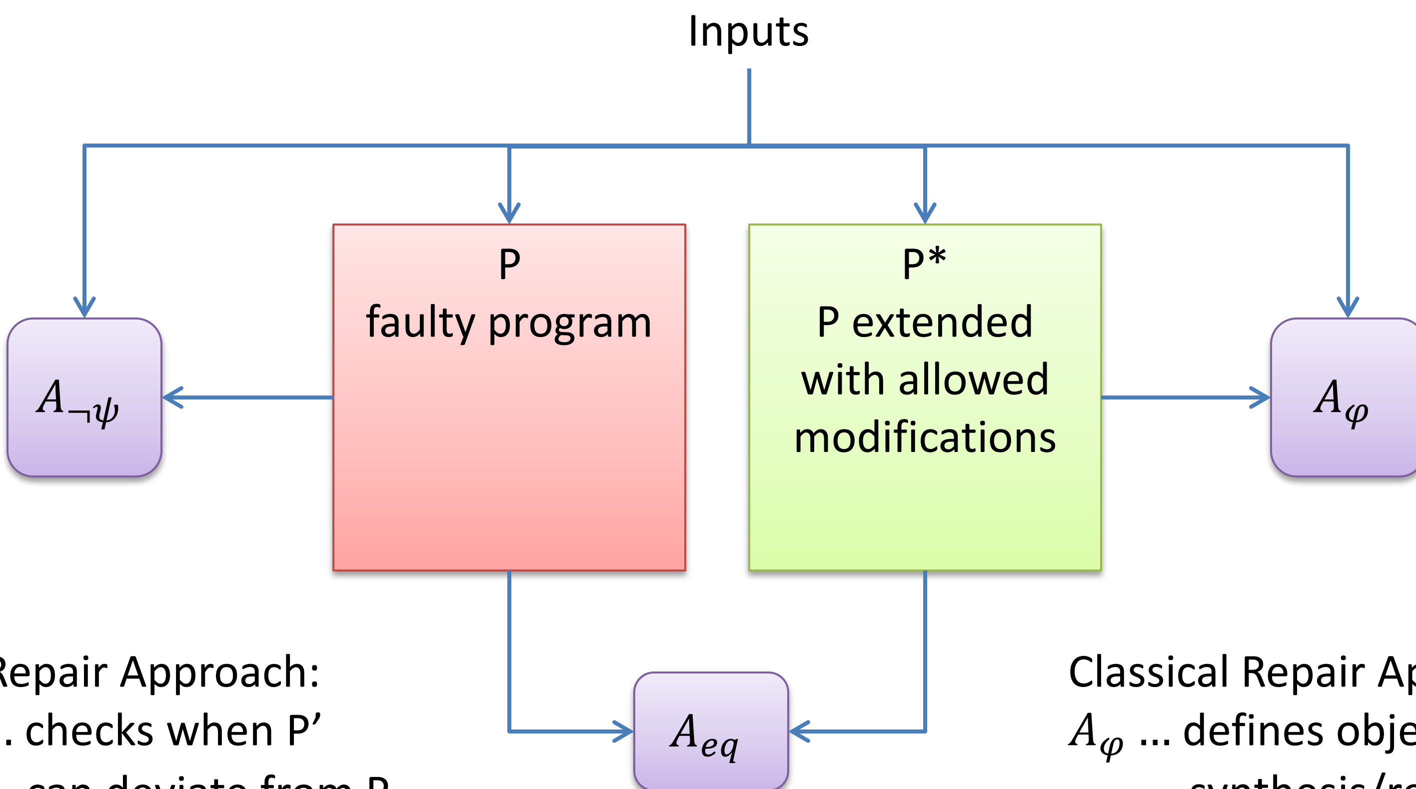
Efficient Algorithm (1)

Lemma: Given program P and LTL formula ψ over variables $V = I \cup O$, for all words $w \in E(V)^\omega$

$$w \in \big(L(P) \cap L(\psi)\big) \downarrow_I \uparrow_V \Leftrightarrow P(w \downarrow_I) \in L(\psi)$$

- This enables a simple procedure to check if a word produced by P' lies in $\big(L(P) \cap L(\psi)\big) \downarrow_I \uparrow_V$
- A synthesizer searching for P' that satisfies α, can simulate P and check it against ψ to decide if P' is allowed to deviate.

Efficient Algorithm (2)



New Repair Approach:
$A_{\neg\psi}$ … checks when P’
can deviate from P
A_{eq} … checks when the output
of P and P’ coincide

Classical Repair Approach:
A_{φ} … defines objective of
synthesis/repair game

New Objective:
A_{φ} and ($A_{\neg\psi}$ or A_{eq})

Implementation

- Ideas/choice:
  - Modification restricted to expressions
  - Use MC instead of game engine:
    - encode "interesting" strategies using initial states (e.g., all memoryless strategies)
    - an initial state that does not lead to a CEX gives a correct repair.
    - Drawbacks: explodes with considered strategies
    - Benefits: can make use of any MC (and SEC optimizations)
- Prototype based on NuSMV (with Cudd)
  - Tiny modification to return initial state(s) without CEX

PCI arbiter (partial specification)

- PCI Bus
- n Devices
- n specifications
  always(r_i **implies eventually** g_i)
  checked in isolation
- Off by 1 error
- Approach without lower bound gives access to device i forever
- Our approach fixes off by 1 error

Processor (unclear error location)

- Error in one of the ALUs
- Partial specification
- Multiple suspected error locations
- Approach without lower bound approach may modify all ALUs
- Our approach only modifies faulty ALU

Read-Write-Lock (minimal locking)

- Faulty implementation leads to dead-lock
- Allow introduction of a lot of locking
- Approach without lower bound may lock everything
- Our approach introduces only necessary locks

Preliminary Results

	#Repairs	Verification time	Verification #Vars	Repair time	Repair #Vars	Classical Repair time	Classical Repair #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

Agenda

- Introduction
  - Motivation
  - Program Repair, related choices, our choices
  - Example
- Our Repair approach
  - Exact and relaxed repair problem
  - Reduction to classical synthesis
  - General and efficient algorithm
  - Implementation and results
- **Conclusion and future work**

Conclusions

- New notion of program repair that
  - ensures that correct behaviors are kept
  - supports fixing bugs incrementally
  - facilitates repair with incomplete specifications
  - avoids degenerated repairs
- Algorithm based on reactive program synthesis
- Preliminary implementation supporting our claims of obtaining “better” repairs

Future Work

- Quantitative notions of semantic similarity
  - Count ratio of modified paths vs all paths
  - Count ratio of modified symbols
  - Define distance measure between symbols
- Repairs with look-ahead to increase the number of repairable systems
- Repairing infinite-state system