“Modular and Verified Automatic Program Repairs”

from Francesco Logozzo and Thomas Ball at Microsoft Research, Redmond

presenter name(s) removed for FERPA considerations

Introduction

Your programs will have bugs!

...and it would be useful to catch these bugs before running your program.

There exist automatic design time programs that report bugs to the developer.

...but they do not automatically provide the repairs for those bugs.

Wouldn’t it be easier if code repairs were automatically suggested to you as you write buggy code?

Research Questions

How do we verify a satisfactory code repair?

Can code repairs be generated for every type of bug found?

Are the code repairs being suggested fast enough to be useful in an active development use case?

The Approach

During design time, automatically suggest code repairs that address bug warnings reported by static analyzers at design time.

- Repairs need to be “verified” for correctness
- Framework needs to be “modular” and work off of different static analyzers
- No program runs or test-suites
- Works on incomplete code

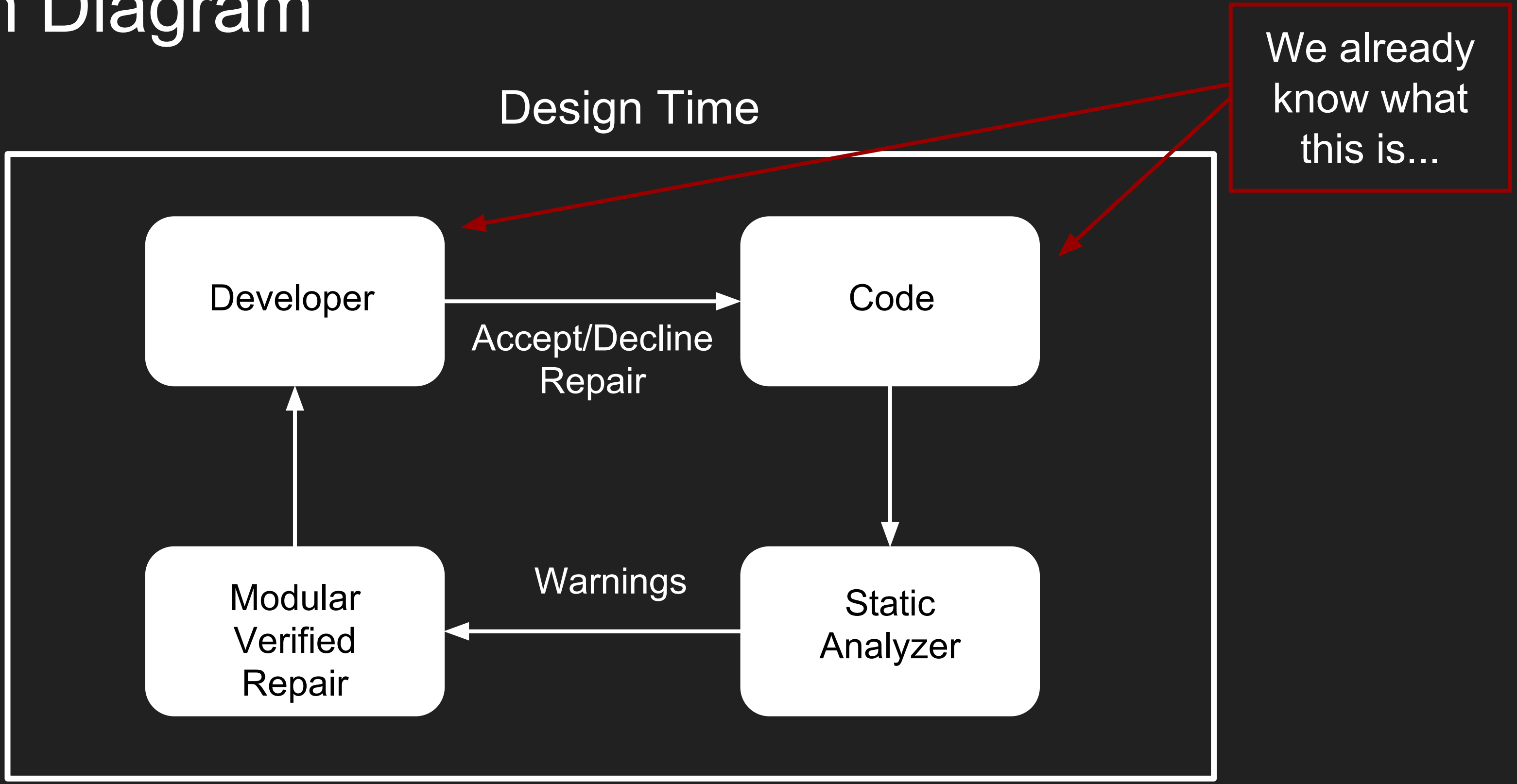
Contributions

Defined the notion of a “verified” repair.

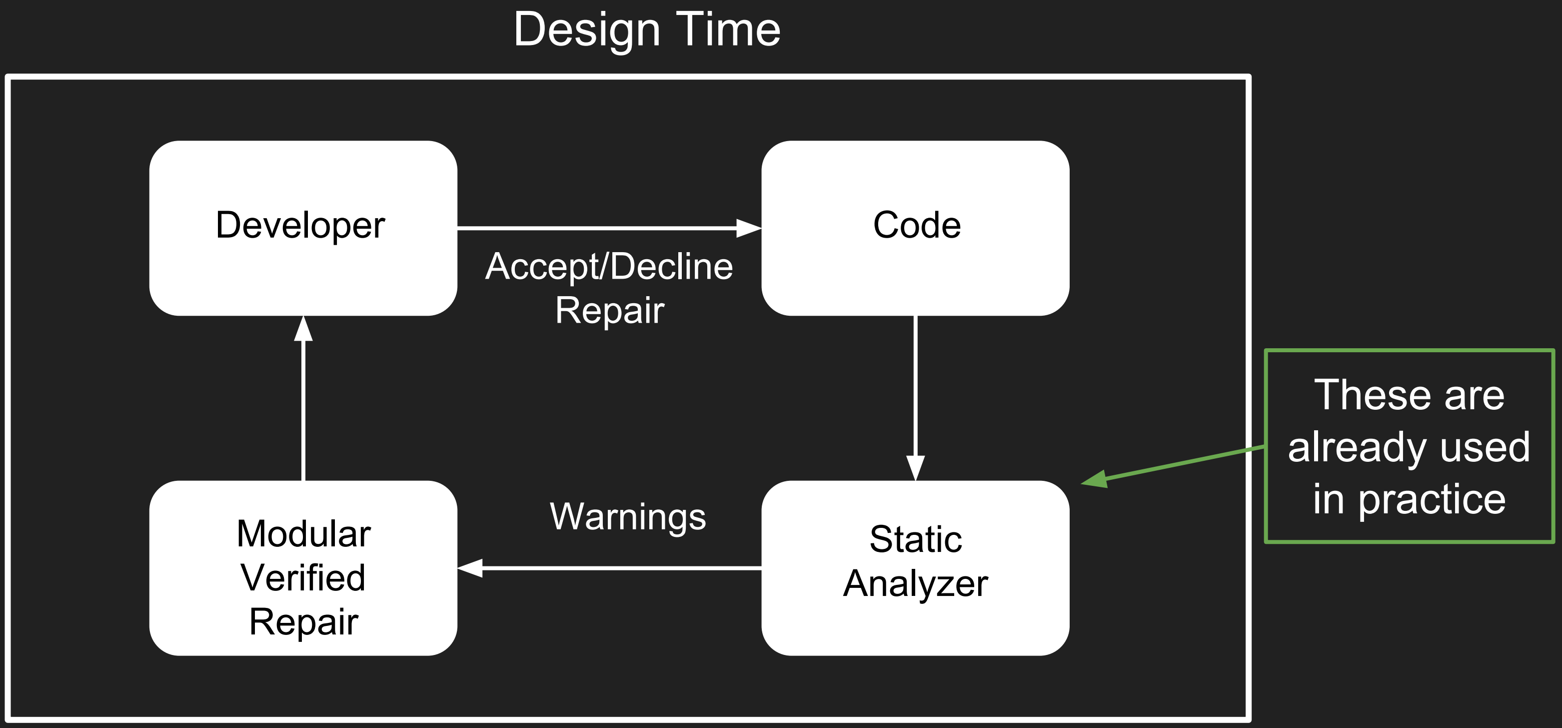
Proposed sound algorithms for suggesting verified code repairs that address bugs at design time.

Evaluated implementation to be accurate and fast enough for use in IDE.

System Diagram



System Diagram

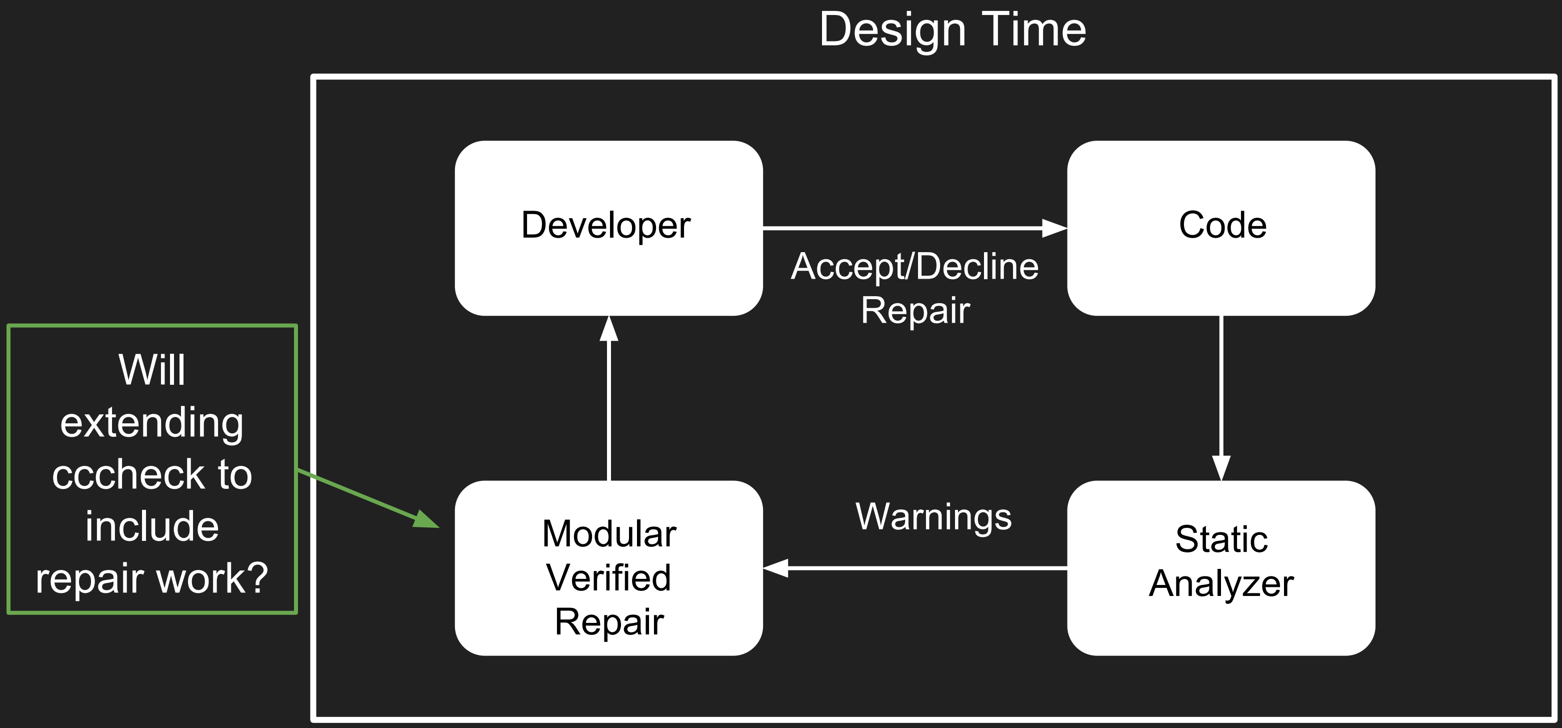


cccheck: a static analyzer

Static analysis that will automatically detect buggy code during design time.
(No program runs or test suite needed)

-Missing contracts
-Incorrect locals
-Incorrect object initialization
-Wrong conditionals
-Buffer overruns
-Arithmetic overflows
-Incorrect floating point comparisons

System Diagram



Trace Semantics

“Traces are a sequence of states…”

Runs

What it means to be a good or bad run

Verified Program Repair

What makes a repair a “good” repair?

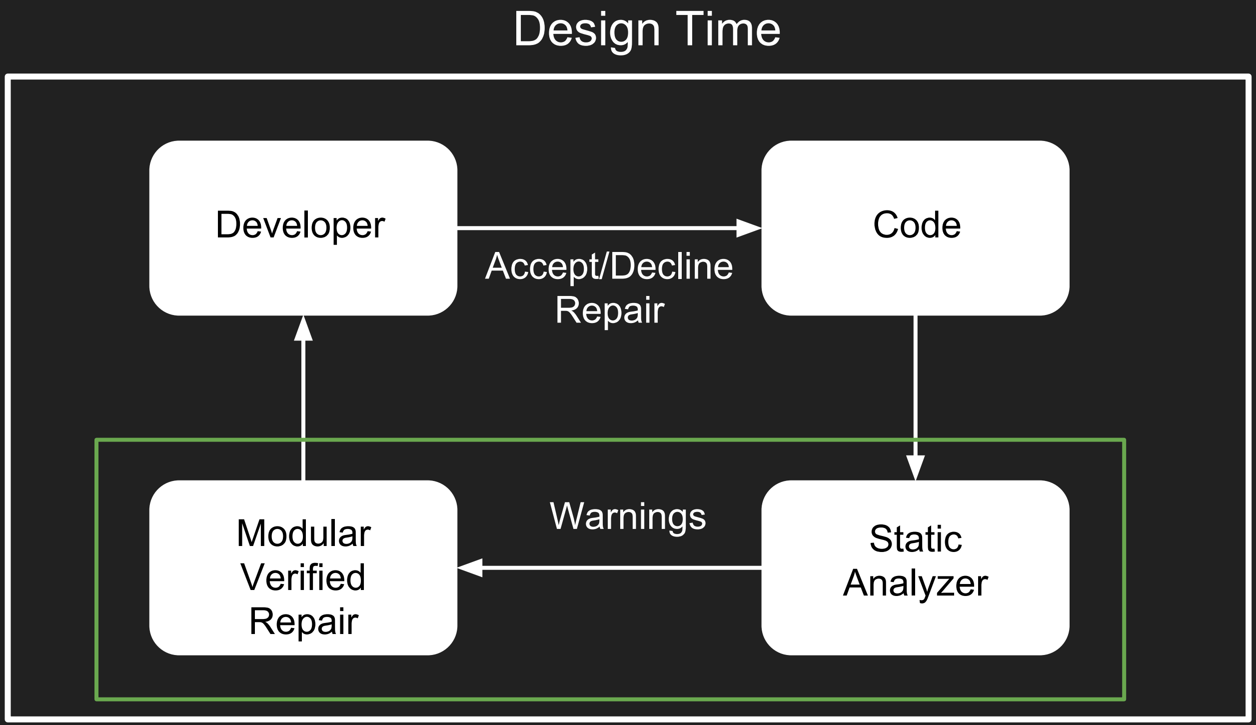
Repair should mean greater good runs and fewer bad runs.

Verified Assertion Repair

Abstracting what verified repair means

How can you determine an improved assertion?

System Diagram



Program Repairs from a Static Analyzer

So does the repair program actually work with a static analyzer?

(1) “cccheck” does not require program runs and does not require test-suites.

(2) By construction, any verified code repair will not break your code (assuming that your assertions are defined correctly).

(3) The verified code repairs are design-time suggestions, and are not applied unless the developer approves it.

(4) Authors claim the repair program can be generalized for ALL static analyzers.

cccheck: Static Analysis Phases (part 1/2)

(1) Gather assertions
  - Provided as the developer codes
  - Can also be inferred by language semantics

(2) Infer facts
  - Abstract interpretation in the abstract domains of heap abstraction, null checks, scalable numerical analysis, universally and existentially quantified properties, and floating point comparisons

cccheck: Static Analysis Phases (part 2/2)

(3) Prove assertions
- Four possible outcomes:
  1. *True*, the assertion holds for all executions that reach it
  2. *False,* the assertion does not hold for all executions that reach it
  3. *Bottom,* the assertion is not reached in any execution
  4. *Top,* the assertion holds only sometimes or the analysis is imprecise

(4) Report warnings and suggest repairs
- Rank the warnings based on severity

Forwards and Backwards Analysis

Verified repairs are property specific.

Verified repairs are inferred by the process of either:

(1) Backwards *must* analysis: to specifically provide repairs involving new contracts, initializations, and guards

(2) Forwards *may* analysis: to specifically provide repairs involving off-by-one, floating point exceptions, and arithmetic overflows

Backwards analysis

Backwards analysis is treated as a function which computes the under-approximation of semantics by computing fixed points at loops.

Begin with a known failing assertion and analyze backwards until a point where the preconditions for the failing assertion do not hold.

Specifically provide repairs for failing assertions with properties involving new contracts, initializations, and guards.

Seeing how it works: Repair by contract

```
int[] ContractRepairs(int index)
{
  var length = GetALength(); // (1)
  var arr = new int[length];
  arr[index] = 9876;
  return arr;
 }
```

Assertions: $0 \leq index$ and $index < arr.Length$

Seeing how it works: Repair by contract

```
int[] ContractRepairs(int index)
{
  var length = GetALength(); // (1)
  var arr = new int[length];
  arr[index] = 9876;
  return arr;
 }
```

Assertions: $\boxed{0 \leq index}$ and $index < arr.Length$

Seeing how it works: Repair by contract

```
int[] ContractRepairs(int index)
{
  var length = GetALength(); // (1)
  var arr = new int[length];
  arr[index] = 9876;
  return arr;
 }
```

Assertions: $\boxed{0 \leq index}$ and $index < arr.Length$

- $B_{entry}(0 \leq index) = 0 \leq index$ is suggested as precondition.
- It is necessary as else underflow shall occur.
- It isn't sufficient (array in bounds isn't ensured)

Seeing how it works: Repair by contract

```
int[] ContractRepairs(int index)
{
  var length = GetALength(); // (1)
  var arr = new int[length];
  arr[index] = 9876;
  return arr;
 }
```

Assertions: $0 \leq index$ and $\boxed{index < arr.Length}$

Seeing how it works: Repair by contract

```
int[] ContractRepairs(int index)
{
  var length = GetALength(); // (1)
  var arr = new int[length];
  arr[index] = 9876;
  return arr;
 }
```

Assertions: $0 \leq index$ and $\boxed{index < arr.Length}$

- $B_{entry}(index < arr.Length) = True$ but $B_1(index < arr.Length) = index < length$ must be true for GetALength.
- Repair: Contract.Assume (index < length)

Seeing how it works: Initialization fixes (1)

```
void P(int[] a)
{


  for (var i = 0; i < a.Length; i++)
    a[i - 1] = 110;
}
```

```
void P'(int[] a)
{
  Contract.Requires(a != null);

  for (var i = 1; i < a.Length; i++)
    a[i - 1] = 110;
}
```

- Necessary condition $1 \leq i$, but $i = 0$. So suggest fix as $i = 1$.

Seeing how it works: Initialization fixes (2)

```
string GetString(string key)
{
 var str = GetString(key, null);
 if (str == null)
 {
  var args = new object[1];
  args[1] = key; // (*)
  throw new ApplicationException(args);
 }
 return str;
}
```

- Inferred necessary condition: $1 < args.Length$
- Suggested repair: new object [2]

Seeing how it works: Initialization fixes (3)

```
void ValidateOwnerDrawRegions(
  ComboBox c, Rectangle updateRegionBox)
{
  if (c == null)
  {
    var r = new Rectangle(0, 0, c.Width); // (*)
    // use r and c
  }
}
```

Not reached ever!

Seeing how it works: Initialization fixes (3)

Change to (c != null)

```
void ValidateOwnerDrawRegions(
  ComboBox c, Rectangle updateRegionBox)
{
  if (c == null)
  {
    var r = new Rectangle(0, 0, c.Width); // (*)
    // use r and c
  }
}
```

Not reached ever!

Seeing how it works: Repairing object initialize

(this.s != null) should be true always.

As else:

    x = new MyClass();
    x.Foo();

fails invariably.

```
public class MyClass
{
 private readonly SomeObj s;

 public MyClass(SomeObj s)
 {
   Contract.Requires(s != null);

   this.s = s;
 }

 public MyClass()
 {
 }

 public int Foo()
 {
   return this.s.f;
 }
 // ...
}
```

Seeing how it works: Repairing object initialize

this.s is private hence can't be made precondition of foo. It should be established during creation

```
public class MyClass
{
 private readonly SomeObj s;

 public MyClass(SomeObj s)
 {
   Contract.Requires(s != null);

   this.s = s;
 }

 public MyClass()
 {
 }

 public int Foo()
 {
   return this.s.f;
 }
 // ...
}
```

Seeing how it works: Repairing object initialize

either initialize this.s to a non-null value in MyClass() or add an object invariant to avoid the null dereference of s in Foo.

```
public class MyClass
{
 private readonly SomeObj s;

 public MyClass(SomeObj s)
 {
   Contract.Requires(s != null);

   this.s = s;
 }

 public MyClass()
 {
 }

 public int Foo()
 {
   return this.s.f;
 }
 // ...
}
```

Forwards analysis

Forwards analysis evaluates repairs using the semantic facts inferred by abstract domains.

Specifically provide repairs for failing assertions with properties involving <u>off-by-one</u>, <u>floating point exceptions</u>, and <u>arithmetic overflows</u>

Seeing how it works: Repairing off-by-one

```
string GetString(string key)
{
 var str = GetString(key, null);
 if (str == null)
 {
  var args = new object[1];
  args[1] = key; // (*)
  throw new ApplicationException(args);
 }
 return str;
}
```

Infer: $1 \leq args.Length = 1$, so suggest 0 as the new index.

Seeing how it works: Repairing floating-point

```
class FloatingPoint
{
 double d;

  [ContractInvariantMethod]
 void ObjectInvariant()
 {
  Contract.Invariant(this.d != 0.0);
 }

 public void Set(double d0)
 {
  // here d0 may have extended double precision
  if (d0 != 0.0)
    this.d = d0; // d0 can be truncated to 0.0
  }
}
```

Comparing against constants. Repair with typecast:
((double)d0 != 0.0)

Seeing how it works: Repairing arithmetic overflow

$$\frac{}{k^? \to k^!} \qquad \frac{}{\mathrm{v}^? \to \mathrm{v}^!}$$

$$\frac{ok(\mathrm{a}_1 op \mathrm{a}_2) \quad op \in \{+,-\}}{(\mathrm{a}_1^! op \mathrm{a}_2^!)^? \to (\mathrm{a}_1^! op \mathrm{a}_2^!)^!}$$

$$\frac{ok(\text{-}\mathrm{a}_2)}{((\mathrm{a}_1^! + \mathrm{a}_2^!)^? \diamond 0)^? \to (\mathrm{a}_1^! \diamond \text{-}\mathrm{a}_2^!)^!}$$

$$\frac{}{((\mathrm{a}^! + \mathrm{b}^!)^?/2^!)^? \to ((\mathrm{a}^! + ((\mathrm{b}^! - \mathrm{a}^!)^?/2^!)^!)^!)^!}$$

$$\frac{ok(\mathrm{c} - \mathrm{b})}{((\mathrm{a}^! + \mathrm{b}^!)^? \diamond \mathrm{c}^!)^? \to (\mathrm{a}^! \diamond (\mathrm{c}^! - \mathrm{b}^!)^!)^!}$$

$$\frac{ok(\mathrm{a}\text{-}\mathrm{c})}{((\mathrm{a}^! + \mathrm{b}^!)^? - \mathrm{c}^!)^? \to ((\mathrm{a}^! - \mathrm{c}^!)^! + \mathrm{b}^!)^?}$$

$$\frac{ok(\mathrm{a}\text{+}\mathrm{b})}{((\mathrm{a}^! - \mathrm{c}^!)^! + \mathrm{b}^!)^? \to ((\mathrm{a}^! + \mathrm{b}^!)^! - \mathrm{c}^!)^?}$$

$$\frac{}{((\mathrm{a}^! + \mathrm{b}^!)^? \diamond \mathrm{c}^?)^? \to (((\mathrm{a}^! + \mathrm{b}^!)^? - \mathrm{c}^?)^? \diamond 0)^?}$$

$$\frac{}{(\mathrm{a}_1^! \diamond \mathrm{a}_2^!)^? \to (\mathrm{a}_1^! \diamond \mathrm{a}_2^!)^!}$$

$$\frac{}{((\mathrm{a}_1^! - \mathrm{a}_2^!)^? \diamond 0)^? \to (\mathrm{a}_1^! \diamond \mathrm{a}_2^!)^!}$$

$$\frac{k \neq 0 \land (\mathrm{a} \neq \texttt{MinInt} \lor k \neq -1)}{(\mathrm{a}^!/k^!)^? \to (\mathrm{a}^!/k^!)^!}$$

$$\frac{}{((\mathrm{a}^! + \mathrm{b}^!)^?/2^!)^? \to ((\mathrm{b}^! + ((\mathrm{a}^! - \mathrm{b}^!)^?/2^!)^!)^!)^!}$$

$$\frac{ok(\mathrm{c} - \mathrm{a})}{((\mathrm{a}^! + \mathrm{b}^!)^? \diamond \mathrm{c}^!)^? \to (\mathrm{b}^! \diamond (\mathrm{c}^! - \mathrm{a}^!)^!)^!}$$

$$\frac{ok(\mathrm{b}\text{-}\mathrm{c})}{((\mathrm{a}^! + \mathrm{b}^!)^? - \mathrm{c}^!)^? \to (\mathrm{a}^! + (\mathrm{b}^! - \mathrm{c}^!)^!)^?}$$

$$\frac{ok(\mathrm{a}\text{+}\mathrm{b})}{(\mathrm{a}^! + (\mathrm{b}^! - \mathrm{c}^!)^?)^? \to ((\mathrm{a}^! + \mathrm{b}^!)^! - \mathrm{c}^!)^?}$$

$$\frac{}{((\mathrm{a}^! + 1^!)^? \texttt{<=} \mathrm{b}^!)^? \to (\mathrm{a}^! \texttt{<} \mathrm{b}^!)^!}$$

Forward analysis on arithmetic overflows is defined over a set of rules.

Seeing how it works: Repairing arithmetic overflow (1)

```
int BinarySearch(int[] array, int value)
{
  Contract.Requires(array != null);
  int inf = 0, sup = array.Length - 1

  while (inf <= sup)
  {
   var index = (inf + sup) /2 ; // (*)
   var mid = array[index];

   if (value == mid) return index;
   if (mid < value) inf = index + 1;
   else              sup = index - 1;
  }
  return -1;
}
```

Seeing how it works: Repairing arithmetic overflow (1)

```
int BinarySearch(int[] array, int value)
{
  Contract.Requires(array != null);
  int inf = 0, sup = array.Length - 1

  while (inf <= sup)
  {
   var index = (inf + sup) /2 ; // (*)
   var mid = array[index];

   if (value == mid) return index;
   if (mid < value) inf = index + 1;
   else                sup = index - 1;
  }
  return -1;
}
```

This can overflow!
Repair by replacing with half sums:
inf + (sup - inf) / 2

Seeing how it works: Repairing arithmetic overflow (2)

```
void ThreadSafeCopy(char* sourcePtr, char[] dest,
                    int destIndex, int count)
{
 if (count > 0)
   if ((destIndex > dest.Length)
    || ((count + destIndex) > dest.Length))
      throw new ArgumentOutOfRangeException();
   { // ... }
}
```

Fix using: count > dest.Length - destIndex

Evaluation

Library	Methods	Overall Time	Asserts	Validated	Warnings	Repairs	Time	Asserts with Repairs	%
system.Windows.forms	23,338	62:00	154,863	137,513	17,350	25,501	1:27	14,617	84.2
mscorlib	22,304	38:24	113,982	103,596	10,386	16,291	0:59	7,180	69.1
system	15,187	26:55	99,907	90,824	9,083	15,618	0:47	6,477	71.3
system.data.entity	13,884	51:31	95,092	81,223	13,869	28,648	1:21	12,906	93.0
system.core	5,953	32:02	34,156	30,456	3,700	9,591	0:27	2,862	77.3
custommarshaler	215	0:11	474	433	41	31	0:00	35	85.3
Total	**80,881**	**3:31:03**	**498,474**	**444,045**	**54,429**	**95,680**	**4:51**	**44,077**	**80.9**

Figure 12. The experimental results of verified repairs on the core .NET libraries. We report the number of methods, the overall analysis time, the number of assertions, validated assertions, warnings, the number of repairs, the time it took to infer them, the number of assertions with at least one repair and the percentage of warnings with at least one repair. Time is in minutes.

80.9% of assertion warnings reported by cccheck had at least one verified repair suggested.

Time spent generating repairs is very small relative to time spent generating warnings.

On integration with Visual Studio, the program analyzed:

- 6+ methods/second and infers 7.5 repairs/second
- 10x faster with cccheck caching

Conclusion

The authors produced a design time solution for automatic suggestion of verified program repair.

- -The program to be repaired does not need to be run (or even complete)
- -Developer does not need to provide a test suite
- -The strict definition of *verified repair* implies correctness for all suggestions
- -Repairs are local and can handle loops/infinite states
- -Can be used as an extension of static analyzers with little added overhead
- -Full functionality operates with reasonable speed and completeness

Discussion 1

What kind of bugs can we fix using this?

Discussion 2

Does it actually bridge the gap between static analyzers and dynamic analyzer?

Discussion 3

Can it repair automatically?

Discussion 4

The program requires a set of regular expressions to capture arithmetic errors. Can this be exhaustive?

Discussion 5

What else can be added to this to make it more robust and user independent?

References

Logozzo, F., & Ball, T. (2012). Modular and verified automatic program repair. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12*. doi:10.1145/2384616.2384626