Modular and Verified Automatic Program Repair

•••

Research by Francesco Logozzo and Thomas Ball
Microsoft Research, Redmond (2012)

slide author names omitted for FERPA compliance

Introduction

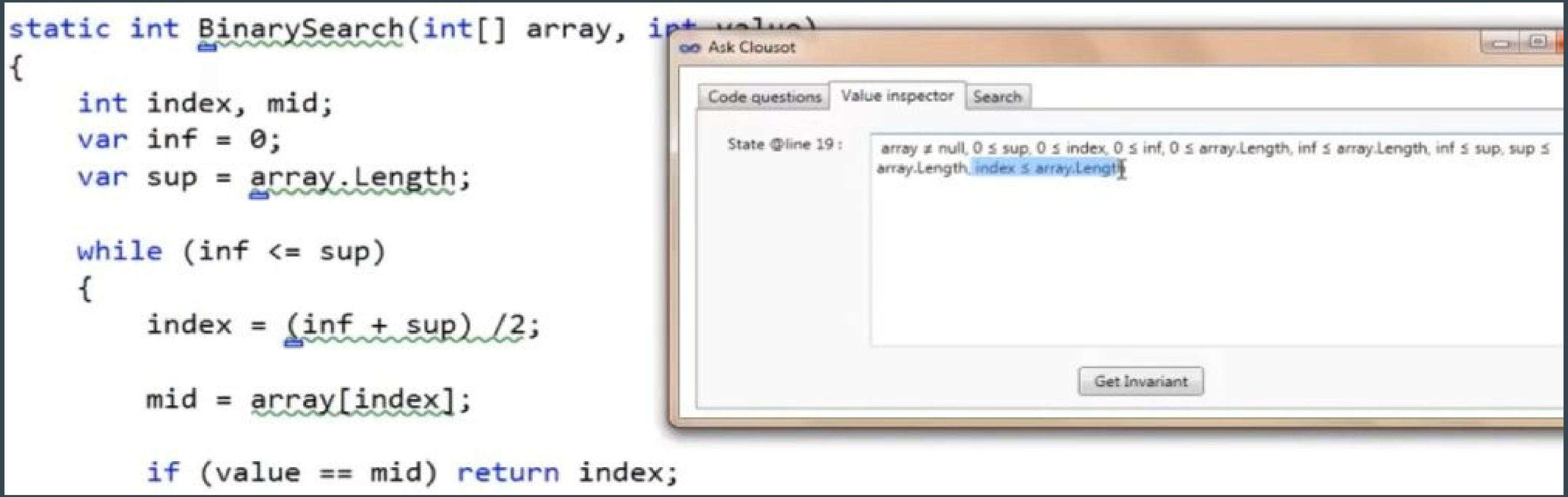
- All code is buggy!
  - What can be done about catching bugs at design time?
- Static analyzers passively provide reports or warnings
- Developers may defer bug finding and other related tasks
  - What if suggested repairs for warning were provided?

.NET CodeContracts for Visual Studio

```
static int BinarySearch(int[] array, int value)
{
    Contract.Requires(array != null);
    |
```

- Is primarily an in-line assertion library
- Provides possibility to **Design by Contract**
- Has a static checker called ccchecker

cccheck



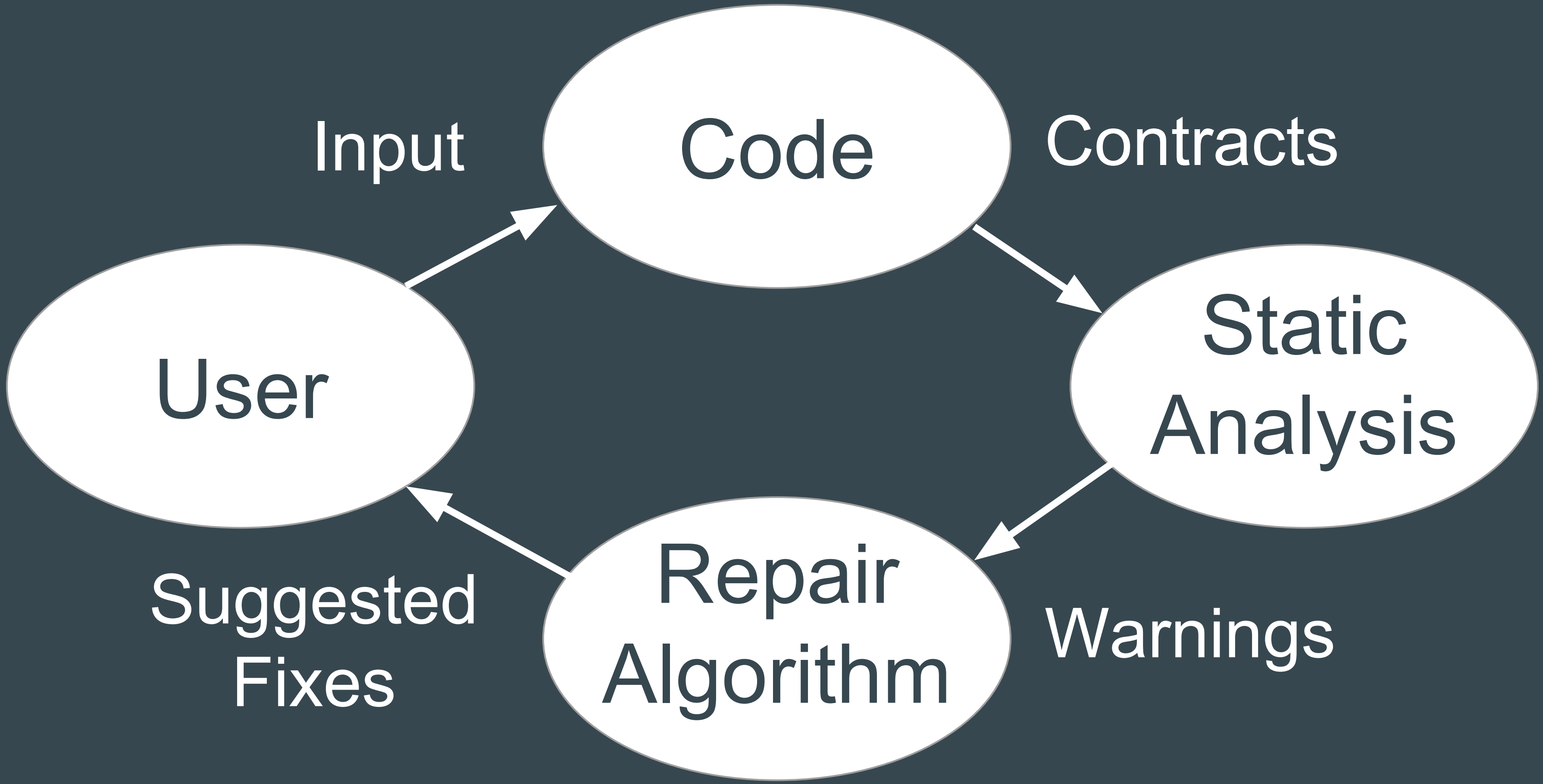
- Squiggly lines bring up warnings
- Invariant detection
- Can it do more?

What is a Modular Program Verifier?

- Decomposes verification from the level of the entire program to individual methods
- Derives semantics from inferred and given contracts:
  - Preconditions
  - Postconditions
  - Invariants
- Contracts are essential for scalability and documentation

Researcher's Vision

- Automatically suggest verified repairs for warnings
- Speculative analysis using knowledge from static analysis
- Tools knows things the developer doesn't?
  - Deep understanding of program
  - On-the-fly, without developer digging in


Input
Code
Contracts
User
Static
Analysis
Suggested
Fixes
Repair
Algorithm
Warnings

Cccheck

- Input: .NET bytecode
- Performs a series of static analyses:
    - Constructs a control flow graph
    - Checks contracts
    - Runs semantic analyses

What warnings will cccheck show you?

- Ranks warnings by severity
- Exposes common errors (other than explicit Contract Violations):
  - Buffer Overflow / Underflow
  - Null Pointer
  - Wrong Conditionals
  - Arithmetic Overflow
  - Floating Point Comparisons
  - Other well-studied, detectable errors

What Is a Code Repair?

- Some definitions tied to results of a test suite
  - Why run code?
  - Is your test suite complete?
- Verified Repair: reduces the number of bad executions in the program while preserving or increasing the number of good runs
  - Good run: Meets all specifications of the program
  - Bad run: Violates a given specification

```
static int BinarySearch(int[] array, int value)
{
    Contract.Requires(array != null);

    int index, mid;
    var inf = 0;
    var sup = array.Length - 1;

    while (inf <= sup)
    {
        index = (inf + sup) / 2;

        mid = array[index];

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

    return -1;
}
```

Clousot Code Fix

```
...
{
    index = inf + (sup - inf) / 2;
...
```

Suggesting fix for two-decades-old bug

Source: Microsoft Research (demo video)

Research Questions

- What constitutes a valid repair?
- Can suggested repairs be generated fast enough to be used in active development (i.e., in an IDE)?
- For how many of the warnings generated by cccheck can potential repairs be found?
- What kind of repairs can be produced automatically?
- How precise will the repairs be? Will they find bugs in actual code libraries?

Contributions

- Define the notion of a verified repair
  - Abstractions of trace semantics
- Propose algorithms that can be easily adapted and implemented
  - Sound, program-specific code repairs
- Show that the analysis and repair inference process is fast
  - Proposes repairs for over 80% of warnings

Typical Warnings and their Repairs

A Few Simple Examples

Repair by Contract Introduction

```
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;
}
```

- Cccheck detects a possible null-dereference and a buffer underflow in P
- It suggests the precondition a != null and initializing i to 1.

Off by One / Initialization Errors

```
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;
}
```

- Cccheck detects a buffer overflow
- Suggests either changing the index to 0 or allocating a buffer of length 2 or more.

Guards and Conditional Statements

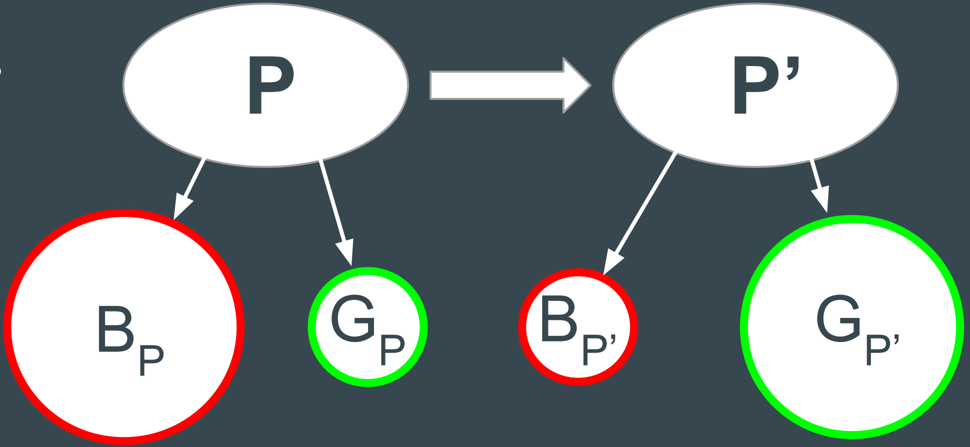
```
// Original code
if (c == null) {
        var r = new Rectangle(0, 0, c.Width);
}
// A Suggested Repair
if (c != null) {
        var r = new Rectangle(0, 0, c.Width);
}
```

- Cccheck notices that the program will crash when c is null, and that c is null in all executions (a definite error)
- Suggests flipping the guard or removing the branch altogether.

So How Does It Work?

Trace Semantics

- P is the original program, P' is the repaired program
- Σ : set of states, and $\tau_P \in \wp(\Sigma \times \Sigma)$ is a nondeterministic transition relation
- For a state $s \in \Sigma$, s(C) denotes the basic command associated with the state
- Traces are sequences of states
- B_P: the set of bad runs of P
- G_P: the set of good runs of P

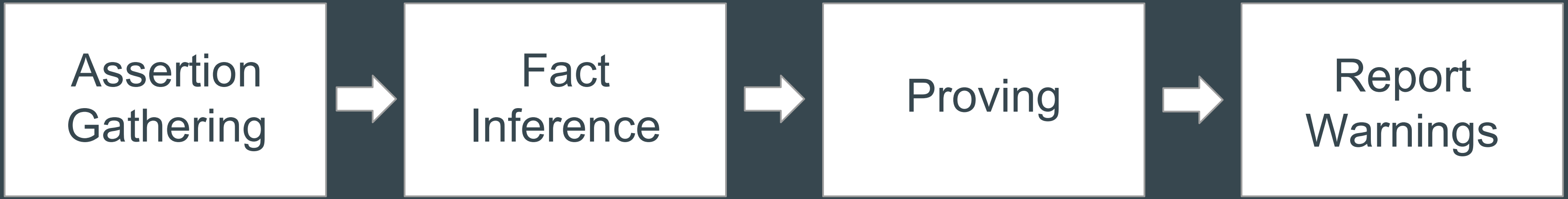


Verified Repair

- Assertion abstraction α_A removes all states but those referring to assertions
- $\delta_{P,P'}$ denotes a repair that transforms program P to program P'
- If $\alpha_A(G_P) \subseteq \alpha\delta_{P,P'} \circ \alpha_A(G_{P'})$ and $\alpha_A(B_P) \supset \alpha\delta_{P,P'} \circ \alpha_A(B_{P'})$, then we say that $\delta_{P,P'}$ is a **verified repair** for P and that P' is an improvement of P
- Denies P as an improvement, since the number of bad traces should strictly decrease
- For B_P and G_P we use the bad and good runs of P inferred by cccheck

Program Repairs in Practice

- Cccheck has four main phases:
  - Assertion Gathering
  - Fact Inference
  - Proving Assertions
  - Report Warnings and Suggest Repairs



Proving Assertions

- There are four possible outcomes:
  - True: Assertion holds for all executions reaching it
  - False: Assertion fails for all executions reaching it
  - Bottom: No execution will ever reach the assertion
  - Top: We do not know; assertion was violated sometimes or the analysis was too imprecise

✓ $\mathcal{F}$ $\top$ $\bot$

Generating Repairs

- On average, a method is analyzed in 156 ms
- Cccheck attempts to generate repairs for false and top outcomes
- Program repairs can be inferred in two ways:
    - Backwards *must* analysis
    - Forwards *may* analysis

Backwards Analysis

```
// Original code
if (c == null) {
        var r = new Rectangle(0, 0, c.Width);
}
// A Suggested Repair
if (c != null) {
        var r = new Rectangle(0, 0, c.Width);
}
```

- Starts with a failing assertion *e* and analyzes backwards until it finds a point where the preconditions of *e* might not hold
- Able to infer repairs for contracts, initializations and guards

Forwards Analysis

- Infers repairs from the abstract domains
- Works for off-by-one errors, floating point comparisons, and arithmetic overflows

So How *Well* Does It Work?

Results Breakdown

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**

- Standard libraries with validated asserts (true, bottom) and warning (false, top)
- Repairs (many to many) and asserts with at least one repair (success)

Results of IDE Integration

- Cccheck was integrated into Visual Studio
- With no caching, cccheck:
  - Analyzes 6+ methods per second
  - Infers 7.5 repairs per second
- With caching:
  - Performance was increased tenfold
- Conclusion: the approach is efficient enough to be used in an IDE

What Makes This Research Different?

- Does not rely on known failing test
- The program does not need to be run
- Property-specific repairs
- Handles loops and infinite state spaces
- More general fixes than symbolic execution
- Precise yet universal definition of code repair

Related Work

- Automated program repair field, which is very active
- Eclipse Fix-it can repair **syntactically** wrong programs
- GenProg, PAR, ARMOR, Staged Program Repair
- Speculative analysis tools like Quick Fix Scout which finds previous fixes from other code

Summary

- Using warnings generated from modular static analysis, it is possible to automatically generate repair suggestions at design time
- This process is **fast**, **consistent**, and **precise** enough to catch bugs in shipped code
- Verified repair: removes bad runs while possibly increasing good runs

Discussion Questions

- What types of bugs can verified automatic program repair fix well?
- What types of bugs might it not fix well?
- Would this type of repair suggestion be useful at design time?
- Could simple errors eventually be corrected without the input of the programmer (like AutoCorrect in MS Word)?

More Discussion Questions

- How could this system be extended in the future to find more complex and abstract errors, or to help with other common programming tasks?
- If a test suite is available, how should it be incorporated into the static analysis of cccheck?
- Can it actually make you, the developer, actually understand your program better (more deeply)?

Sources

http://research.microsoft.com/pubs/170385/res0099-logozzo.pdf

http://research.microsoft.com/en-US/projects/contracts/cccheck.pdf

http://research.microsoft.com/pubs/138696/Main.pdf

https://people.cs.umass.edu/~brun/class/2015Fall/CS521.621/lectures/20151021SpeculativeAnalysis.pdf