Modular and Verified Automatic Program Repair

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

- 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
User → Input → Code → Contracts → Static Analysis → Warnings → Repairs Algorithm → Suggested Fixes → User
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;
}
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

```csharp
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

```csharp
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
- $\Sigma$: 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 $\alpha_A$ 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
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

- 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

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

- 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/138696/Main.pdf