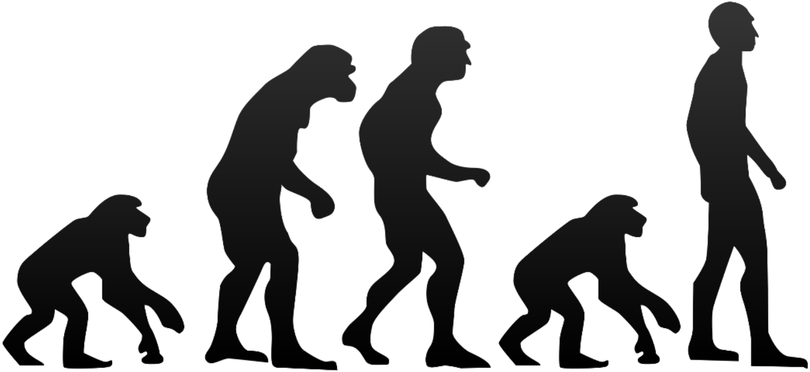
Regression Verification: Final Report

Presentation by Dennis Felsing
within the
Projektgruppe Formale Methoden der Softwareentwicklung

2014-03-28

Karlsruhe Institute of Technology

How to prevent regressions in software development?



How to prevent regressions in software development?

Formal Verification

Formally prove correctness of software
⇒ Requires formal specification

Regression Testing

Discover new bugs by testing for them
⇒ Requires test cases

How to prevent regressions in software development?

Formal Verification

Formally prove correctness of software
$\Rightarrow$ Requires formal specification

Regression Testing

Discover new bugs by testing for them
$\Rightarrow$ Requires test cases

Regression Verification

Formally prove there are no new bugs

Project Objectives

1. Develop a tool for Regression Verification for recursive programs in a simple imperative programming language
2. Evaluate how well our approaches work for different examples
3. Extend the tool to work with more programs and to be more general

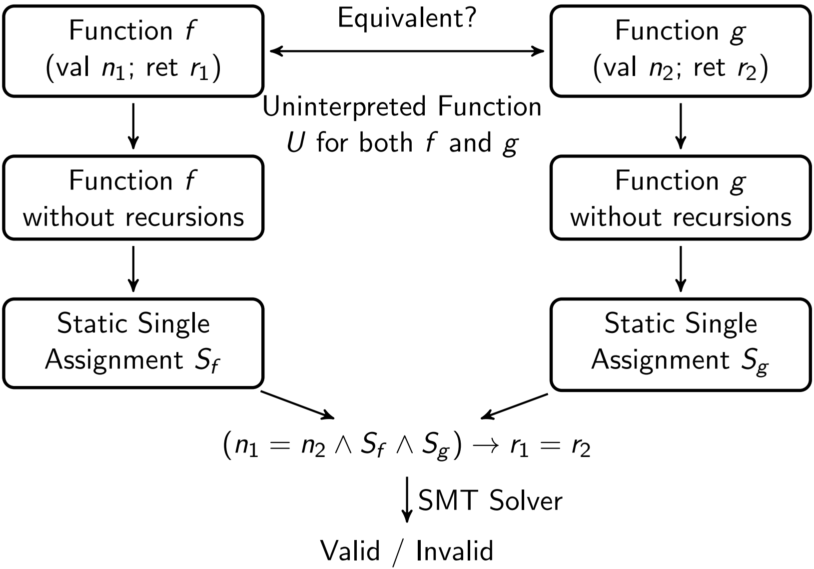
Regression Verification

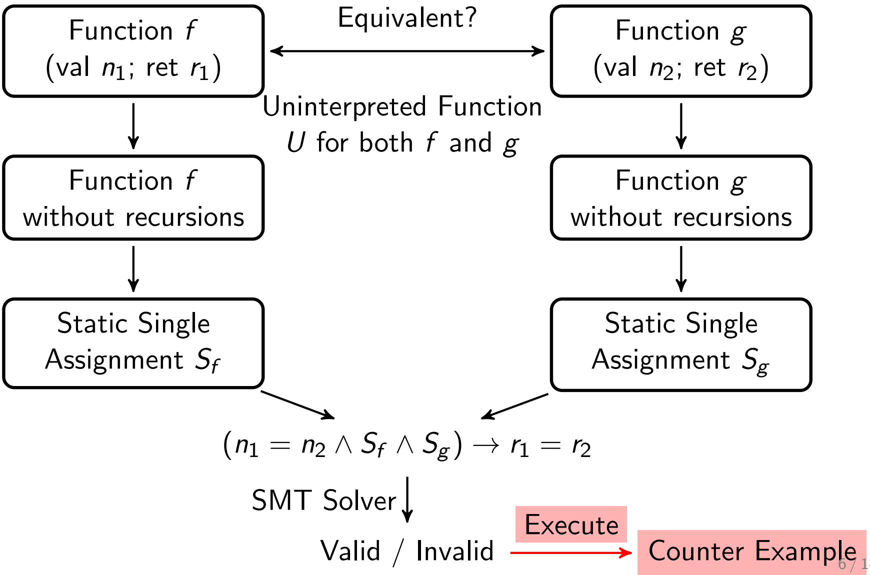
Formally prove there are no new bugs

- Goal: Proving the equivalence of two closely related programs
- No formal specification or test cases required
- Instead use old program version as reference
- Approach by Strichman & Godlin for C using CBMC
- Here: Tool for function equivalence in a simple language using SMT solvers

Function Equivalence

Existing approach by Strichman & Godlin

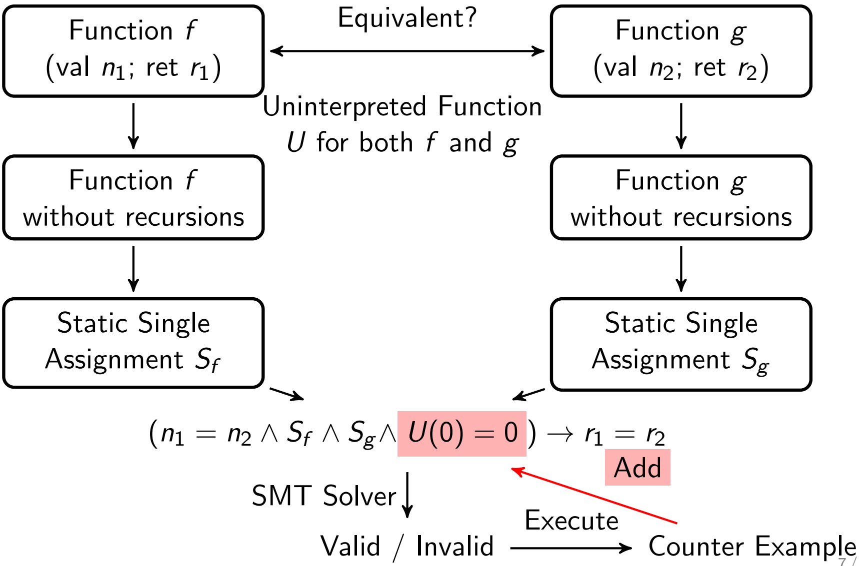




Function f
(val n_1; ret r_1)
Equivalent?
Function g
(val n_2; ret r_2)
Uninterpreted Function
U for both f and g
Function f
without recursions
Function g
without recursions
Static Single
Assignment S_f
Static Single
Assignment S_g
$(n_1 = n_2 \wedge S_f \wedge S_g) \rightarrow r_1 = r_2$
SMT Solver
Valid / Invalid
Execute
Counter Example

Extensions

Determining Corner Cases



Automatic Transition Relation Inference

Recursive approach

- So far recursions replaced by Uninterpreted Function
- Now approximate the recursive calls with a mutual Transition Relation (TR)
- Encode behaviour of both functions for every path into TR
- Assuming TR assert that functions behave equally
- Infer TR automatically using SMT solvers

Working examples

- Different paths for same input
- Inlined recursions
- Recursive versus tail recursive implementation

Automatic Transition Relation Inference

Transition Relation Approximation

$$TR_{real}(p_1, r_1, p_2, r_2) = true \Leftrightarrow r_1 = f(p_1) \wedge r_2 = g(p_2)$$

- p_1 parameter to recursive call of f
- r_1 result of recursive call of f
- r_2 parameter to recursive call of g
- p_2 result of recursive call of g

Automatic Transition Relation Inference

Transition Relation Approximation

$$TR_{real}(p_1, r_1, p_2, r_2) = true \Leftrightarrow r_1 = f(p_1) \wedge r_2 = g(p_2)$$

- p_1 parameter to recursive call of f
- r_1 result of recursive call of f
- r_2 parameter to recursive call of g
- p_2 result of recursive call of g

Approximation: $TR \supseteq TR_{real}$

Examples:

- Identical recursion step:
  $TR(p_1, r_1, p_2, r_2) = (p_1 = p_2 \rightarrow r_1 = r_2)$
- Functions off by one:
  $TR(p_1, r_1, p_2, r_2) = (p_1 = p_2 \rightarrow r_1 = r_2 + 1)$

Results are not specified, but relationship of results!

Automatic Transition Relation Inference

```
int sum1(int n) {
  if (n <= 0)
    return 0;
  int r = n + sum1(n-1);
  return r;
}
```

```
int sum2(int n, int a) {
  if (n <= 0)
    return a;
  int r = sum2(n-1, n+a);
  return r;
}
```

Automatic Transition Relation Inference

```
int sum1(int n) {
  if (n <= 0)
    return 0;
  int r = n + sum1( n-1 );
  return r;
}
```

```
int sum2(int n, int a) {
  if (n <= 0)
    return a;
  int r = sum2( n-1 , n+a );
  return r;
}
```

- Paths (two out of four possible):

$(n \leq 0) \quad \rightarrow TR(n, 0, n, a_2, a_2)$

Automatic Transition Relation Inference

```
int sum1(int n) {
  if (n <= 0)
    return 0;
  int r = n + sum1(n-1);
  return r;
}
```

```
int sum2(int n, int a) {
  if (n <= 0)
    return a;
  int r = sum2(n-1, n+a);
  return r;
}
```

- Paths (two out of four possible):

$$(n \leq 0) \rightarrow TR(\ n\ ,\ 0\ ,\ n\ ,\ a_2\ ,\ a_2\ )$$

$$(n > 0 \wedge TR(\ n-1\ ,\ r_1\ ,\ n-1\ ,\ n+a_2\ ,\ r_2\ )) \rightarrow TR(n, n+r_1, n, a_2, r_2)$$

Automatic Transition Relation Inference

```
int sum1(int n) {
  if (n <= 0)
    return 0;
  int r = n + sum1(n-1);
  return r;
}
```

```
int sum2(int n, int a) {
  if (n <= 0)
    return a;
  int r = sum2(n-1, n+a);
  return r;
}
```

- Paths (two out of four possible):

$$(n \leq 0) \rightarrow TR(\, n\,,\, 0\,,\, n\,,\, a_2\,,\, a_2\,)$$

$$(n > 0 \wedge TR(\, n-1\,,\, r_1\,,\, n-1\,,\, n+a_2\,,\, r_2\,)) \rightarrow TR(n, n+r_1, n, a_2, r_2)$$

- Show the functions behave equally:

$$\exists TR.(n_1 = n_2 \wedge a_2 = 0 \wedge TR(n_1 - 1, r_1, n_2 - 1, n_2 + a_2, r_2))$$
$$\rightarrow ite(n_1 \leq 0, 0, n_1 + r_1) = ite(n_2 \leq 0, a_2, r_2)$$

Automatic Transition Relation Inference

```
int sum1(int n) {
  if (n <= 0)
    return 0;
  int r = n + sum1( n-1 );
  return r;
}
```

```
int sum2(int n, int a) {
  if (n <= 0)
    return a;
  int r = sum2( n-1 , n+a );
  return r;
}
```

- Paths (two out of four possible):

$$(n \leq 0) \rightarrow TR(\ n\ ,\ 0\ ,\ n\ ,\ a_2\ ,\ a_2\ )$$

$$(n > 0 \wedge TR(\ n-1\ ,\ r_1\ ,\ n-1\ ,\ n+a_2\ ,\ r_2\ )) \rightarrow TR(n, n+r_1, n, a_2, r_2)$$

- Show the functions behave equally:
  $$\exists TR.(n_1 = n_2 \wedge a_2 = 0 \wedge TR(n_1-1, r_1, n_2-1, n_2+a_2, r_2)) \rightarrow ite(n_1 \leq 0, 0, n_1+r_1) = ite(n_2 \leq 0, a_2, r_2)$$
- Resulting TR predicate found by SMT solver Eldarica:
  $TR(n_1, r_1, n_2, a_2, r_2) = (n_1 = n_2 \wedge r_1 = r_2 - a_2)$

Automatic Invariant Inference

Iterative approach

- Loops instead of recursions
- Coupling invariant for both programs
- Completely automatic translation to SMT2

Working examples

- Synchronised loops
- Loosely synchronised loops
- Conditional and relational equivalence

Automatic Invariant Inference

```
int f1(int n) {
  int r = 0;
  if (n == 0) return 1;
  while (n > 0) {
    n /= 10; r++;



  }
  return r;

}
```

```
int f2(int n) {
  int r = 1;

  while (true) {
    if(n < 10) return r;
    if(n < 100) return r+1;
    if(n < 1000) return r+2;
    if(n < 10000) return r+3;
    n /= 10000;
    r += 4;
  }
}
```

Automatic Invariant Inference

```
int f1(int n) {
  int r = 0;
  if (n == 0) return 1;
  while (n > 0) {
    n /= 10; r++;
    if(n>0) {n/=10; r++;
    if(n>0) {n/=10; r++;
    if(n>0) {n/=10; r++;
  }}}}
  return r;

}
```

```
int f2(int n) {
  int r = 1;

  while (true) {
    if(n < 10) return r;
    if(n < 100) return r+1;
    if(n < 1000) return r+2;
    if(n < 10000) return r+3;
    n /= 10000;
    r += 4;
  }
}
```

Inlining preserves semantics

Automatic Invariant Inference

```
int f1(int n) {
  int r = 0;
  if (n == 0) return 1;
  while (n > 0) {
    n /= 10; r++;
    if(n>0) {n/=10; r++;
    if(n>0) {n/=10; r++;
    if(n>0) {n/=10; r++;
  }}}}
  return r;
}
```

```
int f2(int n) {
  int r = 1;
  while (true) {
    if(n < 10) return r;
    if(n < 100) return r+1;
    if(n < 1000) return r+2;
    if(n < 10000) return r+3;
    n /= 10000;
    r += 4;
  }
}
```

=

Coupling Invariant

=

Inlining preserves semantics

Automatic Invariant Inference

$$(n_1 = n_2 \wedge init_1(n_1, r_1, n_1', r_1') \wedge init_2(n_2, r_2, n_2', r_2')) \rightarrow inv(n_1', r_1', n_1', r_1', n_2', r_2', n_2', r_2')$$

Automatic Invariant Inference

$$(n_1 = n_2 \wedge init_1 \wedge init_2) \rightarrow inv$$

$$(inv \wedge guard_1 \wedge guard_2 \wedge step_1 \wedge step_2) \rightarrow inv$$

$$(inv \wedge guard_1 \wedge \neg guard_2 \wedge step_1) \rightarrow inv$$

$$(inv \wedge \neg guard_1 \wedge guard_2 \wedge step_2) \rightarrow inv$$

$$(inv \wedge \neg guard_1 \wedge \neg guard_2 \wedge post_1 \wedge post_2) \rightarrow result_1 = result_2$$

Automatic Invariant Inference

$$
\begin{aligned}
&(n_1 = n_2 \quad \land init_1 \land init_2) \rightarrow inv\\
&(inv \land guard_1 \land guard_2 \land step_1 \land step_2) \rightarrow inv\\
&(inv \land guard_1 \land \neg guard_2 \land step_1) \rightarrow inv\\
&(inv \land \neg guard_1 \land guard_2 \land step_2) \rightarrow inv\\
&(inv \land \neg guard_1 \land \neg guard_2 \land post_1 \land post_2) \rightarrow result_1 = result_2
\end{aligned}
$$

- Automatically inferred coupling loop invariant:
  (Using Eldarica again)

$$
\begin{aligned}
&(n_1 > 0 \rightarrow (n_1 = n_2 \land r_1 + 1 = r_2))\\
\land &(n_2 \leq 0 \rightarrow return_2 = r_1)\\
\land &n_1 \geq n_2
\end{aligned}
$$

Automatic Invariant Inference

$$
\begin{aligned}
&(n_1 = n_2 \quad \land init_1 \land init_2) \to inv\\
&(inv \land guard_1 \land guard_2 \land step_1 \land step_2) \to inv\\
&(inv \land guard_1 \land \neg guard_2 \land step_1) \to inv\\
&(inv \land \neg guard_1 \land guard_2 \land step_2) \to inv\\
&(inv \land \neg guard_1 \land \neg guard_2 \land post_1 \land post_2) \to result_1 = result_2
\end{aligned}
$$

- Automatically inferred coupling loop invariant:
  (Using Eldarica again)

$$
\begin{aligned}
&(n_1 > 0 \to (n_1 = n_2 \land r_1 + 1 = r_2))\\
\land&(n_2 \leq 0 \to return_2 = r_1)\\
\land& n_1 \geq n_2
\end{aligned}
$$

- Compare to loop invariant: $n = \dfrac{n_0}{10^r}$
- Coupling invariant is linear and inferable!

Conclusion

Regression Verification

- Initial approach limited to strongly coupled recursions or user feedback
- Automatic Transition Relation inference: More powerful, using recent techniques in SMT solvers like Z3 and Eldarica
- Automatic Invariant inference: Automated approach for loops

Future Work

- More challenging examples
- Real Programming Language (Java)