Regression-Verification for C code

Orna Grumberg

Orly Meir

Ofer Strichman

Technion

We propose…

- To develop a method for formally verifying the equivalence of two closely-related C programs, under certain restrictions.
- To develop a tool that applies this technique to large programs.
- This will enable developers to shift from ‘Regression Testing’ to ‘Regression verification’.

Testing and Regression Testing

- Advantage of Testing (vs. formal verification): controllable complexity. Complexity of testing is linear in the number of test cases.
- Regression Testing is the most popular automatic testing technique for general software.

Regression Testing vs. Property-based Testing

- Advantages of Regression Testing:
  - Does not require formal specification.
    - Formal spec is hard , not always possible
    - Occasionally a property is as complex as the program itself
  - Can be applied from early development stages.

Regression Testing vs. Property-based Testing

- Disadvantages of Regression Testing:
  - Does not require formal specification
    - What does correctness mean without a specification?
    - Temporal properties are very hard to check.
  - The base-case is checked ‘manually’

Limitations of Regression-Verification

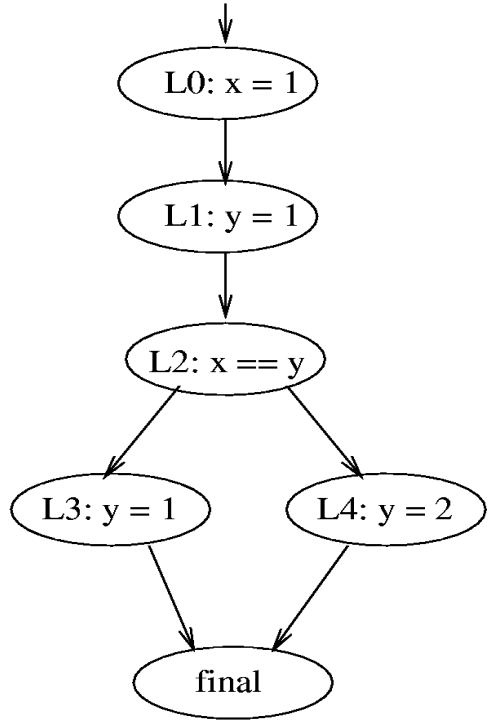
- Proving equivalence of two general programs is in general undecidable.
- Some sacrifice in completeness is required.
- Even when the problem is decidable - the complexity may prevent us from checking large programs.

Existing tools for automated verification of C

- We propose to build regression verification on top of existing tools for verification of C programs.
- Three main categories:
  - Predicate Abstraction based tools (MS-SLAM, CMU-Magic, Berkley's BLAST).
  - Model-checking of C programs with bounded resources (CMU-CBMC, IBM's Wolf).
  - Explicit state-representation (SPIN-based).

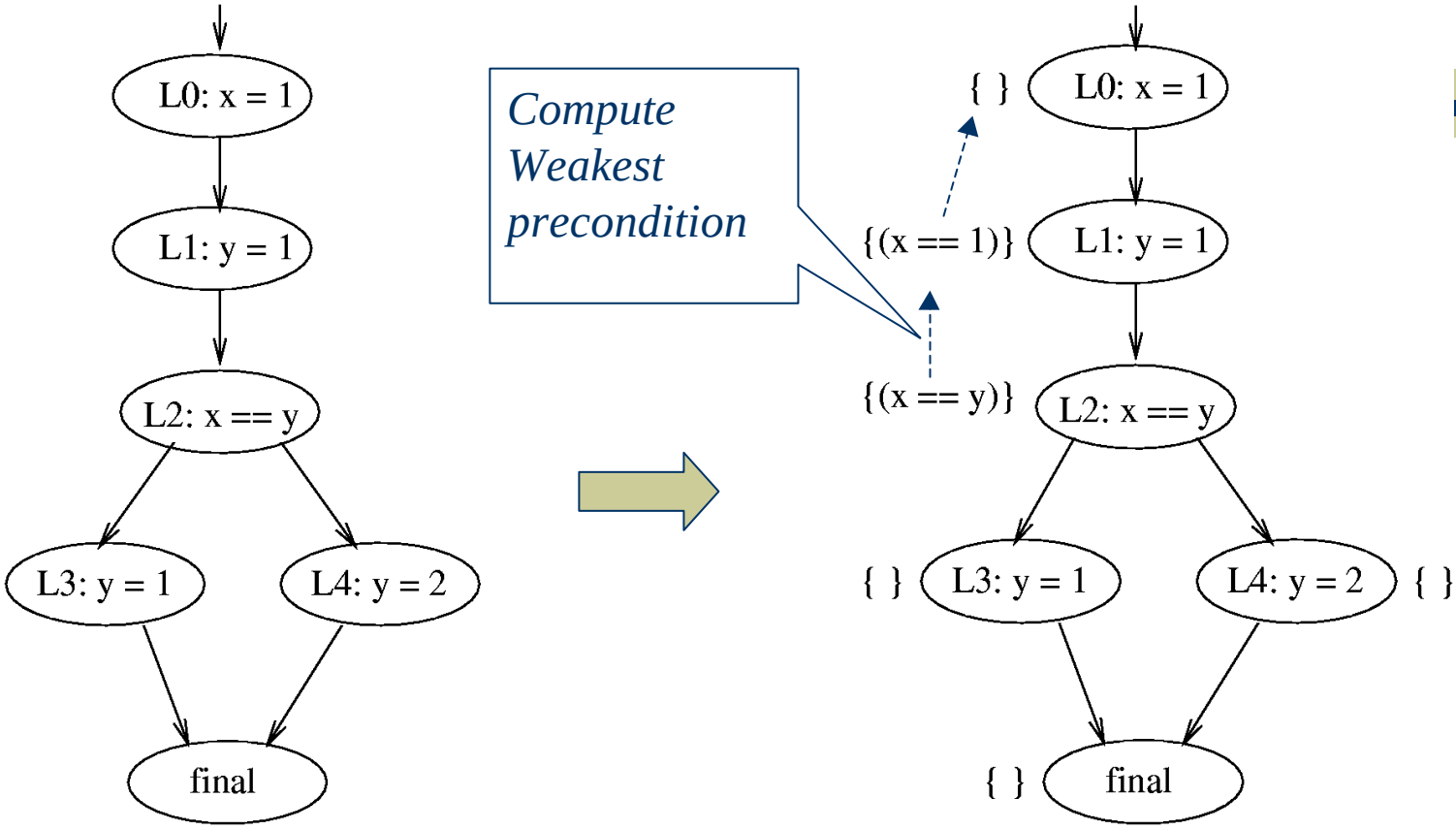
Predicate abstraction

```
int x,y
L0:    x = 1;
L1:    y = 1;
L2:    if (x == y)
L3:           y = 1;
L4:    else   y = 2;
```



Control Flow Automaton

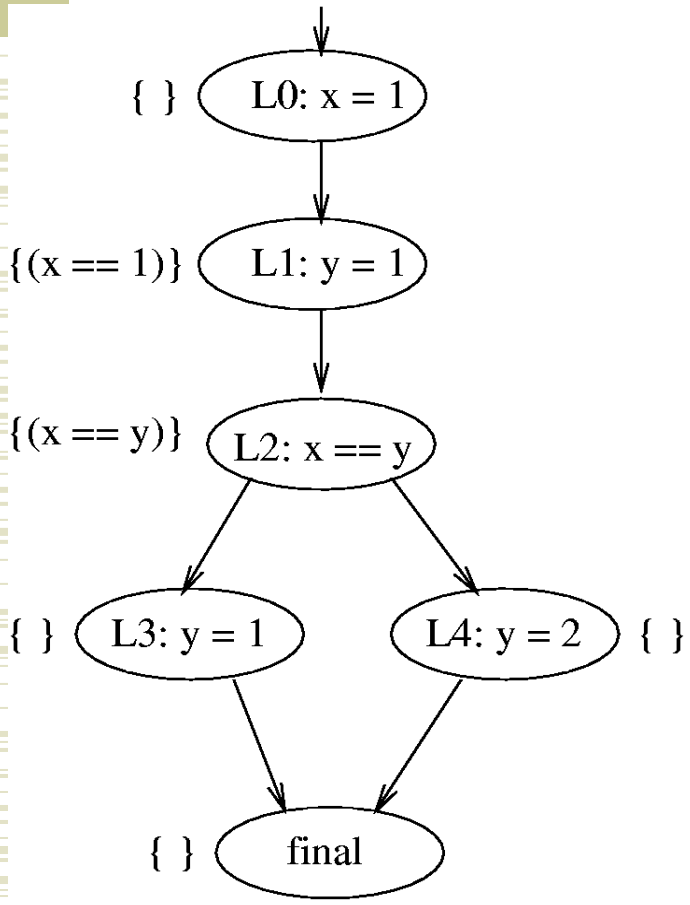
Predicate abstraction (cont'd)



Control Flow Automaton

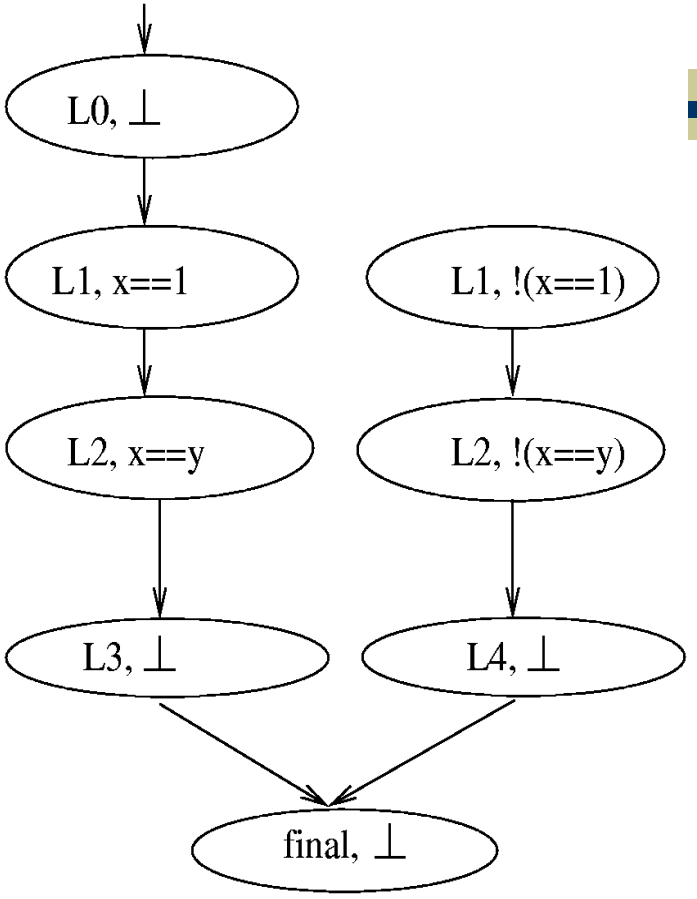
Predicate inference

Predicate abstraction (cont'd)



$$\mathcal{P} = \{(x == y)\}$$

Predicate inference



$$\mathcal{A}(\Pi, \mathcal{P})$$

Abstract model

Disadvantages of predicate-abstraction

- There are properties that cannot be proven correct in certain programs without an infinite number of predicates
- Even when a finite set of predicates is sufficient - finding it automatically is hard.
- Existing tools typically look only at the predicates appearing in the program text. This limits the number of programs that can be verified.

Using Predicate-abstraction for regression verification

Option 1: compare predicates.

- The old and new versions of the code are represented as two abstract machines, A and B.
- Let $P = \{P_1, \ldots, P_n\}$ represent the set of predicates that the user expects to be evaluated the same by both programs.

…

Using Predicate-abstraction for regression verification

Option 1: compare predicates (cont'd).

- Find the sets $a_P, b_P \in 2^P$, containing the reachable sets of valuations of P predicates in the two programs, respectively.
- If $a_P \neq b_P$ then the equivalence check fails.

Using Predicate-abstraction for regression verification

Option 2: Compare variables values in A £ B

- The old and new versions of the code are represented as two abstract machines, A and B. Consider the product C = A £ B.
- Let P = {P_1, … , P_n} represent the set of predicates that the user expects to be True in C in a given program location.

…

Using Predicate-abstraction for regression verification

Option 2: Compare variables values in A £ B (cont'd)

- Find a reachable state in C that satisfies $: (P_1 \wedge \cdots \wedge P_n)$.
- If found such state, the equivalence check fails.

Bounded resources (1): Bounded Model Checking of C

- CBMC is a tool developed by D. Kroening, that:
  - Unrolls a given ANSI-C program up to a given bound on each loop and recursion depth.
  - Translates the resulting transition relation to propositional logic, assuming the given (finite) type of each variable (e.g. an integer is represented by a 32-bit vector).
  - Adds the negation of user-defined assertions to the formula.
  - Sends the resulting formula to a SAT solver.

Bounded resources (1): Bounded Model Checking of C

- Main disadvantages:
  - On realistic programs, very restricted (due to complexity) in the unrolling bound, especially in the presence of nested loops.
  - Typically more (manual) abstractions are required.
- Main advantages:
  - In theory can prove any terminating program.
  - Supports full ANSI-C.

Bounded resources (1): CBMC and Regression Verification

- CBMC was used in the past to verify the equivalence of C and Verilog specifications.
- Proving equivalence between two C programs seems easy: simply add assertions that refer to variables in both programs.

Bounded resources (2): IBM's C verification tool

- Using the power of RuleBase
- Translates most of C to EDL
- Uses a Program Counter
- Models bounded-depth recursion with a bounded stack.
- Models dynamic memory allocation with a bounded heap.
- Automatic specifications: no infinite loops, no assert violations, no memory leaks, no access to dangling pointers, no out of bound access to arrays

It gets more interesting…

- The main challenge is to be able to prove large programs, larger than can be verified by the existing C verification tools
- There are various optimizations and decomposition rules that can be applied only when proving equivalence, but not when performing model-checking.

Using Uninterpreted Functions

- When the two C programs are close, we expect many functions to be syntactically equivalent.
- Q: How can we use this fact to prune the state space spanned by the verification tool ?
- A: With Uninterpreted Functions.

Using Uninterpreted Functions

- Every function, e.g. int f, may have:
  - Arguments $a_1 \ldots, a_n$
  - A set of global variables which it reads G_r
  - A set of global variables to which it writes G_w
  - A return value
- Replace function invocations with new variables int f, f'
- Maintain functional consistency…

Using Uninterpreted Functions

Consistency with global variables:

- *if* two functions
  - receive the same arguments, and
  - read equal global variables,
- *then*
  - their result is the same, and
  - the value of global variables to which they write is the same

$$(\bigwedge_{i=1}^{n} a_i = a'_i \wedge \bigwedge_{g_r \in G_r} g_r = g'_r) \rightarrow (f = f' \wedge \bigwedge_{g_w \in G_w} g_w = g'_w)$$

Uninterpreted functions: example

```
…
int x, a, b, global_R, global_W;
…
int f (int arg1, int arg2) {
    int local;
    local = global_R  + arg1;
    global_W = local + arg2;
    return local;
}
…
x = f(a,b);
…
```

```
…
int x', a', b', global_R', global_W';
…
int f' (int arg1', int arg2') {
    int local';
    local' = global_R' + arg1';
    global_W' = local' + arg2';
    return local';
}
…
x' = 2 * f'(a',b');
…
```

Uninterpreted functions: example

```
…
int x, a, b, global_R, global_W;
…
int f;
```

```
…
int x', a', b', global_R', global_W';
…
int f';
```

```
…
x = f;
…
```

```
…
x' = 2 * f';
…
```

Uninterpreted functions: example

```
…
int x, a, b, global_R, global_W;
…
int f;
…
x = f;
…
```

```
…
int x', a', b', global_R', global_W';
…
int f';
…
x' = 2 * f';
…
```

Add the constraint:

(global_R = global_R' Æ a = a' Æ b = b') ! (global_W = global_W' Æ f = f')

Some questions we would like to answer

- Q: What if two functions are similar but not syntactically the same ?
- Q: Once an error is found, how do we let users approve changes in an efficient way ?
- …