A Constraint-Based Approach to Verification of Programs with Floating-Point Numbers

Carlos Acosta
M.S. Student in Computer Science
Constraint Research and Reading Group
University of Texas at El Paso
Advisor: Dr. Martine Ceberio

ABSTRACT
Software plays an important role in our daily lives. There is software in our cell phones, and in our workplaces, just to mention a few examples. We want reliable software. This is specially desired in critical software such as software in airplanes, software in medical applications, and software in nuclear plants. Software Validation and Verification help us to achieve reliable software. Verification refers to proving that a system satisfies its specifications.

verification of programs is important:

- Software is around us (cell phones, workplaces, cars, …) and we want it to be reliable.
- Verification of programs with floating-points is important:
  - Critical software: medical systems, airplanes, software in nuclear plants, …

SIGNIFICANCE

- Verification of programs:
  - Software is around us (cell phones, workplaces, cars, …) and we want it to be reliable.
  - Critical software: medical systems, airplanes, software in nuclear plants, …

COMMON VERIFICATION TECHNIQUES

- Automatic Test Case Generation
    - The approach consists of automatically generating test data that will execute a selected point in the code.
    - Steps:
      1. Transform the code into Static Single Assignment form and analyze control-dependencies.
      2. Build a constraint system with this information (Kset).
      3. Solve Kset to generate test data for the selected point.

CONTRAINT-BASED APPROACHES TO VERIFICATION (1)

- Automatic Test Case Generation
    - The approach consists of automatically generating test data that will execute a selected point in the code.
    - Steps:
      1. Transform the code into Static Single Assignment form and analyze control-dependencies.
      2. Build a constraint system with this information (Kset).
      3. Solve Kset to generate test data for the selected point.

CONTRAINT-BASED APPROACHES TO VERIFICATION (2)

- Conformity of Specifications and Code
    - The approach is similar to the process of resolution:
      - We want to show that the implementation models the specification.
      - This is equivalent to:
        \[ A \rightarrow B \]
      - Steps:
        1. Translate the program into a constraint system. (A)
        2. Translate the negation of the specifications into a constraint system. (¬B)
        3. Consider the conjunction of these two constraint systems as a CSP (possibly involving guarded constraints). (A ∧ ¬B)
        - If a solution is found, the program does not meet its specification, and the solutions to the CSP are the test cases that would fail to meet the specifications.
        - If no solution is found, it means that the program meets its specification.

Second approach:

- Calculate inner approximation of A
- Solve the inner approximation of ¬A for B. This is an outer approximation of B
- Problem: We may add more false positives
- Improvement: We do not miss solutions.

PROPOSED CONSTRAINT-BASED VERIFICATION APPROACH (1)

- Our approach is based on Collavizza's and Rueher's previous work.
- Contribution: we do not generate guarded constraints in the translation of if-then-else statements.
- Steps:
  1. Translate the if-then-else statements into a CSP involving guarded constraints
  2. Transform the CSP involving guarded constraints into their equivalent by using the definition of logical implication.
  3. Transform the CSP in the form of a CNF into a DNF.
  4. Solve each clause in the DNF form as a CSP.
  5. The final solution is:
     \[ \bigcup \text{solutions of CSPs} \]

- In the third above-mentioned step, we can apply two rules to reduce the number of CSPs to be solved.

Elimination Rules

- Rule 1: Eliminate CSPs that contain
  \[ C \land \neg C \rightarrow \neg C \]
- Rule 2: Eliminate CSPs that contain
  \[ C \lor \neg C \rightarrow C \lor C \]

REFERENCES