A New Verification Tool: From LLVM Code to SMT Formulae

Milena Vujošević-Janičić
Faculty of Mathematics, University of Belgrade, Serbia
www.matf.bg.ac.rs/~milena
email: milena@matf.bg.ac.rs

Fourth Workshop on Formal and Automated Theorem Proving and Applications
University of Belgrade
Belgrade, February 4-5, 2011.

- The work is done in collaboration with:
  - prof. Dušan Tošić (University of Belgrade)
  - prof. Viktor Kunčak (EPFL, Lausanne)
  - prof. Filip Marić (University of Belgrade)

Abstract

A new approach and a corresponding tool for bug finding and for checking correctness conditions is going to be presented. The system works over LLVM code so it can be used for analysis of programs in several programming languages. The approach combines symbolic execution, SAT encoding of program's behavior and some features of bounded model checking. Namely, single blocks of the code are modeled by first-order logic formulae constructed by symbolic execution while relationships between blocks are modeled by propositional formulae. Formulae that describe program's behavior are combined with correctness conditions for individual commands to produce correctness conditions of the program to be verified. These conditions are passed to an SMT solver covering a suitable combination of theories. Currently, there is support for the following SMT solvers: Boolector, MathSAT, Yices, and Z3.

Overview of the talk

- Motivation and a short overview of the system
- Background
- Modeling correctness properties
- Implementation and preliminary results
- Conclusions and future work

- Motivation and a short overview of the system
- Background
- Modeling correctness properties
- Implementation and preliminary results
- Conclusions and future work

Motivation

- Verification of software and automated bug finding are one of the greatest challenges in computer science
- Software bugs cost the world economy hundreds of billions of dollars annually
- There are many existing approaches, but new ideas and tools are still welcome

A Short Overview of the System

- The system works over LLVM code so it can be used for analysis of programs in several programming languages
- It combines symbolic execution, SAT encoding of program's behavior and some features of bounded model checking
- Correctness conditions are generated and then passed to a SMT solver covering a suitable combination of theories

- Motivation and a short overview of the system
- Background
- Modeling correctness properties
- Implementation and preliminary results
- Conclusions and future work

Background: LLVM - Low Level Virtual Machine

- LLVM is a rich compiler framework, with front-ends for C and C++; front-ends for Java and Scheme are in development
- LLVM libraries provide a modern source- and target-independent optimizer
- LLVM object code uses simple RISC-like instructions and provides language-independent type information and data-flow information about operands
- Each program function consists of blocks of instructions: block can be entered only at its entry point, and left only through its last command

Background: Bounded Model Checking

- *Model Checking* is a technique for automatically verifying correctness properties of finite state systems, typically hardware or software systems

- *Bounded Model Checking* checks properties of the system which is underapproximated by loop unrolling (loops are unrolled a fixed number of times)

- Bounded Model Checking typically involves encoding the restricted model as an instance of SAT or as a binary decision diagram (BDD)

Background: Symbolic Execution

- Symbolic execution refers to the analysis of programs by tracking symbolic values rather than actual values; it is used to reason about all the inputs that take the same path through a program

- Symbolic execution is typically used in conjunction with an automated theorem prover, therefore there are limitations on the classes of constraints they can represent and solve

- The main problem in symbolic execution is path explosion and path coverage, therefore its main focus is finding bugs in software, rather than demonstrating program correctness

Background: SAT/SMT

- SAT is a problem of deciding if a propositional formulae in conjunctive-normal form is satisfiable

- The *Satisfiability Modulo Theory* (SMT) problem is a decision problem for satisfiability of formulas with respect to combinations of theories. Examples:
  - the theory of linear arithmetic (LA)
  - the theory of bit vector arithmetic (BVA)
  - the theory of uninterpreted functions with equality (EUF)
  - the theory of arrays (ARRAYS)

- Motivation and a short overview of the system
- Background
- Modeling correctness properties
- Implementation and preliminary results
- Conclusions and future work

Modeling correctness properties

- Dealing with variables, data types and instructions
- Modeling control flow and interprocedural analysis
- Constructing correctness conditions
- Translating correctness conditions to SMT formula

Dealing with Variables, Data types and Instructions

- We need to model
  - Variables and simple data types
  - Pointers and complex data types (buffers and structures)
  - Dereferencing pointers
  - Global variables
  - Function calls
- Each instruction transforms the store of a program and may add some constraints over variables (the store is a mapping from variables to values from their domains)

Dealing with Variables, Data types and Instructions

- For a command of a form $l = r$, where r is a variable or a operation over variables, the value of l in the store is replaced by the value of r in the current store

- Pointers are treated as simple variables; also, functions $left(p)$ and $right(p)$ are introduced to keep track of numbers of bytes reserved for the pointer p on its left and its right side

- Buffers and structures are treated uniquely as sequences of bytes accessible by a pointer and an offset

Dealing with Variables, Data types and Instructions

- Accessing memory via pointers is modeled by the theory of arrays. This theory provides functions
  - *read* — for reading a value at a certain index in the array
  - *write* — for storing a value at a certain index in the array
- The array representing memory is kept in the store
- Global variables and variables that are referenced by address operator are tracked through the memory array, others are tracked through their slots in the store

Dealing with Variables, Data types and Instructions

- Function calls are modeled according to the available information about the function:
  - If a contract of a function is available, then the current store is updated and additional constraints are added according to the contract
  - If a contract of the function is not available, while the definition of function is, then all correctness conditions from the called function are checked in this context and the constructed postcondition of the function is inserted
  - If neither a contract nor the definition of function are available, then the memory array is set to a new (fresh) variable

Modeling Control Flow and Interprocedural Analysis

- Instructions belong to blocks, there are no branching and no loops in blocks

- Block summary, $Transformation(b)$, describes the way in which a block b transforms the store of the program; it is constructed by symbolic execution

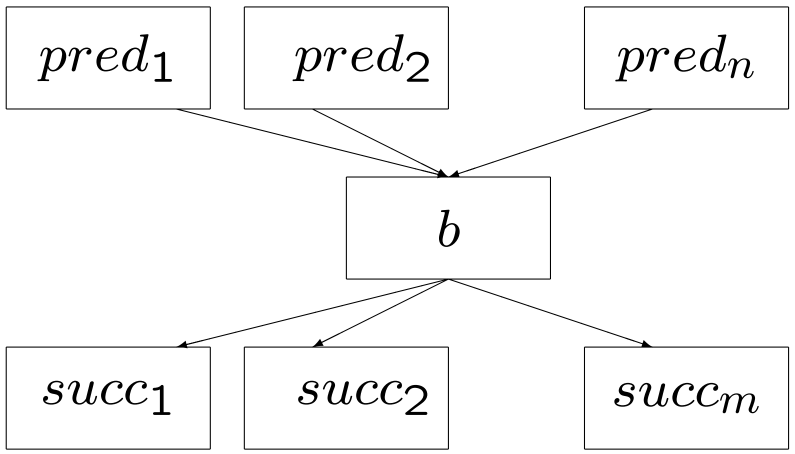
$$Transformation(b) = \bigwedge_{v \in V} (e(b, v) = e_v) \bigwedge AdditionalConstraints(b)$$

- Values of the variables at the exit point are given in terms of variables at the entry point

Modeling Control Flow and Interprocedural Analysis

- Links between blocks are modeled by propositional variables
- Postcondition of a block contains control flow information and is defined as

$$Postcondition(b) = EntryCond(b) \wedge Transformation(b) \wedge ExitCond(b)$$



Modeling Control Flow and Interprocedural Analysis

$$\begin{aligned}
Postcondition(b) &= EntryCond(b) \ \wedge \ Transformation(b) \ \wedge \ ExitCond(b) \\
EntryCond(b) &= activating(b) \wedge initialize(b) \\
Transformation(b) &= \bigwedge_{v \in V} (e(b,v) = e_v) \bigwedge AdditionalConstraints(b) \\
ExitCond(b) &= jump(b) \wedge leaving(b) \\
activating(b) &= \left(\bigvee_{pred \in Predcesors} transition(pred, b) \right) \Leftrightarrow active(b) \\
initialize(b) &= \bigwedge_{pred \in Predcesors} \left(transition(pred, b) \Rightarrow \bigwedge_{v \in V_f} e(pred, v) = s(b, v) \right) \\
jump(b) &= \bigwedge_{succ_i \in Successors} ((active(b) \wedge e(b, c_i)) \Leftrightarrow transition(b, succ_i)) \\
leaving(b) &= active(b) \Leftrightarrow \left(\bigvee_{succ \in Successors} transition(b, succ) \right)
\end{aligned}$$

Modeling Control Flow and Interprocedural Analysis

- Loops are unrolled, there are two possibilities
  - Underapproximation of loops (loops are unrolled fixed number of times like in bounded model checking)
  - Overapproximation of loops (unrolled code simulates first m and last n entries to the loop)

Modeling Control Flow and Interprocedural Analysis

- LLVM assures that there are a unique entry and a unique leaving point of each function
- The postcondition of a function is constructed as a conjunction of postconditions of the function blocks
- For each function call, formula representing postcondition of a function is added to additional constraints
- Recursive function calls are not handled yet

Constructing Correctness Conditions

- Correctness conditions are of the form $A \Rightarrow B$

- A is a formula describing context (empty context, block context, function context, wider context)

- B is a formula describing (in)correctness condition of an instruction:

  - it can be given by a bug definition — division by zero, buffer overflow, dereferencing null pointers

  - it can be given by an annotation

Constructing Corectness Conditions

- Example:
  - `a = b/5` — the command can be proved safe by using an empty context
  - `a = b/c` — if the variable c is assigned a concrete value in the current block, then the command can be proved safe/flawed using a block context
  - `a = b/c` — if the variable c is not assigned any concrete value in the current block, then, in order to prove that the command is safe/flawed, postconditions of predecessors of the block have to be included into the context
  - ...

Translating correctness conditions to SMT formula

- Select a suitable SMT theory (LA, BVA, EUF, ARRAYS)
- Translate program data-types to SMT types
- Use EUF or ackermannization
- Select solver
- Take advantage of incremental mode of solver

- Motivation and a short overview of the system
- Background
- Modeling correctness properties
- Implementation and preliminary results
- Conclusions and future work

Implementation and preliminary results

- The tool LAV is implemented in C++, $\approx$ 330kB, 10 000 lines of source code plus the code based on Filip Marić's code for shared expressions and for APIs for SMT solvers

- For efficiency, many aspects of the general theoretical model are optimized for number of special cases

- There is support for the following SMT solvers: Boolector (BVA and ARRAYS), Yices and MathSAT (LA, BVA, EUF) and Z3 (LA, BVA, EUF, ARRAYS)

Preliminary results

- Tested on several benchmark sets
- Applicable on a wide range of input programs
- Very precise for many types of problems
- Concerning efficiency, comparable (based still on a small set of inputs) to Klee and ESBMC (state of the art tools based on symbolic execution and on bounded model checking)

- Motivation and a short overview of the system
- Background
- Modeling correctness properties
- Implementation and preliminary results
- Conclusions and future work

Conclusions and future work

- A new verification approach and a corresponding tool presented

- Still under development

- Make deep evaluation and detailed comparison with other tools

- Make LAV open-source

- Further improvement of efficiency

Conclusions and future work

- Explore different variations of the current modeling
- Improve interprocedural analysis
- Model recursion and make handling loops more sophisticated
- Add new features, like unit tests generation
- Take advantage of LLVM code optimizations

Thank you for your attention!