Modelling Program Behaviour within Software Verification Tool LAV

Milena Vujošević Janičić

Faculty of Mathematics
University of Belgrade
Serbia

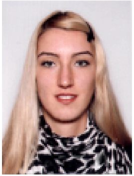
TTT, Paris, January 2017.

Joint work with

Viktor Kuncak
EPFL Switzerland

Dušan Tošić
University of Belgrade, Serbia

Filip Marić
University of Belgrade, Serbia



Branislava Živković
University of Belgrade, Serbia

Overview of the talk

Modelling Program Behaviour within LAV

Scope and aims

Modelling program behaviour

- One of the first steps for using logical reasoning for software verification
- We describe the model used and the way the program semantics is treated in our software verification tool LAV.

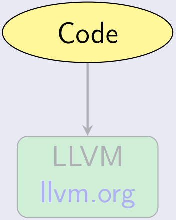
Scope and aims

LAV — http://argo.matf.bg.ac.rs/?content=lav

- LAV — a **lion** in Serbian (**L**LVM **A**utomated **V**erifier)
- Proving user given assertions and a bug finding tool:
  - division by zero
  - buffer overflows
  - null pointer dereferencing
- Primarily aimed for C programs
- Implemented in C++, publicly available and open source
- LAV combines symbolic execution, SAT encoding of program's control-flow, bounded model checking

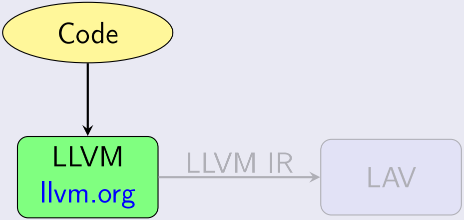
LAV and External Systems

LLVM, LAV and SMT solvers



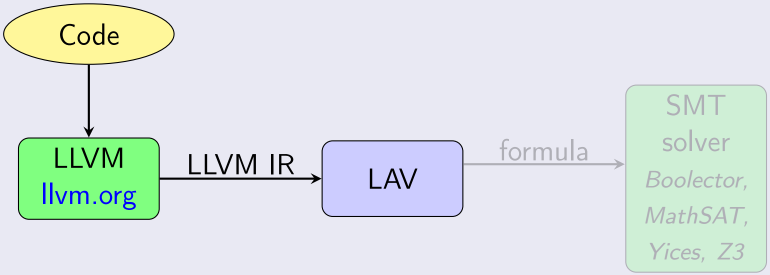
LAV and External Systems

LLVM, LAV and SMT solvers



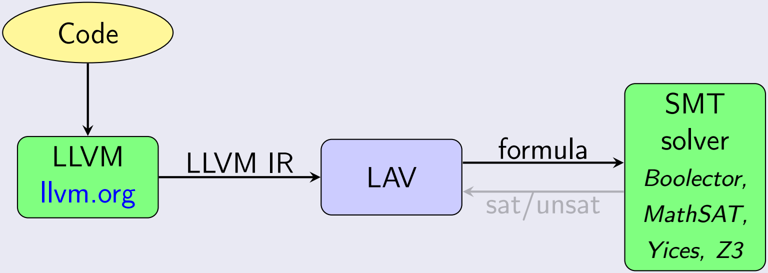
LAV and External Systems

LLVM, LAV and SMT solvers



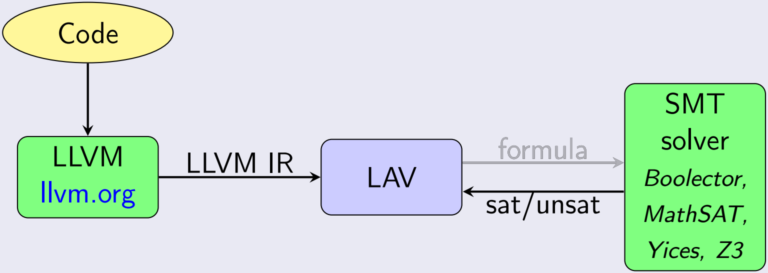
LAV and External Systems

LLVM, LAV and SMT solvers



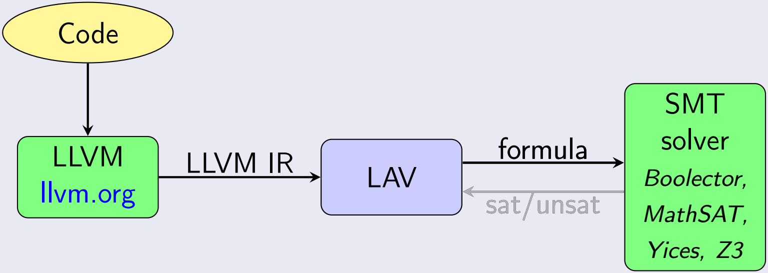
LAV and External Systems

LLVM, LAV and SMT solvers



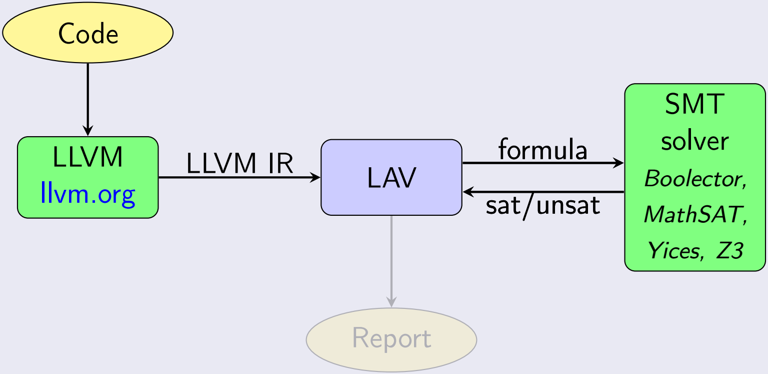
LAV and External Systems

LLVM, LAV and SMT solvers



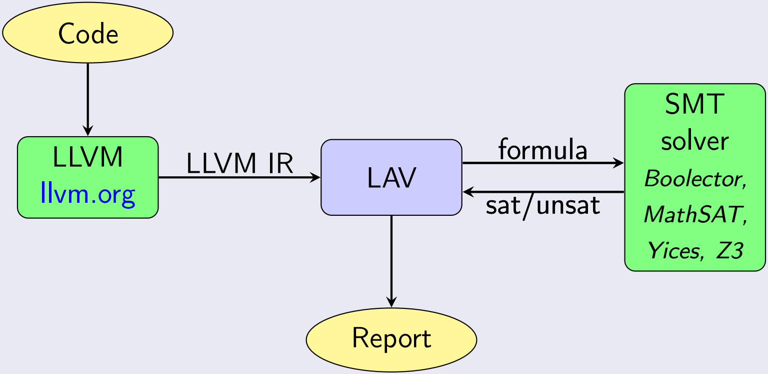
LAV and External Systems

LLVM, LAV and SMT solvers



LAV and External Systems

LLVM, LAV and SMT solvers



Symbolic execution

Block summary

- LLVM IR — blocks of code with no internal branching or loops
- LAV performs symbolic execution to obtain block summaries: FOL formulas describing each block

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

where V is a set of variables and e_v is the value of v at the end of the block, $e(b, v)$, expressed in terms of initial values (values at the starting point of the block)

- *AdditionalConstraints* keep track of some important constraints for variables

Symbolic execution

Pointers and memory

- Flat memory model, accessing memory via pointers — the theory of arrays:
  - *store* — function for storing a value at a certain index
  - *select* — function for reading a value at a certain index
  - Axioms

$$\forall a\ \forall i\ \forall v \qquad (select(store(a,i,v),i) = v)$$
$$\forall a\ \forall i\ \forall j\ \forall v \qquad (i \neq j \Rightarrow select(store(a,i,v),j) = select(a,j))$$

Symbolic execution

Buffers, Structures and Unions

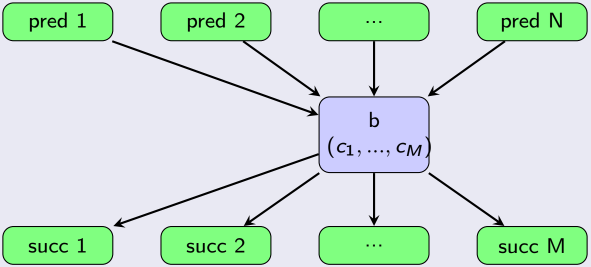
- Buffers — sequences of memory locations allocated statically or dynamically and accessible by a pointer p and an offset n.
- Uninterpreted functions *left* and *right* keep track of the number of bytes reserved for a pointer
- Axioms:

$$\forall p \; \forall n \qquad left(p+n) = left(p) - n$$
$$\forall p \; \forall n \qquad right(p+n) = right(p) - n$$

- For efficiency reasons, only relevant instances of these axioms are added to the set of additional constraints attached to the block.

Control Flow

Blocks of code



SAT encoding

- Propositional variables encode transitions between blocks
- Propositional variables are used to reconstruct a program path from the model generated by a solver

FOL encoding

Description of a block: block's summary ∧ control flow information

$$
\begin{aligned}
Descripton(b) &= EntryCond(b) \ \wedge \ Transformation(b) \ \wedge \ ExitCond(b) \\
EntryCond(b) &= activating(b) \wedge initialize(b) \\
Transformation(b) &= \bigwedge_{v \in V} (e_b(v) = e_v) \bigwedge AdditionalConstraints(b) \\
ExitCond(b) &= jump(b) \wedge leaving(b)
\end{aligned}
$$

Descriptions are used for constructing compound correctness/incorrectness conditions of individual instructions

FOL encoding

Entry condition

activating(b): There was a transition from a predecessor block to the block b iff the block b was active:

$$\left(\bigvee_{pred \in \mathcal{P}} transition(pred, b)\right) \Leftrightarrow active(b)$$

initialize(b): If the block b is reached from the block *pred*, then the initial values of variables within the block b will be the values of the variables at the leaving point of *pred*:

$$\bigwedge_{pred \in \mathcal{P}} \left(transition(pred, b) \Rightarrow \bigwedge_{v \in V_f} e(pred, v) = s(b, v)\right)$$

FOL encoding

Exit conditions

jump(b): If the block b was active and if a leaving condition c_i of the block b was met, then the control was passed to the block $succ_i$, and vice versa:

$$\bigwedge_{succ_i \in \mathcal{S}} ((active(b) \wedge e(b, c_i)) \Leftrightarrow transition(b, succ_i))$$

leaving(b): The block b was active iff it led to some other block (or to exit of the function):

$$active(b) \Leftrightarrow \bigvee_{succ \in \mathcal{S}} transition(b, succ)$$

Control Flow

Loops

- Loops are eliminated:
  - Overapproximation: simulation of the first n and the last m entries to the loop
  - Underapproximation: loops are unrolled

Function calls

Case 1: A contract available

Case 2: A definition available:

- The postcondition ψ of the called function is conjunction of descriptions of its blocks
- ψ is *inlined* into caller's summary

Case 3: Nothing available: the memory is set to a new fresh variable

Correctness conditions

Correctness conditions

- $C \Rightarrow safe(c)$ — correctness condition
  $C \Rightarrow \neg safe(c)$ — incorrectness condition
- C — context C defines command's neighbourhood that is taken into consideration
- safe(c) — safety property of a command c given by a bug definition or by an assertion

Correctness conditions

Types of commands: Safe, Flawed, Unreachable and Unsafe

- $\models C \Rightarrow safe(c)$ — the command c is safe in the context C. *It is also safe in all wider contexts (if it is reachable).*
- $\models C \Rightarrow \neg safe(c)$ — the command c is flawed in the context C. *It is also flawed in all wider contexts (if it is reachable)*
- $\models C \Rightarrow safe(c)$ and $\models C \Rightarrow \neg safe(c)$ — the command c is unreachable. *It is also unreachable in all wider contexts.*
- $\not\models C \Rightarrow safe(c)$ and $\not\models C \Rightarrow \neg safe(c)$ — the command c is unsafe in the context C. *In some wider context it may change its status.*

Correctness conditions

Types of commands: Safe, Flawed, Unreachable and Unsafe

- $\models C \Rightarrow safe(c)$ — the command c is safe in the context C. *It is also safe in all wider contexts (if it is reachable).*
- $\models C \Rightarrow \neg safe(c)$ — the command c is flawed in the context C. *It is also flawed in all wider contexts (if it is reachable)*
- $\models C \Rightarrow safe(c)$ and $\models C \Rightarrow \neg safe(c)$ — the command c is unreachable. *It is also unreachable in all wider contexts.*
- $\not\models C \Rightarrow safe(c)$ and $\not\models C \Rightarrow \neg safe(c)$ — the command c is unsafe in the context C. *In some wider context it may change its status.*

Correctness conditions

Contexts

- Checking status in wider contexts usually takes more time
- LAV: empty context —> block context —> function context —> other wider contexts
- Wider contexts are considered only for unsafe commands
- Different contexts give room for different kind of parallelisation (ongiong work)

Transforming a Code Model to a SMT Goal

Code model

- The (quantifier-free) formula that models a program code typically uses:
  - bit-vector arithmetic (or linear arithmetic),
  - theory of uninterpreted functions,
  - the theory or arrays (optionally)
- There are several SMT solvers that provide support for such combinations of theories.

Optimizations

Some optimizations

- Only one description is constructed for consecutive blocks
- Rewriting is applied for simplifying expressions in formulas
- Unchanged values of variables are monitored and propagated through the blocks
- Selective usage of information in different contexts
- Incremental usage of SMT solvers
- Reduction of the number of solver calls

Future work

Optimisations are not formally described and should be formally justified.

Related tools

Comparison to related tools

- Related tools are based on symbolic execution and model checking
- **CBMC** (`http://www.cprover.org/cbmc/`), **LLBMC** (`http://llbmc.org/`), **ESBMC** (`http://www.esbmc.org/`), **Klee** (`https://klee.github.io/`)
- Comparison was done on different benchmarks, LAV gave good results
- Details in: *M.V. Janicic, V. Kuncak "Development and Evaluation of LAV: an SMT-Based Error Finding Platform" (VSTTE '12)*

Applications in Education

Applications in Education

- Safety-critical computer programs vs students' programs
- Software verification can add to the quality of automated grading
- Details in: *M.V. Janičić, M. Nikolić, D. Tošić, V. Kuncak, "Software Verification and Graph Symilarity for Automated Evaluation of Students' Programs", Information and Software Technology, Elsevier, 2013.*

Applications in Education

Regression verification

- Functional equivalence of similar programs (student's and teacher's solution)
- Partial equivalence and k-equivalence
- Advantages and challenges
  - Higher level of reliability
  - No need for explicit specification
  - Undecidability
  - Nontrivial transformations of programs are necessary

Applications in Education

Regression verification

- Developing set of tools for necessary program transformations
- Using LAV for proving partial functional equivalence (methods described in *B. Godlin, O. Strichman "Regression verification: proving the equivalence of similar programs", (2013) Software Testing, Verification & Reliability. John Wiley & Sons.*) and for proving k equivalence.
- Details in: *M. V. Janičić and F. Marić. Regression Verification for Automated Evaluation of Students Programs, 2016. Submitted.*
- We are interested in developing new methods

Parallelisation in LAV

Motivation

- Take advantage of both hardware properties and characteristics of software verification conditions
- Different contexts give room for different kind of parallelisation
- BMC — one compound formula describing program execution, does not scale well
- Simple example

```
int f(int a, int b, int c, int d) {
 a = (b<<3)*((c>>2)/3);
 b = (a<<3)*((c>>2)/3);
 c = (b<<3)*((a>>2)/3); //3 commands simulating complex calculations
 a = b/c + c/a + a/b;
 b = a/d;               //4 divisions, 4 checks for division by zero
 return b;
}
```

Experiments

no lines	LAV	CBMC
14	0.08	0.64
15	0.08	0.99
16	0.09	1.17
17	0.10	1.34
18	0.09	2.44
19	0.09	3.16
20	0.11	4.06
21	0.10	18.63
22	0.14	27.20
23	0.11	22.56
24	0.11	48.25
25	0.12	79.45
26	0.14	108.93
27	0.13	215.31
28	0.17	$\nearrow$
29	0.13	$\nearrow$
30	0.13	$\nearrow$
60	0.23	$\nearrow$

Justification for the previous intuition: already 28 commands time out for CBMC (state of the art BMC), while parallelisation of block context within LAV gives results that are scaling well. Results are given in seconds.

Parallelisation in LAV

Parallelisation of functions

- Programs consist of functions — parallelisation may be naturally done by verifying functions in parallel
- There are similar examples where this parallelisation may significantly speed-up verification time

Ongoing and future work

- We have very promising experimental results, but need formal justification that these parallelisations keep semantics and produce valid results.
- We also need to formally describe types of commands.

Ongoing and future work

We hope that firmer theoretical grounds would lead us to new insights and further improvements of the tool.

Thank you!