Modelling Program Behaviour within Software Verification Tool LAV

Milena Vujošević Janićić

Faculty of Mathematics
University of Belgrade
Serbia

Overview of the system LAV
Ongoing and future work

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

Modelling Program Behaviour within LAV
Modelling Program Behaviour within LAV

1. Overview of the system LAV
   - External Systems
   - Symbolic execution and SAT encoding
   - Correctness conditions
   - Optimizations

2. Ongoing and future work
   - Evaluation
   - Applications
   - Functional correctness
   - Parallelisation
Overview of the system LAV

Ongoing and future work
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 (**LLVM Automated Verifier**)
- 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

- Code
- LLVM
  - llvm.org
- SMT solvers:
  - Boolector
  - MathSAT
  - Yices
  - Z3
Overview of the system LAV
Ongoing and future work

LAV and External Systems

LLVM, LAV and SMT solvers

Code

LLVM
llvm.org

LLVM IR

LAV

External Systems
Symbolic execution and SAT encoding
Correctness conditions
Optimizations

LLVM, LLVM IR, SMT, Boolector, MathSAT, Yices, Z3
LAV and External Systems

**LLVM, LAV and SMT solvers**

- **Code**
  - LLVM
    - llvm.org
  - LLVM IR
  - LAV

**SMT solver**
- Boolector,
- MathSAT,
- Yices,
- Z3
Overview of the system LAV

Ongoing and future work

External Systems
Symbolic execution and SAT encoding
Correctness conditions
Optimizations

LAV and External Systems

LLVM, LAV and SMT solvers

LLVM, llvm.org

LLVM IR

LAV

SMT solver
Boolector, MathSAT, Yices, Z3

Code

LLVM IR

formula

sat/unsat
Overview of the system LAV
Ongoing and future work
External Systems
Symbolic execution and SAT encoding
Correctness conditions
Optimizations

LAV and External Systems

LLVM, LAV and SMT solvers

Code

LLVM
llvm.org

LLVM IR

LAV

SMT solver

Boolector,
MathSAT,
Yices, Z3

sat/unsat

formula
Overview of the system LAV
Ongoing and future work

External Systems
Symbolic execution and SAT encoding
Correctness conditions
Optimizations

LAV and External Systems

LLVM, LAV and SMT solvers

- Code
- LLVM: llvm.org
- LLVM IR
- LAV
  - formula
- SMT solver: Boolector, MathSAT, Yices, Z3
  - sat/unsat
Overview of the system LAV
Ongoing and future work

External Systems
Symbolic execution and SAT encoding
Correctness conditions
Optimizations

LAV and External Systems

LLVM, LAV and SMT solvers

- Code
- LLVM
  llvm.org
- LLVM IR
- LAV
- SMT solver
  Boolector,
  MathSAT,
  Yices,
  Z3
- Report

LLVM, LAV and SMT solvers

Milena Vujošević Janićić
Modelling Program Behaviour within LAV
LAV and External Systems

LLVM, LAV and SMT solvers

- Code
- LLVM (llvm.org)
- LLVM IR
- LAV
- SMT solver (Boolector, MathSAT, Yices, Z3)
- formula
- sat/unsat
- Report

External Systems
Symbolic execution and SAT encoding
Correctness conditions
Optimizations

Overview of the system LAV
Ongoing and future work

Modelling Program Behaviour within LAV
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

\[
\text{Transformation}(b) = \bigwedge_{v \in V} (e_b(v) = e_v) \land \text{AdditionalConstraints}(b)
\]

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:
  - \textit{store} — function for storing a value at a certain index
  - \textit{select} — function for reading a value at a certain index
  - Axioms

\[
\forall a \forall i \forall v \quad (\text{select}(\text{store}(a, i, v), i) = v)
\]
\[
\forall a \forall i \forall j \forall v \quad (i \neq j \Rightarrow \text{select}(\text{store}(a, i, v), j) = \text{select}(a, j))
\]
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:

\[
\forall p \forall n \quad left(p + n) = left(p) - n
\]
\[
\forall p \forall n \quad right(p + n) = right(p) - n
\]

- For efficiency reasons, only relevant instances of these axioms are added to the set of additional constraints attached to the block.
Overview of the system LAV
Ongoing and future work

Control Flow

Blocks of code

pred 1 → ... → pred N

succ 1 → ... → succ M

b \((c_1, \ldots, c_M)\)

SAT encoding

- Propositional variables encode transitions between blocks
- Propositional variables are used to reconstruct a program path from the model generated by a solver
Description of a block: block’s summary $\land$ control flow information

\[
\begin{align*}
\text{Description}(b) &= \text{EntryCond}(b) \land \text{Transformation}(b) \land \text{ExitCond}(b) \\
\text{EntryCond}(b) &= \text{activating}(b) \land \text{initialize}(b) \\
\text{Transformation}(b) &= \bigwedge_{v \in V} (e_b(v) = e_v) \land \text{AdditionalConstraints}(b) \\
\text{ExitCond}(b) &= \text{jump}(b) \land \text{leaving}(b)
\end{align*}
\]

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:

$$\bigg( \bigvee_{\text{pred} \in \mathcal{P}} \text{transition}(\text{pred}, b) \bigg) \iff \text{active}(b)$$

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

$$\bigwedge_{\text{pred} \in \mathcal{P}} \left( \text{transition}(\text{pred}, b) \Rightarrow \bigwedge_{v \in \mathcal{V}_f} e(\text{pred}, v) = s(b, v) \right)$$
Exit conditions

$\textit{jump}(b)$: 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 $\text{succ}_i$, and vice versa:

$$\bigwedge_{\text{succ}_i \in S} ((\text{active}(b) \land e(b, c_i)) \iff \text{transition}(b, \text{succ}_i))$$

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

$$\text{active}(b) \iff \bigvee_{\text{succ} \in S} \text{transition}(b, \text{succ})$$
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 $\psi$ of the called function is conjunction of descriptions of its blocks
  - $\psi$ is inlined into caller’s summary

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

- $C \Rightarrow \text{safe}(c)$ — correctness condition
- $C \Rightarrow \neg \text{safe}(c)$ — incorrectness condition
- $C$ — context $C$ defines command’s neighbourhood that is taken into consideration
- \text{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

- $\models C \Rightarrow \text{safe}(c)$ — the command $c$ is safe in the context $C$. It is also safe in all wider contexts (if it is reachable).

- $\models C \Rightarrow \neg\text{safe}(c)$ — the command $c$ is flawed in the context $C$. It is also flawed in all wider contexts (if it is reachable).

- $\models C \Rightarrow \text{safe}(c)$ and $\models C \Rightarrow \neg\text{safe}(c)$ — the command $c$ is unreachable. It is also unreachable in all wider contexts.

- $\not\models C \Rightarrow \text{safe}(c)$ and $\not\models C \Rightarrow \neg\text{safe}(c)$ — 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

1. \( \models C \Rightarrow \text{safe}(c) \) — the command \( c \) is safe in the context \( C \). *It is also safe in all wider contexts (if it is reachable).*

2. \( \models C \Rightarrow \neg \text{safe}(c) \) — the command \( c \) is flawed in the context \( C \). *It is also flawed in all wider contexts (if it is reachable).*

3. \( \models C \Rightarrow \text{safe}(c) \) and \( \models C \Rightarrow \neg \text{safe}(c) \) — the command \( c \) is unreachable. *It is also unreachable in all wider contexts.*

4. \( \not\models C \Rightarrow \text{safe}(c) \) and \( \not\models C \Rightarrow \neg \text{safe}(c) \) — the command \( c \) is unsafe in the context \( C \). *In some wider context it may change its status.*
Correctness conditions

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

**Contexts**
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.
Overview of the system LAV

Ongoing and future work
Related tools

Comparison to related tools

- Related tools are based on symbolic execution and model checking
- 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

- Safety-critical computer programs vs students’ programs
- Software verification can add to the quality of automated grading
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
- 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

```c
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

<table>
<thead>
<tr>
<th>no lines</th>
<th>LAV</th>
<th>CBMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>0.08</td>
<td>0.64</td>
</tr>
<tr>
<td>15</td>
<td>0.08</td>
<td>0.99</td>
</tr>
<tr>
<td>16</td>
<td>0.09</td>
<td>1.17</td>
</tr>
<tr>
<td>17</td>
<td>0.10</td>
<td>1.34</td>
</tr>
<tr>
<td>18</td>
<td>0.09</td>
<td>2.44</td>
</tr>
<tr>
<td>19</td>
<td>0.09</td>
<td>3.16</td>
</tr>
<tr>
<td>20</td>
<td>0.11</td>
<td>4.06</td>
</tr>
<tr>
<td>21</td>
<td>0.10</td>
<td>18.63</td>
</tr>
<tr>
<td>22</td>
<td>0.14</td>
<td>27.20</td>
</tr>
<tr>
<td>23</td>
<td>0.11</td>
<td>22.56</td>
</tr>
<tr>
<td>24</td>
<td>0.11</td>
<td>48.25</td>
</tr>
<tr>
<td>25</td>
<td>0.12</td>
<td>79.45</td>
</tr>
<tr>
<td>26</td>
<td>0.14</td>
<td>108.93</td>
</tr>
<tr>
<td>27</td>
<td>0.13</td>
<td>215.31</td>
</tr>
<tr>
<td>28</td>
<td>0.17</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>0.13</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0.13</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>0.23</td>
<td></td>
</tr>
</tbody>
</table>

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!