Modelling Program Behaviour within Software Verification Tool LAV

Milena Vujošević Janičić
Faculty of Mathematics, University of Belgrade
milena@matf.bg.ac.rs

Abstract

Describing program behaviour is one of the most important issues in automated software verification and there is a number of approaches for this task. We describe how the program behaviour is modelled within our software verification tool LAV in order to produce correctness conditions in terms of logical formulae. We also discuss our ongoing and future work concerning regression verification and parallelisation of verification tasks.

Keywords software verification, automated bug finding, SMT solving, modelling program behaviour

1. Introduction

In order to use logical reasoning for software verification, one of the first steps is to model program behaviour. We describe the model used and the way the program semantics is treated in our software verification tool LAV. LAV uses the LLVM intermediate representation (IR) of programs and combines bounded model checking, symbolic execution, and SAT encoding of program's control-flow to construct correctness conditions in form of first order logical formulae (by correctness conditions we mean conditions for absence of bugs such as division by zero or buffer overflows). These correctness conditions are then checked by external SMT solvers like Boolector, MathSAT, Yices, and Z3. The tool LAV is implemented in C++ and is publicly available and open-source.[1] Since it uses LLVM infrastructure, it supports several programming languages that compile into LLVM, but is primarily aimed at programs written in the C programming language. LAV was successfully applied on different benchmarks (Janičić and Kuncak 2012) and also used in automated evaluation of student's programs (Janičić et al. 2013; Janičić and Marić 2016).

2. Modelling

LLVM IR of programs consists of blocks of code. Single blocks of code are modelled by first-order logic formulae constructed by symbolic execution, while relationship between blocks is modelled by propositional formulae. Formulae that describe blocks' behaviour are combined with correctness conditions for individual commands to produce correctness conditions of the program.

Store and Blocks A *store* of a program is a mapping from variables to values from their domains. Each instruction transforms the store and may add some constraints over variables. In our approach, symbolic execution is used to compute a FOL formula $Transformation(b)$ that describes the transformation of an individual block b (i.e., a *block summary*):

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

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, $s(b,v)$. The formulae $AdditionalConstraints(b)$ are introduced for modelling some sorts of operations and constraints not discussed here in more detail. A formula $Transformation(b,i)$, a *block context* for the $(i+1)$th command, is defined by analogy, but only considering the first i instructions of the block b.

Buffers, Structures and Unions Buffers are sequences of memory locations allocated statically or dynamically and accessible by a pointer and an offset. While these pointers are treated as any other simple variables, they are also associated with sizes of corresponding buffers which are captured via introduced functions: $left(p)$ and $right(p)$ for numbers of bytes reserved for the pointer p on its left and its right hand side. Note that it always holds $left(p+n) = left(p)-n$ and $right(p+n) = right(p)-n$. These equalities can be considered as axioms, but, instead of introducing an universally quantified formula into the generated formula, we can only add all of its relevant instances to the set of additional constraints attached to the block.

Memory Contents The memory can be treated as an array mem of memory locations, that may get updated during the symbolic execution, just as any other variable. For modelling commands that access the memory via pointers, we use the theory of arrays. The theory of arrays provides functions for storing a value at a certain index in the array (*store*) and for reading a value at a certain index in the array (*select*). If there is a reference operator on a local variable within a function, then this variable is not tracked through its slot in the store, as other variables, but through the memory content.

Global Variables Global variables are accessible in all functions (and, hence, in all blocks), but instead of representing them individually within all functions, they are modelled by the variable modelling memory.

Function Calls Function calls are modelled 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 of the function. If a contract of the function is not available, but the definition of function is, then interprocedural analysis is required. If neither a contract nor the definition of function are available, then the memory contents (i.e. the current array mem) is set to a new (fresh) variable as an effect of the function call.

Intraprocedural Loop-free Control Flow Relationship between blocks can be encoded by propositional variables and SAT formulae. Generally, a block can be reached from several blocks and, also, it can lead (subject to certain conditions) to several blocks.

[1] `http://argo.matf.bg.ac.rs/?content=lav`

Suppose, for a moment, that the program has no loops in the control-flow graph. A path in this graph is then determined by the sequence of nodes (representing blocks) and edges (representing transitions from one block to another). We introduce $active(b)$ (that can be represented by a propositional variable) to denote that a node b is in the path (or, intuitively, that the block b was reached), and introduce $transition(b_i, b_j)$ (that can be again represented by a propositional variable) to denote that the edge from b_i to b_j is in the graph (i.e., that the block j was reached from the block i). Given such a path, we can consider all program executions that follow this path, by composing formulae for each basic block.

Let us suppose that a block b is reachable from blocks $\mathcal{P} = \{pred_1, pred_2, \ldots, pred_n\}$ and let blocks $\mathcal{S} = \{succ_1, succ_2, \ldots, succ_m\}$ be reachable from the block b (it can be assumed that all $succ_i$ are different and that all $pred_i$ are different) . We define several formulae describing control flow involving the block b.

A formula $EntryCond(b)$ is defined as $activating(b) \wedge initialize(b)$.

$activating(b)$**:** The formula $activating(b)$ says that there was a transition from a predecessor block to the block b iff the block b was active. It is defined as follows:

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

If the block does not have predecessors (i.e. it is the entry block of the function), then $activating(b)$ is defined as $active(b)$. This way, it is not analyzed whether a function itself is reachable. Rather, it is analyzed assuming that it can be reached.

$initialize(b)$**:** 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 $pred$. This defines the $initialize(b)$ formula:

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

If the block b does not have predecessors, then $initialize(b)$ is defined as $\top$.

The formula $ExitCond(b)$ is defined as $jump(b) \wedge leaving(b)$.

$jump(b)$**:** 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. This defines the $jump(b)$ formula:

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

If the block has only one successor, then $jump(b)$ is defined as $active(b) \Leftrightarrow transition(b, succ)$. If the block does not have successors, then $jump(b)$ is defined as $\top$.

$leaving(b)$**:** The formula $leaving(b)$ says that the block b was active iff it led to some other block (or to exit of the function). It is defined as follows:

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

If the block does not have successors, then $leaving(b)$ is defined as $\top$.

Finally, the formula *Description(b)* is describing the block b:

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

Note that all of the above formulae are of polynomial size in the number of predecessors and successors of b, the number of variables of the function that b belongs to, and the number of instructions in b. Consequently, the size of the above formula is bounded polynomially in the size of the function.

Loops Our system supports two techniques for dealing with loops: underapproximation of loops (like in bounded model checking) and overaproximation of loops (in this case, the unrolled code simulates the first n and the last m entries to the loop, where n and m are configurable parameters). In both cases, loops are eliminated by unrolling. This way, the control flow graph of the function has no cycles and the basic modelling mechanism can be applied.

Interprocedural Control Flow Starting with block descriptions as building blocks, and given there are a unique entry and a unique leaving point of each function (this uniqueness can be ensured, as in the LLVM code) the description of a function is constructed as a conjunction of descriptions of the function blocks. Recursive function calls can be unrolled in the same way as loops.

Safe, Unsafe and Flawed Commands In order to check whether some command leads to an error we build (two) formulae of the form $C \Rightarrow (\neg)safe(c)$, where C is a formula describing context (empty context, block context, function context, wider context) and $(\neg)safe(c)$ is a formula describing (in)correctness condition of a command (it can be given by a bug definition — division by zero, buffer overflow, dereferencing null pointers, or it can be given by an annotation). The main verification goal is to check (for all commands) whether it holds $safe(c)$. If $safe(c)$ holds, then the command c is *safe*, and if $\neg safe(c)$ holds, the command c is *flawed*. If neither $safe(c)$ nor $\neg safe(c)$ holds in a general case, then the command c is considered *unsafe*. The difference between a flawed and an unsafe command is that the flawed command always leads to an error in the program, while unsafe command leads to an error only in some cases, depending on the context of the command, i.e., to the path condition leading to the command. Therefore, an unsafe command may turn safe/flawed if a wider context.

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

3. Ongoing and Future Work

Currently, we are working on enriching LAV by regression verification techniques (Janičić and Marić 2016). We use these techniques in context of automated evaluation of students' programs to automatically prove functional equivalence (partial and k-equivalence) between student's and teacher's solution, in order to gain higher level of reliability in automated grading. We develop a set of tools for transformations of programs that are necessary for this purpose and use LAV for checking equivalence.

We are also working on parallelisation of verification tasks within LAV, by taking advantage of hardware properties and characteristics of software verification conditions. Different contexts used in LAV give room for different kind of parallelisations. We implemented parallel verification of different functions and parallel checking of correctness conditions within one block. We got very promising experimental results in both cases, showing that parallelisation may scale well in cases where classical bounded model checking times out.

In addition, we are looking for ways to build firmer theoretical grounds for LAV, possibly within the context of type theory, hopefully leading to new insights and further improvements of the tool.

References

M. V. Janičić and V. Kuncak. Development and Evaluation of LAV: An SMT-Based Error Finding Platform. In *VSTTE*, LNCS, 2012.

M. V. Janičić and F. Marić. Regression Verification for Automated Evaluation of Students Programs, 2016. Submitted.

M. V. Janičić, M. Nikolić, D. Tošić, and V. Kuncak. Software verification and graph similarity for automated evaluation of students assignments. *Information and Software Technology*, 55(6), 2013.