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 formulæ constructed by symbolic execution, while relationship between blocks is modelled by propositional formulæ. Formulæ 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):

\[ Transformation(b) = \bigwedge_{v \in V} (e(b, v) = e_v) \wedge AdditionalConstraints(b) \]

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 formulæ 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 formulæ. Generally, a block can be reached from several blocks and, also, it can lead (subject to certain conditions) to several blocks.
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 \( \text{activating}(b) \) (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 \( \text{transition}(b, b_j) \) (that can be again represented by a propositional variable) to denote that the edge from \( b \) 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 \( P = \{ \text{pred}_1, \text{pred}_2, \ldots, \text{pred}_n \} \) and let blocks \( S = \{ \text{succ}_1, \text{succ}_2, \ldots, \text{succ}_m \} \) be reachable from the block \( b \) (it can be assumed that all \( \text{succ}_i \) are different and that all \( \text{pred}_i \) are different). We define several formulae describing control flow involving the block \( b \).

A formula \( \text{EntryCond}(b) \) is defined as \( \text{activating}(b) \land \text{initialize}(b) \).

\( \text{activating}(b) \): The formula \( \text{activating}(b) \) 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:

\[
\bigvee_{\text{pred} \in P} \text{transition}(\text{pred}, b) \iff \text{active}(b)
\]

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

\( \text{initialize}(b) \): 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 \( \text{pred} \). This defines the \( \text{initialize}(b) \) formula:

\[
\bigwedge_{\text{pred} \in P} \left( \text{transition}(\text{pred}, b) \Rightarrow \bigwedge_{v \in V} \text{c}(\text{pred}, v) = s(b, v) \right)
\]

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

The formula \( \text{ExitCond}(b) \) is defined as \( \text{jump}(b) \land \text{leaving}(b) \).

\( \text{jump}(b) \): 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. This defines the \( \text{jump}(b) \) formula:

\[
\bigwedge_{\text{succ}_i \in S} \left( (\text{active}(b) \land c(b, c_i)) \iff \text{transition}(b, \text{succ}_i) \right)
\]

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

\( \text{leaving}(b) \): The formula \( \text{leaving}(b) \) 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:

\[
\text{active}(b) \iff \bigvee_{\text{succ} \in S} \text{transition}(b, \text{succ})
\]

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

Finally, the formula \( \text{Description}(b) \) is describing the block \( b \):

\[
\text{EntryCond}(b) \land \text{Transformation}(b) \land \text{ExitCond}(b)
\]

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 check-