Software Verification — Homework

Emanuel Fleury, Jérôme Leroux, and Grégoire Sutre

Univ. de Bordeaux & CNRS, LaBRI, UMR 5800, Talence, France

Abstract The goal of this homework is to implement a bounded model-checker based on a depth-first-search exploration of all the feasible paths of the input program. Moreover, the SMT-solver Z3 will be used as an execution engine to compute all the successive states along the exploration. We will first present the global picture of it, then introduce a few definition and semantics and, finally, list the steps you may follow to implements the whole model-checking program.

1 Project Overview

The point of this homework is to implement a small bounded model-checker in OCaml. Remember that the "*Bounded model-checking*" method is a way to ensure termination by exploring feasible paths up to a given depth (let say 10 by default). Exploration will terminate in one of the following three states:

- "`No feasible path`":
  All feasible paths have been explored and terminate within the given bound.
- "`No feasible path of length <= 10`":
  Some feasible paths did hit the depth-bound, the result is inconclusive.
- "`Feasible path`":
  A feasible error path has been found and is displayed.

The bounded model-checker will use a *depth-first search* (DFS) strategy to explore all the feasible paths. And, Z3 will be used as an execution engine to compute the effect of the program on its context. More precisely, each feasible path will start from the "*initial*" state and explore each branch of the control-flow automaton (CFA) by collecting the guards and the operations along the edges and adding it to a path formula. This path formula will, then, be checked at each step with Z3 to know if the "*final*" state (or "*bad*" state) is reachable. When a state can reach a context which allows to reach the final state, an error trace is extracted and displayed.

In the remainder, you will find a more formal definition of program operations and of feasible paths and, finally, of the bounded model-checking method.

2 Program Operations

Expressions, *guards*, and *program operations* over a finite set X of variables are defined thanks to the following grammars where $x \in X$ and $z \in \mathbb{Z}$:

$$\begin{array}{rcl} \mathit{exp} & := & x \mid z \mid \mathit{exp} + \mathit{exp} \mid \mathit{exp} - \mathit{exp} \mid \mathit{exp} * \mathit{exp} \mid \mathit{exp}/\mathit{exp} \\ \mathit{guard} & := & \mathit{exp} = \mathit{exp} \mid \mathit{exp} < \mathit{exp} \mid \mathit{exp} > \mathit{exp} \mid \mathit{exp} \leq \mathit{exp} \mid \mathit{exp} \geq \mathit{exp} \\ \mathit{op} & := & x := \mathit{exp} \mid \mathit{guard} \mid \mathit{skip} \end{array}$$

3 Feasible Sequences of Program Operations

An *indexed variable* of a finite set of variables X is a pair (x, i) in $X \times \mathbb{Z}$. Such a pair (x, i) is simply denoted as $x^{(i)}$ in the sequel.

We associate to every expression e over X, the expression $e^{(i)}$ defined by induction as follows:

$$e^{(i)} = \begin{cases} x^{(i)} & \text{if } e \text{ is the variable } x \text{ in } X \\ z & \text{if } e \text{ is the integer } z \text{ in } \mathbb{Z} \\ e_1^{(i)} + e_2^{(i)} & \text{if } e \text{ is } e_1 + e_2 \\ e_1^{(i)} - e_2^{(i)} & \text{if } e \text{ is } e_1 - e_2 \\ e_1^{(i)} * e_2^{(i)} & \text{if } e \text{ is } e_1 * e_2 \\ e_1^{(i)} / e_2^{(i)} & \text{if } e \text{ is } e_1/e_2 \text{ and } e_2 \not\equiv 0 \end{cases}$$

Symetrically, we associate to every guard g of the form $e_1 \sim e_2$ where $\sim$ is an operator in $\{=, <, >, \leq, \geq\}$, the guard $g^{(i)}$ defined as $e_1^{(i)} \sim e_2^{(i)}$.

We associate to a program operation *op* the formula $\mathit{op}^{(i)}$ defined as follows:

$$\mathit{op}^{(i)} = \begin{cases} x^{(i+1)} = e^{(i)} \wedge \bigwedge_{y \in X \setminus \{x\}} y^{(i+1)} = y^{(i)} & \text{if } \mathit{op} \text{ is } x := e \\ g^{(i)} \wedge \bigwedge_{y \in X} y^{(i+1)} = y^{(i)} & \text{if } \mathit{op} \text{ is a guard } g \\ \bigwedge_{y \in X} y^{(i+1)} = y^{(i)} & \text{if } \mathit{op} \text{ is } \mathit{skip} \end{cases}$$

Example 3.1. Assume that X is $\{x, y\}$ and *op* is the assignement $x := x + y$ then $\mathit{op}^{(0)}$ is the formula $x^{(1)} = x^{(0)} + y^{(0)} \wedge y^{(1)} = y^{(0)}$.

A sequence $\mathit{op}_1, \ldots, \mathit{op}_k$ of operations is said to be *feasible* if the following formula is satisfiable:

$$\mathit{op}_1^{(0)} \wedge \ldots \wedge \mathit{op}_k^{(k-1)}$$

4 Bounded Model Checking

A *control flow automaton (CFA for short)* is a tuple $(Q, q_{\text{ini}}, q_{\text{bad}}, X, \varDelta)$ where Q is a non-empty finite set of *control states*, q_{ini} is the *initial control state*, q_{bad} is the *final one*, X is a finite set of variables, and $\varDelta$ a finite set of triples $\delta = (p, op, q)$ where $p, q \in Q$ and op is an operation. δ is called a *transition.*

The *bounded model checking problem* takes as input a CFA and an integer $\ell \geq 0$, and checks if there exists a feasible sequence of operations $op_1, \ldots, op_k$ with $k \leq \ell$, that labels a path in the CFA from the initial control state to the final one, i.e., if there exists a sequence $q_0, \ldots, q_k$ of control states such that $q_0 = q_{\text{ini}}$, $q_k = q_{\text{bad}}$, and such that $(q_{j-1}, op_j, q_j) \in \varDelta$ for every $1 \leq j \leq k$.

5 Working on the Project

5.1 Homework's archive

The archive encloses a skeleton of your project with:

- This PDF document (in the root directory).
- The "`.mli`" files that define the API of the whole project. Run '`make doc`' to generate the HTML documentation of the modules and visit `./doc/index.html` with a web browser to look at it.
- A few "`.ml`" files to provide you functions to help you to focus on the heart of the program (look at it and use it to save time!).
- An example usage of the Z3 OCaml API in the file `z3_ml_example.ml`. Compile it with '`make z3_ml_example.byte`'.
- Once your have implemented enough of the two missing modules (see below), you can try to compile the whole thing with: '`make`'. You can also run a few unit tests with '`make test`'.
- A binary program (`bmc.d.byte`) to give you a taste of the final model-checking program that you should obtain.

5.2 What to code ?

You need to implement the following OCaml modules:

- `Semantics.ml`: Translate the operators of the program into Z3 expressions in the integer theory module.
- `BoundedModelChecking.ml`: Explore all the feasible paths in depth-first search with a bounded depth. And, returns the result of the exploration.

5.3 How and when to send your homework ?

The deadline for the project is on Friday, December 16, 2016 at 14h00. Send an archive with your full project to Grégoire Sutre `<gregoire.sutre@labri.fr>`. Your archive should contain the whole project plus:

- `Semantics.ml` and `BoundedModelChecking.ml` modules fully implemented.
- A few extra tests for the implemented modules.
- A report (written in LaTeX but sent as a PDF file) about your implementation and the choices that you had to do while programming.