# Software Verification

2024-2025

## TP3: Bounded Model Checking

Vincent Penelle

---

**Content**

- Implementation of Depth-First Search Bounded Model-Checking.
- Implementation of Global Algorithm for Bounded Model-Checking.

---

The goal of this lab is to make you program two modules: DepthFirstBMC.ml and GlobalBMC.ml of our tool, Simple Program Analyser. These modules will each implement an algorithm for Bounded Model-Checking exploring the execution tree, the first in depth, the second level by level thanks to the unfolding of the step formula.

**Reminder:**

A *control-flow automaton* is a tuple $(Q, q_i, q_{bad}, \Delta)$ where $Q$ is a finite set of states, $q_i \in Q$ is the initial state, $q_{bad}$ is the final state (or "bad"), and $\Delta \subseteq Q \times \mathrm{Op} \times Q$ is the set of transitions, where Op is the set of possible operation of the automaton. This set is defined hereafter. Said otherwise, it is an automaton labelled with operations (over variables)

We consider a set of variables $X$. An *expression* is an arithmetical expression over $\mathbb{Z}$ containing or not variables from $X$:

$$\texttt{exp} ::= n \in \mathbb{Z} | x \in X | \texttt{exp} + \texttt{exp} | \texttt{exp} - \texttt{exp} | \texttt{exp} \times \texttt{exp} | \texttt{exp} / \texttt{exp}$$

A *guard* is a comparison between two expressions:

$$\texttt{guard} ::= \texttt{exp} = \texttt{exp} | \texttt{exp} < \texttt{exp} | \texttt{exp} \leq \texttt{exp} | \texttt{exp} > \texttt{exp} | \texttt{exp} \geq \texttt{exp}$$

An operation is either `skip`, or a guard, or an affectation of the form $x := exp$, for $x \in X$ and $exp$ an arithmetical expression:

$$op ::= \texttt{skip} | \texttt{guard} | x := \texttt{exp}$$

(cf TP précédent pour plus de détails sur les opérations et leur sémantique)

**Technical Point:**

A *configuration* of a control-flow automaton $\mathcal{A}$ is a pair $(q, \sigma)$, where $q$ is a state, and $\sigma$ a function from $X$ to $\mathbb{Z}$ called *valuation*.

The *semantic* of a operation *op* is a relation containing valuations $(\sigma_1, \sigma_2)$ such that $\sigma_2$ can be obtained by applying *op* to $\sigma_1$. Formally, we have $[\![\texttt{skip}]\!] = \mathrm{id}$, $[\![\mathrm{guard}]\!]$ is the set containing all $(\sigma, \sigma)$ such that $\sigma$ satisfies the guard, and $[\![\texttt{x} := \texttt{exp}]\!] = \{(\sigma, \sigma[x := exp])\}$. Cf last lab session for more details on the implementation of the semantics in an FO formula.

The semantic of a transition $t = (q, \mathrm{op}, q')$ is the binary relation $\xrightarrow{t}$ on configurations defined by $c \xrightarrow{t} c'$ if $c = (q, \sigma)$, $c' = (q', \sigma')$ and $(\sigma, \sigma') \in [\![\mathrm{op}]\!]$.

The *step relation* $\to_{\mathcal{A}}$ is the union of semantics of all transitions of $\mathcal{A}$ : $\bigcup_{t \in \Delta_{\mathcal{A}}} \xrightarrow{t}$.

Am *execution* is a sequence $c_0, t_1, c_1, \ldots, t_n, c_n$ alternating configurations $c_i$ and transitions $t_i$ such that $c_{i-1} \xrightarrow{t_i} c_i$ for all $0 < i \leq n$. Such a execution is also written $c_0 \xrightarrow{t_1} c_1 \cdots \xrightarrow{t_n} c_n$ for improved readability, and $n$ is its *length*.

A path $q_0, \mathrm{op}_1, q_1, \ldots, \mathrm{op}_n, q_n$ is said to be *executable* if there exist valuations $\rho_0, \ldots, \rho_n \in \mathbb{Z}^X$ such that $(q_0, \rho_0) \xrightarrow{t_1} (q_1, \rho_1) \cdots \xrightarrow{t_n} (q_n, \rho_n)$ is an execution, with $t_i = (q_{i-1}, \mathrm{op}_i, q_i)$.

**Technical Point:**

The bounded model-checking problem asks, given a control-flow automaton $\mathcal{A}$ and a bound $k \in \mathbb{N}$, whether there exist two configurations $(q, \rho)$ and $(q', \rho')$ such that:

- $q = q_i$,
- $q' = q_{bad}$, and
- $(q, \rho) \underbrace{\to_{\mathcal{A}} \cdot \to_{\mathcal{A}} \cdots \to_{\mathcal{A}}}_{i \text{ times}} (q', \rho')$ for $i \leq k$.

The problem can be equivalently expressed as: Does there exist an executable path of lenght at most $k$ from $q_i$ to $q_{bad}$.

The algorithm of bounded model-checking can simply be summarised as follows: enumerate every paths of lenght at most $k$, and as soon as we find an executable one from initial to target state, return that path.

Of course, there exist several possible exploration order of these paths, and some simple but efficient optimisations (namely, it is useless to consider a path whose one of the prefixes is not executable). We can also ask to the program to determine if it did an exhaustive exploration or not (whether longer executions exist).

For now, we will implement a depth-first exploration order: given an (arbitrary) order on transitions, look first all paths starting by the first, then the second, etc. As soon as either the bound is reached or a non-executable path is found, we backtrack one level and continue to explore from there. As soon that a faulty execution is reached, we stop exploration and return it.

More precisely, the (recursive) algorithm receives as argument a CFA $\mathcal{A}$, a path $\tau$, a current state $q$, the current depth $\ell$, and the bound $k$, and can be summarised as follows:

- if $\ell > k$, return "non-exhaustive".

- if $\tau$ is not executable, return "exhaustive".
- if $\tau$ is executable and $q = q_{bad}$, we found a faulty execution : return $\tau$.
- if $\tau$ is executable and $q \neq q_{bad}$, we continue exploration from the current node of the tree: for each transition of the form $(q, \mathrm{op}, q')$, we call the algorithm on $(\mathcal{A}, \tau :: ((q, \mathrm{op}, q'), q', \ell + 1, k)$. If all calls return "exhaustive", we return "exhaustive". If all calls return either "exhaustive" or "non-exhaustive", we return "non-exhaustive". As soon as we receive a path $\tau'$ on one of the calls we return this $\tau'$ and do not explore other executions.

The algorithm is called on $(\mathcal{A}, (q_i), q_i, 0, k)$ to perform bounded model-checking with bound $k$ on the automaton $\mathcal{A}$.

**Exercise 1: Depth-first search bounded model-checking** Download the sources of the lab session. In `DepthFirstBMC.ml`, you have to implement `dfs` which performs the depth-first exploration of the tree described above. The commentary above the prototype contains all elements needed to help you implementing it, especially by adapting the arguments to facilitate the computation (e.g. by retaining a formula to describe the path up to the current state, instead of the path itself). There will be some subtlety left to you to return correctly the faulty execution.

You can also test your program over the given examples and compare with the given working executable `bmc.solution.d.byte`.

**Technical Point:**

The global algorithm consist in determining, length by length up to the bound, whether there exists a faulty run of that length. It is equivalent to ask if, for a length $k$, there exist states $q_1, \cdots, q_{k-1}$ and valuations $X_0, \cdots, X_k$, such that $(q_{in}, X_0 \underset{\mathcal{A}}{\rightarrow} (q_1, X_1) \underset{\mathcal{A}}{\rightarrow} \cdots \underset{\mathcal{A}}{\rightarrow} (q_{k-1}, X_{k-1}) \underset{\mathcal{A}}{\rightarrow} (q_{bad}, X_k)$.

To this end, we start by computing the formula $\phi_{\text{step}}$ of the automaton, relating two configuration if and only if the automaton allows to go from one to the other. With it, for each length $i$ up to the bound, we check whether the following formula is satisfiable:

$$\psi_i(q_0, X_0, \cdots, q_i, X_i) \overset{\text{def}}{=} q_0 = q_{in} \wedge \bigwedge_{j=1}^{i} \varphi_{\text{step}}(q_{j-1}, X_{j-1}, q_j, X_j) \wedge q_i = q_{bad}$$

If so, we have the existence of a faulty execution (and we must reconstruct it from the model of the formula, knowing that some information are missing in the formula given here). If not, there is no faulty execution of length $i$. If we treated length one by one, we can thus follow through to the next.

Note that it is useless to test $\psi_i$ if no execution of length $i$ exist. To determine it, it is sufficient to ask the solver to satisfy $\psi_i$ without ensure that the last state is $q_{bad}$ (i.e., without the red part). If so, we have to continue to explore, but if it is unsatisfiable, there is no execution of length $i$, and we can certify the program to be correct (exhaustive exploration).

### Exercise 2: Global bounded model-checking

In `GlobalBMC.ml`, implement the function asked for. Indications are given there. Do not hesitate to implement auxiliary functions. In `Z3Helper.mli`, you have functions to help you manipulate states and transitions of the automaton in the formula.

Start with implementing a version without caring about sending back the counter-example, then, when it works as indented, modify it to send the counter-example.