Initiation à la Vérification

Emptiness Test for Büchi automata

Stefan Schwoon

November 15, 2018
Overview

Result from the first half of the course:

Model-checking LTL reduces to checking emptiness of some Büchi automaton $B$.

Reminder (for universal model-checking, existential is analogue):

$B$ is the intersection of a Kripke structure $K$ with a BA for the negation of an LTL formula $\phi$.

If $B$ accepts some word, we call such a word a counterexample.

$K \models \phi$ iff $B$ accepts the empty language.
Typical instances:

Size of $\mathcal{K}$: between several hundreds to millions of states.

Size of $B_{\neg \phi}$: exponential in $|\phi|$, but usually just a couple of states.

Typical setting:

$\mathcal{K}$ indirectly given by some concise description (modelling or programming language); model-checking tools will generate $\mathcal{K}$ internally.

$B_{\neg \phi}$ can be generated from $\phi$ before start of emptiness check.
Typical setting:

\( \mathcal{B} \) generated “on-the-fly” from (the description of) \( \mathcal{K} \) and from \( \mathcal{B}_{\neg \phi} \) and tested for emptiness \textit{at the same time}.

As a consequence, the size of \( \mathcal{K} \) (and of \( \mathcal{B} \)) is not known initially!

At the beginning, only the initial state is known, and we have a function
\( \text{succ} : S \rightarrow 2^S \) for computing the immediate successors of a given state (where \( \text{succ} \) implements the semantics of the description).
Naïve solution: Check for Lassos

Let $\mathcal{B} = (\Sigma, S, s_0, \delta, F)$ be a Büchi automaton.

$\mathcal{L}(\mathcal{B}) \neq \emptyset$ iff there is $s \in F$ such that $s_0 \rightarrow^* s \rightarrow^+ s$

Naïve solution:

- Check for each $s \in F$ whether there is a cycle around $s$; let $F_\circ \subseteq F$ denote the set of states with this property.

- Check whether $s_0$ can reach some state in $F_\circ$.

Time requirement: Each search takes linear time in the size of $\mathcal{B}$, altogether quadratic run-time $\rightarrow$ unacceptable for millions of states.
Strongly connected components

\[ C \subseteq S \text{ is called a strongly connected component (SCC) iff} \]

\[ s \rightarrow^* s' \text{ for all } s, s' \in C; \]

\[ C \text{ is maximal w.r.t. the above property, i.e. there is no proper superset of } C \text{ satisfying the above.} \]

An SCC \( C \) is called \textit{trivial} if \(|C| = 1\) and for the unique state \( s \in C \) we have \( s \not\rightarrow s \) (single state without loop).
Example: SCCs

The SCCs \{s_0\} and \{s_1\} are trivial.
Depth-first search (basic version)

nr = 0;
hash = {};
dfs(s0);
exit;

dfs(s) {
    add s to hash;
nr = nr+1;
s.num = nr;

    for (t in succ(s)) {
        // deal with transition s -> t
        if (t not yet in hash) { dfs(t); }
    }
}
Memory usage

Global variables: counter $nr$, hash table for states

Auxiliary information: “DFS number” $s.num$

search path: Stack for memorizing the “unfinished” calls to $dfs$
Solution (1): based on SCCs

The algorithm of Tarjan (1972) can identify the SCCs in linear time (i.e. proportional to $|S| + |\delta|$).

Said algorithm is a slight extension of basic DFS with some additional constant-time operations on each state and transition.

Given the SCCs, one can then check if there exists a non-trivial SCC containing an accepting state.
Solution (2): nested DFS


The nested-DFS algorithm is an alternative requiring only two bits per state.

States are “white” initially.

A first DFS makes all the states that it visits blue.

Whenever the first (blue) DFS backtracks from an accepting state $s$, it starts a second (red) DFS to see if there is a cycle around $s$.

The red DFS only visits states that are not already red (including from a previous visit). Thus, every state and edge are considered at most twice.
Nested depth-first search: Algorithm

hash = {};
blue(s0);
report "no accepting run"

blue(s) {
    add (s, 0) to hash;
    for t in succ(s)
        if (t, 0) not in hash { blue(t) }
    if s is accepting and (s, 1) not in hash { seed=s; red(s) }
}

red(s) {
    add (s, 1) to hash;
    for t in succ(s)
        if t=seed { report "accepting run found"; exit }
        if (t, 1) not in hash { red(t) }
}
Nested DFS: Example

Blue phase: Start at initial state.
Nested DFS: Example

Visit states depth-first, colouring them blue.
Nested DFS: Example

Simply backtrack from non-accepting states.
Nested DFS: Example

Continue blue search …
Nested DFS: Example

Continue blue search until backtracking from an accepting state.
Nested DFS: Example

Before backtracking, start a “red” DFS …
Nested DFS: Example

...that searches for a loop back to that accepting state.
Nested DFS: Example

If red search is unsuccessful, backtrack.
Nested DFS: Example

Carry on …
Nested DFS: Example

Future red searches only consider non-red states.
Properties of Nested DFS

Very economic in terms of memory

Implemented in state-of-the-art tools like Spin

Can be combined with further optimization (partial-order reduction)

Tends to prefer long counterexamples “deep down” in the state graph

→ variants of Tarjan (not shown) can identify counterexamples more quickly, but are less economic on memory and more difficult to combine with other optimizations