Non-emptyness Check for Generalized Büchi Automata

Master Thesis Topic

Sütő Ágoston

Research Institute for Symbolic Computation

Thesis supervisor: Prof. Wolfgang Schreiner

May 15, 2022

Previously...

Discussed last time:

- What is model checking
- RISCAL software system
- Kripke-structures and LTL
- Generalized Büchi Automata
- A concrete approach for automaton-based model checking

Next up

Will be discussed today:

- It's alive!
- But the original approach wasn't very good
- How it was improved
- Why it's still not very good
- How it will be improved further

Demo

DEMO

Automaton based model checking (as described last time)

Definition

Model checking problem
Given a Kripke-structure $K = (S, I, T, \mathcal{L})$ and an LTL formula f determine whether $K \models f$, and if not, provide a trace π of K such that $\pi \not\models f$.

1. Negate the formula and preprocess it
2. Transform this formula into an LGBA $\mathcal{A}_{\neg f}$
3. Given the Kripke-structure $K = (S, I, T, \mathcal{L})$ of the system, construct LGBA $\mathcal{A}_K = (S, I, 2^{\mathcal{P}}, \mathcal{L}', T, \emptyset)$ with $\mathcal{L}'(s) = \{\mathcal{L}(s)\}$ for any $s \in S$.
4. Construct the automaton which accepts the intersection of the languages of $\mathcal{A}_{\neg f}$ and $\mathcal{A}_K$
5. Transform the resulting LGBA to a simple Büchi automaton
6. Check if the language of the resulting automaton is empty. If so, the property holds.

Automaton based model checking (as described last time)

Definition

Model checking problem
Given a Kripke-structure $K = (S, I, T, \mathcal{L})$ and an LTL formula f determine whether $K \models f$, and if not, provide a trace π of K such that $\pi \not\models f$.

1. Negate the formula and preprocess it ✓
2. Transform this formula into an LGBA $\mathcal{A}_{\neg f}$ ✓
3. Given the Kripke-structure $K = (S, I, T, \mathcal{L})$ of the system, construct LGBA $\mathcal{A}_K = (S, I, 2^{\mathcal{P}}, \mathcal{L}', T, \emptyset)$ with $\mathcal{L}'(s) = \{\mathcal{L}(s)\}$ for any $s \in S$. ✓
4. Construct the automaton which accepts the intersection of the languages of $\mathcal{A}_{\neg f}$ and $\mathcal{A}_K$ ✓
5. Transform the resulting LGBA to a simple Büchi automaton
6. Check if the language of the resulting automaton is empty. If so, the property holds.

Automaton based model checking (as described last time)

Definition

Model checking problem

Given a Kripke-structure $K = (S, I, T, \mathcal{L})$ and an LTL formula f determine whether $K \models f$, and if not, provide a trace π of K such that $\pi \not\models f$.

1. Negate the formula and preprocess it ✓
2. Transform this formula into an LGBA $\mathcal{A}_{\neg f}$ ✓
3. Given the Kripke-structure $K = (S, I, T, \mathcal{L})$ of the system, construct LGBA $\mathcal{A}_K = (S, I, 2^{\mathcal{P}}, \mathcal{L}', T, \emptyset)$ with $\mathcal{L}'(s) = \{\mathcal{L}(s)\}$ for any $s \in S$. ✓
4. Construct the automaton which accepts the intersection of the languages of $\mathcal{A}_{\neg f}$ and $\mathcal{A}_K$ ✓
5. Transform the resulting LGBA to a simple Büchi automaton ✗
6. Check if the language of the resulting automaton is empty. If so, the property holds. ✗

Simple emptiness check for Büchi automata

```
function isLanguageEmpty(initialStates, acceptingStates) {
    S1: stack of states = stack(initialStates)
    S2: stack of states = ∅
    M1, M2: sets of states = ∅

    while (S1 ≠ ∅) {
        x = S1.top()

        if (there is a state y ∈ x.next with y ∉ M1) {
            M1 = M1 ∪ {y}
            S1.push(y)
        } else {
            S1.pop()
            if (x ∈ acceptingStates) {
                S2.push(x)
                while (S2 ≠ ∅) {
                    v = S2.top()
                    if (x ∈ v.next) {
                        return false
                    } else if (there is a state w ∈ v.next with w ∉ M2) {
                        M2 = M2 ∪ {w}
                        S2.push(w)
                    } else {
                        S2.pop()
                    }
                }
            }
        }
    }

    return true
}
```

Emptyness check comparisons

algorithm	run-time
ASCC	67.0 %
GV	69.2 %
AND	69.7 %
SE	96.3 %
HPY	100.0 %
C99	128.3 %

Can work directly with GBAs!

The one on the previous slide is even worse!

The one used in SPIN (at least back in 2009)

Fig. 4. Performances

Figure: Comparison of emptyness check algorithms, according to Gaiser & Schwoon 2009 [1]

Strongly connected components

Definition

A *strongly connected component (SCC)* of a directed graph $\mathcal{G} = (V, E)$ is a subset $S \subseteq V$ such that for any pair $s, t \in S$ we have that $s \rightarrow_S^* t$.
An SCC is called *trivial* if $S = \{s\}$ and $s \not\rightarrow s$.

Strongly connected components

Definition

A *strongly connected component (SCC)* of a directed graph $\mathcal{G} = (V, E)$ is a subset $S \subseteq V$ such that for any pair $s, t \in S$ we have that $s \to_S^* t$.
An SCC is called *trivial* if $S = \{s\}$ and $s \not\to s$.

Recall:

Proposition

The language described by a Büchi automaton $\mathcal{A} = (A, I, \Sigma, \mathcal{L}, \to, F)$ is non-empty if and only if there exists a state $s \in F$ such that $s_I \to^* s$ for some $s_I \in I$ and $s \to^+ s$.

Strongly connected components

Using SCCs this can be reformulated as:

Proposition

The language described by a Büchi automaton $\mathcal{A} = (A, I, \Sigma, \mathcal{L}, \rightarrow, F)$ *is non-empty if and only if there exists an SCC* $\mathcal{C}$ *reachable from* I *such that* $\mathcal{C} \cap F \neq \emptyset$.

Strongly connected components

Using SCCs this can be reformulated as:

Proposition

The language described by a Büchi automaton $\mathcal{A} = (A, I, \Sigma, \mathcal{L}, \rightarrow, F)$ *is non-empty if and only if there exists an SCC* $\mathcal{C}$ *reachable from* I *such that* $\mathcal{C} \cap F \neq \emptyset$.

For generalized Büchi automata the acceptance condition using reachability is harder to state, but using SCCs we have:

Proposition

The language described by a generalized Büchi automaton $\mathcal{A} = (A, I, \Sigma, \mathcal{L}, \rightarrow, \mathcal{F})$ *is non-empty if and only if there exists an SCC* $\mathcal{C}$ *reachable from* I *such that* $\mathcal{C} \cap F \neq \emptyset$ *for all* $F \in \mathcal{F}$.

The ASCC algorithm

- The ASCC algorithm works by finding the strongly connected components of the automaton and checking if they contain at least one state in each final set.
- Avoids a potential polynomial blowup of states.
- In reality most properties have a corresponding automaton with one or zero final sets (90-95% according to [2], 92% in the test-set of [1]), so it doesn't help that much.
- Still it simplifies the implementation a bit.
- It is an improvement over Couvreur's algorithm [3]

The ASCC algorithm

```
procedure couv(s_I) {
    count: integer := 0;
    roots: stack⟨pair⟨state, set⟨integer⟩⟩⟩ := ∅
    active: stack⟨state⟩ := ∅
    call couv_dfs(s_I)
}

procedure couv_dfs(s) {
    count := count + 1
    s.dfsnum := count
    s.current := true
    roots.push(s, A(s))
    active.push(s)
    for (all t successors of s) {
        if (t.dfsnum = 0) then call couv_dfs(t)
        else if (t.current) {
            B: set of integers := ∅
            repeat {
                (u, C) := roots.pop()
                B := B ∪ C
                if (B = K) then report cycle
            } until (u.dfsnum ≤ t.dfsnum)
        }
    }
    if (roots.top() = (s, _)) {
        roots.pop()
        repeat {
            u: state := active.pop()
            u.current := false
        } until (u = s)
    }
}
```

How it works

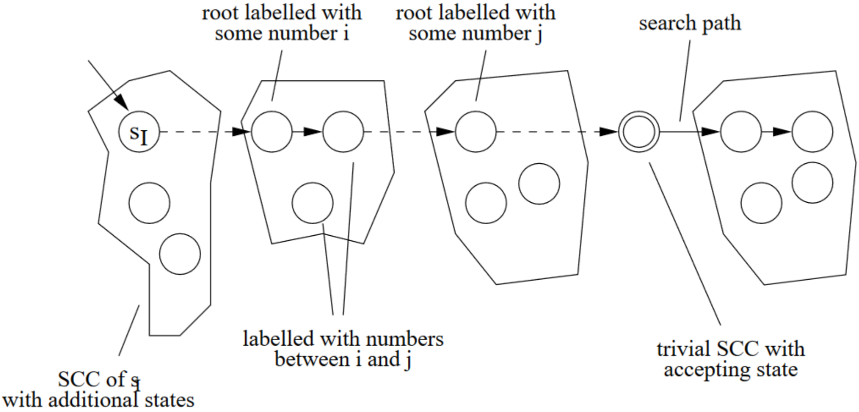

Figure: Shape of the active graph taken from [1]

Why it is still not very good

- ASCC (as described) does not provide a clear way to determine the violating trace.
- Converting from recursive to iterative (even by just simulating the recursion) would immediately give us the trace leading to the SCC.
- On the programming side this and a few other things need to be cleaned up.
- The implementation of fairness conditions is still missing.
- On the research side optimizations (partial order reduction) are still missing.

Bibliography

[1] Andreas Gaiser and Stefan Schwoon. *Comparison of Algorithms for Checking Emptiness on Buechi Automata*. 2009. DOI: 10.48550/ARXIV.0910.3766. URL: https://arxiv.org/abs/0910.3766.

[2] Ivana Cerna and Radek Pelánek. "Relating Hierarchy of Temporal Properties to Model Checking". In: vol. 2747. Aug. 2003, pp. 318–327. ISBN: 978-3-540-40671-6. DOI: 10.1007/978-3-540-45138-9_26.

[3] Jean-Michel Couvreur. "On-the-Fly Verification of Linear Temporal Logic.". In: Sept. 1999, pp. 253–271. ISBN: 978-3-540-66587-8. DOI: 10.1007/3-540-48119-2_16.