Transformation of LTL Formulas Into Automata
Master Thesis Topic

Sütő Ágoston

Research Institute for Symbolic Computation
Thesis supervisor: Prof. Wolfgang Schreiner

March 8, 2022
Previously...

Discussed last time:

- What is model checking
- RISCAL software system
- Kripke-structures and LTL
- Basic ideas behind the automaton-based model checking algorithm
- Goals of the thesis
Next up

Will be discussed today:

- Labelled generalized Büchi automata
- Concrete details for the automaton-based approach
- Preliminary steps
- Algorithm for constructing the automaton
- Algorithm for emptiness check for a language
Definition

A Kripke-structure $K$ over a set of atomic propositions $\mathcal{A}$ is defined as the tuple $(S, I, T, \mathcal{L})$ consisting of the following components:

- a set of states $S$
- a set of initial states $I \subseteq S$, $I \neq \emptyset$
- a total transition relation $T \subseteq S \times S$
- a labelling function $\mathcal{L}: S \rightarrow \mathcal{P}(\mathcal{A})$

Definition

A trace $\pi$ is a finite or infinite sequence of states of a Kripke-structure, such that $\forall i: s_i \xrightarrow{T} s_{i+1}$

$\pi = (s_0, s_1, ...)$
The alphabet of LTL consists of atomic propositions $A$, the standard logical operators ($\neg$, $\lor$, $\land$ etc.) and special temporal operators $X$ (next), $F$ (finally), $G$ (globally), and $U$ (until). The language of LTL formulas is defined inductively as follows:

- If $p$ is an atomic proposition, then it is an LTL formula
- If $g$ and $h$ are LTL formulas, then $\neg g$, $g \lor h$, $g \land h$ etc. are LTL formulas
- If $g$ and $h$ are LTL formulas, then $Xg$, $Fg$, $Gg$, and $gUh$ are LTL formulas.
The semantics of LTL for infinite paths $\pi$ of a Kripke-structure $K = (S, I, T, L)$ are defined as follows:

- $\pi \models p$ iff $p \in L(\pi(0))$
- $\pi \models \neg g$ iff $\pi \not\models g$
- $\pi \models g \lor h$ iff $\pi \models g$ or $\pi \models h$
- $\pi \models g \land h$ iff $\pi \models g$ and $\pi \models h$
- $\pi \models Xg$ iff $\pi^1 \models g$
- $\pi \models Fg$ iff $\exists i \in \mathbb{N}: \pi^i \models g$
- $\pi \models Gg$ iff $\forall i \in \mathbb{N}: \pi^i \models g$
- $\pi \models g U h$ iff $\exists i \in \mathbb{N}: \pi^i \models h \land \forall j \in \mathbb{N}: j < i \rightarrow \pi^j \models g$
Definition

A *labelled Büchi automaton* is defined as the tuple \((S, I, \Sigma, \mathcal{L}, T, F)\) consisting of the following components:

- a finite set of states \(S\)
- a set of initial states \(I \subseteq S, I \neq \emptyset\)
- an input alphabet \(\Sigma\)
- a labelling of the states \(\mathcal{L} : S \rightarrow 2^\Sigma\)
- a transition relation \(T \subseteq S \times S\)
- a set of accepting states \(F\)
Definition

An accepting execution $\sigma$ of a Büchi automaton $A = (S, I, \Sigma, L, T, F)$ is an infinite sequence of states $\sigma = s_0 s_1 s_2 \ldots \in S^\omega$ such that $s_0 \in I$ and there exists at least one state $s \in F$ which appears infinitely often in $\sigma$. 
Definition

An accepting execution $\sigma$ of a Büchi automaton $\mathcal{A} = (S, I, \Sigma, L, T, F)$ is an infinite sequence of states $\sigma = s_0s_1s_2... \in S^\omega$ such that $s_0 \in I$ and there exists at least one state $s \in F$ which appears infinitely often in $\sigma$.

Definition

A Büchi automaton $\mathcal{A} = (S, I, \Sigma, L, T, F)$ accepts a word $w = a_0a_1a_2... \in \Sigma^\omega$ if there exists an accepting execution $\sigma = s_0s_1s_2... \in S^\omega$ such that for each $i \geq 0$, $a_i \in L(s_i)$.
Definition

A labelled generalized Büchi automaton (LGBA) consists of the same components \((S, I, \Sigma, \mathcal{L}, T, \mathcal{F})\) as a simple labelled Büchi automaton, except that the accepting set is replaced by a set of accepting sets \(\mathcal{F} \subseteq 2^S\), \(\mathcal{F} = \{F_1, F_2, \ldots, F_n\}\).
Labelled generalized Büchi automaton

**Definition**

A *labelled generalized Büchi automaton* (LGBA) consists of the same components \( (S, I, \Sigma, \mathcal{L}, T, \mathcal{F}) \) as a simple labelled Büchi automaton, except that the accepting set is replaced by a set of accepting sets \( \mathcal{F} \subseteq 2^S \), \( \mathcal{F} = \{F_1, F_2, \ldots, F_n\} \).

**Definition**

An *accepting execution* \( \sigma \) of an LGBA \( A = (S, I, \Sigma, \mathcal{L}, T, \mathcal{F}) \) is an infinite sequence of states \( \sigma = s_0s_1s_2\ldots \in S^\omega \) such that \( s_0 \in I \) and for each \( i \geq 0 \), \( s_i \rightarrow s_{i+1} \), and for each acceptance set \( F_j \in \mathcal{F} \) there exists at least one state \( s_j \in F_j \) which appears infinitely often in \( \sigma \).
Useful theorems

**Proposition**

Given two LGBA \( A_i = (S_i, I_i, \Sigma, \mathcal{L}_i, T_i, \mathcal{F}_i) \), \( i \in \{1, 2\} \), there exists an LGBA \( A = (S, I, \Sigma, \mathcal{L}, T, \mathcal{F}) \) such that \( \mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2) \).

**Proposition**

Given an LGBA \( A = (S, I, \Sigma, \mathcal{L}, T, \mathcal{F}) \) there exists a Büchi automaton \( A' = (S', I', \Sigma, \mathcal{L}', T', \mathcal{F}') \) such that \( |S'| = |S| \times k \), where \( k = |\mathcal{F}| \) and \( \mathcal{L}(A) = \mathcal{L}(A') \).
The big picture

**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 $\pi$ of $K$ such that $\pi \not\models f$. 

1. Negate the formula and preprocess it
2. Transform this formula into an LGBA $A$
3. Given the Kripke-structure $K = (S, I, T, \mathcal{L})$ of the system, construct the LGBA $A_K = (S, I, 2P, \mathcal{L}', T, \emptyset)$ with $\mathcal{L}'(s) = \{\mathcal{L}(s)\}$ for any $s \in S$.
4. Construct the automaton which accepts the intersection of the languages of $A \neg f$ and $A_K$
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.
The big picture

**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 $\pi$ of $K$ such that $\pi \not\models f$.

1. Negate the formula and preprocess it
2. Transform this formula into an LGBA $A_{\neg f}$
3. Given the Kripke-structure $K = (S, I, T, \mathcal{L})$ of the system, construct the LGBA $A_K = (S, I, 2^P, \mathcal{L}', T, \emptyset)$ with $\mathcal{L}'(s) = \{\mathcal{L}(s)\}$ for any $s \in S$.
4. Construct the automaton which accepts the intersection of the languages of $A_{\neg f}$ and $A_K$
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.
History of the translation algorithm


- Later papers by Vardi, such as [2] improve the theoretical aspects of the construction

- In these cases the size of the automaton guaranteed to be exponential in the size of the formula

- An improved algorithm developed by R. Gerth et al. in 1996 [3] will be presented here
Preprocessing step

- Introduce new temporal operator $V$, defined as the dual of $U$:
  $$f V g \equiv \neg (\neg f U \neg g).$$

- Replace the temporal operators $F$ and $G$ using $Fp \equiv \top U p$ and $Gp \equiv \bot V p$.

- Convert $\neg f$ into negation normal form by replacing $\Rightarrow$ and $\Leftrightarrow$ with $\neg$, $\land$ and $\lor$, and pushing the negations to in front of propositions using De Morgan’s laws.

Example:

$$\neg (Fp \Rightarrow Gq) \equiv \neg (\top U p \Rightarrow \bot V q) \equiv \neg (\neg (\top U p) \lor \bot V q) \equiv (\top U p) \land \neg (\bot V q) \equiv (\top U p) \land (\top U \neg q)$$
LTL to LGBA algorithm, main ideas

- Two step construction: first a directed graph, then it is converted into an automaton

- Uses the expansion formulas of temporal operators:
  - $Xp$ holds if $p$ holds in the next state
  - $p \land q$ holds if $p$ and $q$ hold in the current state
  - $p \lor q$ holds if either $p$ or $q$ holds in the current state
  - $p U q$ holds if either $q$ holds in the current state or $p$ holds in the current state and $p U q$ holds in the next state
  - $p V q$ holds if either both $p$ and $q$ hold in the current state or if $q$ holds in the current state and $p V q$ holds in the next state
LTL to LGBA algorithm

record GraphNode = {
    incoming: set of GraphNode references,
    new: set of formulas,
    old: set of formulas,
    next: set of formulas,
}

function createGraph(f: formula) {
    return expand({incoming: {init}, new: {f}, old: {}, next: {}}, {})
}

function expand(node: GraphNode, nodesSet: set of GraphNode) {
    if (node.new is empty) {
        if (there is a GraphNode n ∈ nodesSet with
            n.old = node.old and n.next = node.next) {
            n.incoming = n.incoming ∪ node.incoming
            return nodesSet
        } else {
            return expand({incoming: {node}, new: node.next,
                old: {}, next: {}}, nodesSet ∪ {node})
        }
    } else {
        let f ∈ node.new
        node.new.remove(f)
        if (f = p_i or f = ¬p_i or f = ⊤ or f = ⊥) {
            if (f = ⊥ or ¬f ∈ node.old) {
                return nodesSet
            } else {
                node.old = node.old ∪ {f}
                return expand(node, nodesSet)
            }
        }
        ...
    }
}
... else if (f = X g) {
    return expand({
        incoming: node.incoming,
        new: node.new, old: node.old ∪ {f}, next: node.next ∪ {g}
    }, nodesSet)
} else if (f = g ∧ h) {
    return expand({
        incoming: node.incoming,
        new: node.new ∪ ({g, h} \ node.old),
        old: node.old ∪ {f}, next: node.next
    }, nodesSet)
} else if (f = g ∨ h or f = g U h or f = g V h) {
    let node1 = {incoming: node.incoming,
                 new: node.new ∪ (new1(f) \ node.old),
                 old: node.old ∪ {f}, next = node.next ∪ next1(f) }
    let node2 = {incoming: node.incoming,
                 new: node.new ∪ (new2(f) \ node.old),
                 old: node.old ∪ {f}, next = node.next }
    return expand(node2, expand(node1, nodesSet))
}

<table>
<thead>
<tr>
<th></th>
<th>new1(f)</th>
<th>next1(f)</th>
<th>new2(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>g ∨ h</td>
<td>{g}</td>
<td>{}</td>
<td>{h}</td>
</tr>
<tr>
<td>g U h</td>
<td>{g}</td>
<td>{g U h}</td>
<td>{h}</td>
</tr>
<tr>
<td>g V h</td>
<td>{h}</td>
<td>{g V h}</td>
<td>{g, h}</td>
</tr>
</tbody>
</table>
Example execution

Processing is started with

```
node0 = { incoming: [init], new: [p U q] }
```
node0 is split into
node1 = { incoming: [init], new: [p],
          old: [p U q], next: [p U q] }
and node2 = { incoming: [init], new: [q],
             old: [p U q] }
Example execution

Moving literal p in node1 to the set old.
Result: node1 = {
  incoming: [init],
  old: [p U q, p],
  next: [p U q]
}
node1 has no more properties in the set new.

New child node is added: node3 = { incoming: [node1],
    new: [p U q] }
Example execution

node1
new: ∅
old: {p U q, p}
next: {p U q}

node5
new: {q}
old: {p U q}
next: ∅

node2
new: {q}
old: {p U q}
next: ∅

node4
new: {p}
old: {p U q}
next: {p U q}

node3 is split into
node4 = { incoming: [node1], new: [p], old: [p U q],
next: [p U q] }
and node5 = { incoming: [node1], new: [q],
old: [p U q] }

Sütő Ágoston (RISC)
Example execution

Moving literal $p$ in node4 to the set $old$.
Result: $node4 = \{\text{incoming: [node1]}, \text{old: [p U q, p]}, \text{next: [p U q]} \}$
node4 is equivalent to node1. Merging incoming edges.

Result: node1 = { incoming: [node1, init],
old: [p U q, p], next: [p U q] }
Example execution

Moving literal \( q \) in node5 to the set old.
Result: node5 = { incoming: [node1], old: [\( p \cup q, q \) ] }
Example execution

node1
new: \emptyset
old: \{ p \lor q, p \}
next: \{ p \lor q \}

node2
new: \{ q \}
old: \{ p \lor q \}
next: \emptyset

node5
new: \emptyset
old: \{ p \lor q, q \}
next: \emptyset

node6
new: \emptyset
old: \emptyset
next: \emptyset

node5 has no more properties in the set new.
New child node is added: node6 = \{ incoming: [node5] \}
Example execution

node1
new: ∅
old: {p U q, p}
next: {p U q}

node2
new: {q}
old: {p U q}
next: ∅

node5
new: ∅
old: {p U q, q}
next: ∅

node6
new: ∅
old: ∅
next: ∅

node7
new: ∅
old: ∅
next: ∅

node6 has no more properties in the set new.

New child node is added: node7 = { incoming: [node6] }
Example execution

node1
new: ∅
old: \{p \cup q, p\}
next: \{p \cup q\}

node2
new: \{q\}
old: \{p \cup q\}
next: ∅

node5
new: ∅
old: \{p \cup q, q\}
next: ∅

node6
new: ∅
old: ∅
next: ∅

node7 is equivalent to node6. Merging incoming edges.
Result: node6 = \{ incoming: [node6, node5] \}
Example execution

Moving literal $q$ in node2 to the set old.
Result: $\text{node2} = \{ \text{incoming: [init], old: [p U q, q] } \}$
Example execution

node1
new: ∅
old: \{p U q, p\}
next: \{p U q\}

node5
new: ∅
old: \{p U q, q\}
next: ∅

node6
new: ∅
old: ∅
next: ∅

node2 is equivalent to node5. Merging incoming edges.
Result: node5 = { incoming: [node1, init],
old: [p U q, q] }
LTL to LGBA algorithm

Last step: transforming the graph created by the algorithm to the LGBA

$$\mathcal{A}_f = (S, I, \Sigma, \mathcal{L}, T, \mathcal{F})$$

- $S$ is the set of nodes returned by the algorithm.
- $I$ is the set of all the nodes $n$ for which $\text{init} \in n.\text{incoming}$.
- $(n_1, n_2) \in T$ iff $n_1 \in n_2.\text{incoming}$.
- $\Sigma = 2^P$

Let $\text{pos}(n) = \{ p \in P : p \in n.\text{old} \}$ and $\text{neg}(n) = \{ p \in P : \neg p \in n.\text{old} \}$ then

$$\mathcal{L}(n) = \{ l \in \Sigma : \text{pos}(n) \in l \land \text{neg}(n) \cap l = \emptyset \}.$$  

- $\mathcal{F}$ consists of sets $F_i \subseteq S$ for each subformula of $f$ of the form $g \mathbf{U} h$ such that $n \in F_i$ if $g \mathbf{U} h \notin n.\text{old}$ or $h \in n.\text{old}$. 
Generated automaton

Figure: LGBA corresponding to the formula $p \mathbf{U} q$
Emptiness check for Büchi automata

```javascript
function isLanguageEmpty(initialStates, acceptingStates) {
    let S1 = stack(initialStates);
    let S2 = stack();
    let M1 = set();
    let M2 = set();

    while (S1.length > 0) {
        let 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.length > 0) {
                    let 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();
                    }
                }
            } else {
                return true;
            }
        }
    }
    return true;
}
```
Next steps

- Implement the full translation algorithm in RISCAL (with some small extra optimizations mentioned in the paper)
- Implement the emptiness check in an on-the-fly manner
- Take measurements of the resulting implementation and compare it to established model checker(s)
- Implement more sophisticated optimizations, such as partial order reduction (planned for next presentation)
Bibliography

