IA169 System Verification and Assurance

LTL Model Checking
(continued)

Jiří Barnat

Reminder

Problem

- Kripke structure M
- LTL formula φ
- $M \models \varphi$?

Solution Based on Büchi Automata

- A_{sys} – automaton accepting all system runs
- $A_{\neg\varphi}$ – automaton accepting all runs violating φ
- $L(A_{sys}) \cap L(A_{\neg\varphi}) = L(A_{sys} \times A_{\neg\varphi})$
- $L(A_{sys} \times A_{\neg\varphi}) \neq \emptyset \iff$ system exhibits invalid run
- $L(A_{sys} \times A_{\neg\varphi}) = \emptyset \iff M \models \varphi$

Algorithm for Detection of Accepting Cycles

Algorithm Input

- Product Büchi automaton given implicitly
  - $|F|\_init()$ — Returns initial state of automaton.
  - $|F|\_succs(s)$ — Gives immediate successors of a given state.
  - $|Accepting|(s)$ — Gives whether a state is accepting or not.

Algorithm Output

- Present/ Not present
- Counterexample.

Algorithm

- Two nested depth-first search procedures – **Nested DFS**.
- Outer procedure detects accepting states, inner procedure checks for each accepting state if it is self-reachable (lies on a cycle).

Section

Detection of Accepting Cycles

Detection of Accepting Cycles

Problem

- Let $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$ be a Büchi automaton.
- Is the language accepted by $\mathcal{A}$ non-empty?

Reduction to Accepting Cycle Detection Problem

- Let $G = (S, E)$, where
  $$E = \{(u, v) \in S \times S \mid \exists a \in \Sigma \text{ such that } v \in \delta(u, a)\}$$
  be a graph of a Büchi automaton.
- $L(\mathcal{A})$ is non-empty if and only if the graph of the automaton $\mathcal{A}$ contains reachable accepting cycle, i.e. a cycle whose at least one vertex v corresponds to an accepting state ($v \in F$), and is, at the same time, reachable from the initial state ($(s_0, v) \in E^*$).

Detection of Accepting Cycles

Algorithmic Solution

1) Identify all reachable accepting states in the graph of Büchi automaton. (Outer procedure.)
2) Check for every such the state that is not self-reachable (Inner procedure.)

Reachability in Directed Graph

- The standard graph algorithm.
- To compute the set of reachable vertices (or accepting vertices) can be done in in time $\mathcal{O}(|V| + |E|)$.
- Using the standard algorithm, accepting cycle detection can be done in time $\mathcal{O}(|V| + |E| + |F|(|V| + |E|))$.
- Clever techniques can employ depth-first search post-order to reduce the time complexity to $\mathcal{O}(|V| + |E|)$.

Depth-First Search Procedure

```
proc Reachable($V$,$E$,$v_0$)
  $Visited$ = $\emptyset$
  DFS($v_0$)
  return ($Visited$)
end

proc DFS($vertex$)
  if $vertex \notin Visited$
    then /* Visits $vertex$ */
      $Visited$ := $Visited \cup \{vertex\}$
      foreach { $v$ | ($vertex$,$v$)$\in$ E } do
        DFS($v$)
      od
      /* Backtracks from $vertex$ */
  fi
```

Colour Notation in DFS

Observation

- When running DFS on a graph all vertices can be classified into one of the three following categories (denoted with colours).

Colour Notation for Vertices

- White vertex – Has not been visited yet.
- Gray vertex - Visited, but yet not backtracked.
- Black vertex - Visited and backtracked.

Recursion Stack

- Gray vertices form a path from the initial vertex to the vertex that is currently processed by the outer procedure.

Properties of DFS, $G = (V, E)$ a $v_0 \in V$

Observation

- If two distinct vertices v_1, v_2 satisfy that
    - $(v_0, v_1) \in E^*$,
    - $(v_1, v_1) \notin E^+$,
    - $(v_1, v_2) \in E^+$.
- Then procedure $|DFS|(v_0)$ backtracks from vertex v_2 before it backtracks from vertex v_1.

DFS post-order

- If $(v, v) \notin E^+$ and $(v_0, v) \in E^*$, then upon the termination of sub-procedure $|DFS|(v)$, called within procedure $|DFS|(v_0)$, all vertices u such that $(v, u) \in E^+$ are visited and backtracked.

Detection of Accepting Cycles in $\mathcal{O}(|V| + |E|)$

Observation

- If a sub-graph reachable from a given accepting vertex does not contain accepting cycle, then no accepting cycle starting in an accepting state outside the sub-graph can reach the sub-graph.

The Key Idea

- Execute the inner procedures in a bottom-up manner.
- The inner procedures are called in the same order in which the outer procedure backtracks from accepting states, i.e. the ordering of calls follows a DFS post-order.

Detection of Accepting Cycles in $\mathcal{O}(|V| + |E|)$

```
proc Detection_of_accepting_cycles
  $Visited$ := $\emptyset$
  DFS($v_0$)
end

proc DFS($vertex$)
  if ($vertex$) $\notin$ $Visited$
    then $Visited$ := $Visited \cup \{vertex\}$
    foreach {$s$ | ($vertex$,$s$) $\in$ $E$} do
      DFS($s$)
    od
    if IsAccepting($vertex$)
      then DetectCycle($vertex$)
    fi
  fi
end
```

Detection of Accepting Cycles in $\mathcal{O}(|V| + |E|)$

Assumption On Early Termination

- The inner procedure reports the accepting cycle and terminates the whole algorithm if called for an accepting vertex that lies on an accepting cycle.

Consequences

- If the inner procedure called for an accepting vertex v reports no accepting cycle, then there is no accepting cycle in the graph reachable from vertex v.

Detection of Accepting Cycles in $\mathcal{O}(|V| + |E|)$

Linear Complexity of Nested DFS Algorithm

- Employing DFS post-order it follows that vertices that have been visited by previous invocation of inner procedure may be safely skipped in any later invocation of the inner procedure.

$\mathcal{O}(|V| + |E|)$ Algorithm

1) Nested procedures are called in DFS post-order as given by the outer procedure, and are limited to vertices not yet visited by inner procedure.
2) All vertices are visited at most twice.

Detecting Cycles in Inner Procedures

Theorem

- If the immediate successor to be processed by an inner procedure is grey (on the stack of the outer procedure), then the presence of an accepting cycle is guaranteed.

Application

- It is enough to reach a vertex on the stack of the outer procedure in the inner procedure in order to report the presence of an accepting cycle.

$\mathcal{O}(|V| + |E|)$ Algorithm

```
proc Detection_of_accepting_cycles
  $Visited$ := $Nested$ := $in\_stack$ := $\emptyset$
  DFS($v_0$)
  Exit("Not Present")
end
```

```
proc DFS($vertex$)
  if ($vertex$) $\notin$ $Visited$
    then $Visited$ := $Visited$ $\cup$ {$vertex$}
    $in\_stack$ := $in\_stack$ $\cup$ {$vertex$}
    foreach {$s$ | ($vertex$,$s$) $\in$ $E$} do
      DFS($s$)
    od
    if IsAccepting($vertex$)
      then DetectCycle($vertex$)
    fi
    $in\_stack$ := $in\_stack$ $\setminus$ {$vertex$}
  fi
end
```

```
proc DetectCycle ($vertex$)
  if $vertex$ $\notin$ $Nested$
    then $Nested$ := $Nested$ $\cup$ {$vertex$}
    foreach {$s$ | ($vertex$,$s$) $\in$ $E$} do
      if $s$ $\in$ $in\_stack$
        then WriteOut($in\_stack$)
          Exit("Present")
        else DetectCycle($s$)
      fi
    of
  fi
end
```

Time and Space Complexity

Outer Procedure

- Time: $\mathcal{O}(|V| + |E|)$
- Space: $\mathcal{O}(|V|)$

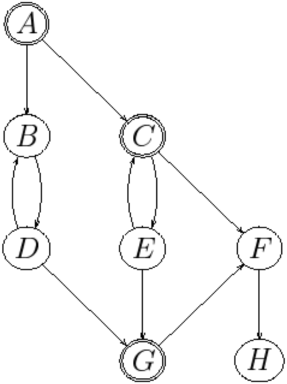
Inner Procedures

- Time (overall): $\mathcal{O}(|V| + |E|)$
- Space: $\mathcal{O}(|V|)$

Complexity

- Time: $\mathcal{O}(|V| + |E| + |V| + |E|) = \mathcal{O}(|V| + |E|)$
- Space: $\mathcal{O}(|V| + |V|) = \mathcal{O}(|V|)$

Nested DFS – Example



- 1st DFS: A,B,D,B,G,F,H,H,F,G
  1st DFS stack: A,B,D,G
  visited: A,B,D,F,G,H / –
- 2nd DFS: G,F,H,H,F,G
  visited: A,B,D,F,G,H / F,G,H
- 1st DFS: G,D,B,C,E,C,G,E,F,C
  1st DFS stack: A,C
  visited: all / F,G,H
- 2nd DFS: C,E,C
  counterexample: A,C,E,C

visited state   backtrack non-accepting state   backtrack accepting state

Section

Classification of Büchi Automata

Sub-Classes of Büchi Automata

Terminal Büchi Automata

- All accepting cycles are self-loops on accepting states labelled with `true`.

Weak Büchi Automata

- Every strongly connected component of the automaton is composed either of accepting states, or of non-accepting states.

Impact on Verification Procedure

Automaton $A_{\neg\varphi}$

- For a number of LTL formulae φ is $A_{\neg\varphi}$ terminal or weak.
- $A_{\neg\varphi}$ is typically quite small.
- Type of $A_{\neg\varphi}$ can be pre-computed prior verification.
- Types of components of $A_{\neg\varphi}$
  - Non-accepting – Contains no accepting cycles.
  - Strongly accepting – Every cycle is accepting.
  - Partially accepting – Some cycles are accepting and some are not.

Product Automaton

- The graph to be analysed is a graph of product automaton $A_S \times A_{\neg\varphi}$.
- Types of components of $A_S \times A_{\neg\varphi}$ are given by the corresponding components of $A_{\neg\varphi}$.

Impact on Verification Procedure – Terminal BA

$A_{\neg\varphi}$ is terminal Büchi automaton

- For the proof of existence of accepting cycle it is enough to proof reachability of any state that is accepting in $A_{\neg\varphi}$ part.
- Verification process is reduced to the reachability problem.

„Safety" Properties

- Those properties φ for which $A_{\neg\varphi}$ is a terminal BA.
- Typical phrasing: „Something bad never happens."
- Reachability is enough to proof the property.

Impact on Verification Procedure – Weak BA

$A_{\neg\varphi}$ is weak Büchi automaton

- Contains no partially accepting components.
- For the proof of existence of accepting cycle it is enough to proof existence of reachable cycle in a strongly accepting component.
- Can be detected with a single DFS procedure.
- Time-optimal algorithm exists that does not rely on DFS.

„Weak" LTL Properties

- Those properties φ for which $A_{\neg\varphi}$ is a weak BA.
- Typically, responsiveness: $G\,(a \implies F\,(b))$.

Classification of LTL Properties

Classification

- Every LTL formula belongs to one of the following classes:
  Reactivity, Recurrence, Persistance, Obligation, Safety, Guarantee

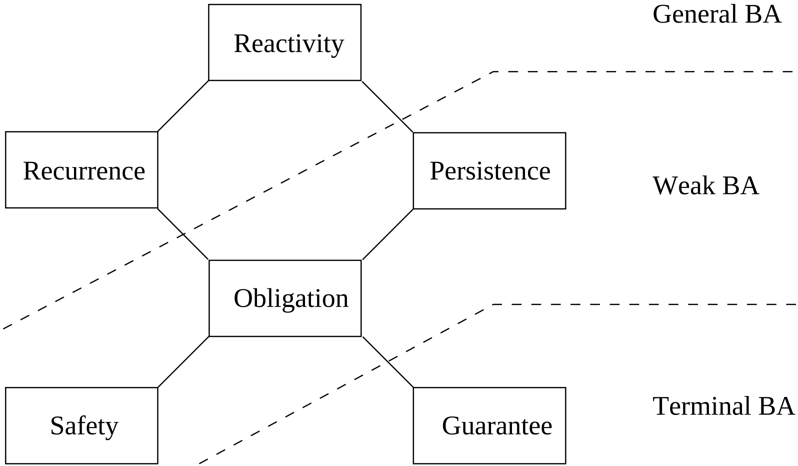
Interesting Relations

- Guarantee class properties can be described with a terminal Büchi automaton.
- Persistance, Obligation, and Safety class properties can be described with a weak Büchi automaton.

Negation in Verification Process ($\varphi \mapsto A_{\neg\varphi}$)

- $\varphi \in$ Safety $\iff$ $\neg\varphi \in$ Guarantee.
- $\varphi \in$ Recurrence $\iff$ $\neg\varphi \in$ Persistance.

Classification of LTL Properties



Reactivity

Recurrence

Persistence

Obligation

Safety

Guarantee

General BA

Weak BA

Terminal BA

Section

Fighting State Space Explosion

State Space Explosion Problem

What is State Space Explosion

- System is usually given as a composition of parallel processes.
- Depending on the order of execution of actions of parallel processes various intermediate states emerge.
- The number of reachable states may be up to exponentially larger than the number of lines of code.

Consequence

- Main memory cannot store all states of the product automaton as they are too many.
- Algorithms for accepting cycle detection suffer for lack of memory.

Some Methods to Fight State Space Explosion

State Compression

- Lossless compression.
- Lossy compression – Heuristics.

On-The-Fly Verification

Symbolic Representation of State Space

Reduced Number of States the Product Automaton

- Introduction of atomic blocks.
- Partial order on execution of process actions.
- Avoid exploration of symmetric parts.

Parallel and Distributed Verification

On-The-Fly Verification

Observation

- Product automaton graph is defined implicitly with:
  - $|F|\_init()$ — Returns initial state of automaton.
  - $|F|\_succs(s)$ — Gives immediate successors of a given state.
  - $|Accepting|(s)$ — Gives whether a state is accepting or not.

On-The-Fly Verification

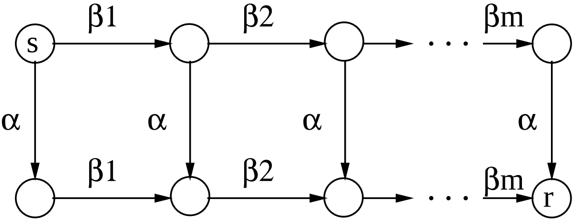
- Some algorithms may detect the presence of accepting cycle without the need of complete exploration of the graph.
- Hence, $\mathcal{M} \models \varphi$ can be decided without the full construction of $A_{sys} \times A_{\neg\varphi}$.
- This is referred to as to on-the-fly verification.

Partial Order Reduction

Example

- Consider a system made of processes A and B.
- A can do a single action α, while B is a sequence of actions β, e.g. $\beta_1, \ldots, \beta_m$.

Unreduced State Space:

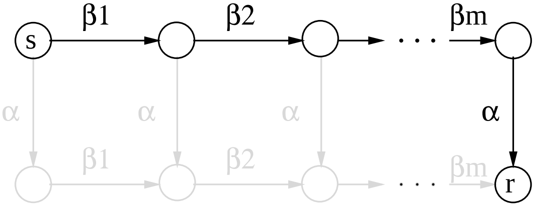


Property to be verifed: Reachability of state r.

Partial Order Reduction

Observation

- Runs $(\alpha\beta_1\beta_2\ldots\beta_m)$, $(\beta_1\alpha\beta_2\ldots\beta_m)$, …, $(\beta_1\beta_2\ldots\beta_m\alpha)$ are equivalent with respect to the property.
- It is enough to consider only a representative from the equivalence class, say, e.g. $(\beta_1\beta_2\ldots\beta_m\alpha)$.



- The representative is obtained by postponing of action α.

Partial Order Reduction

Reduction Principle

- Do not consider all immediate successor during state space exploration, but pick carefully only some of them.
- Some states are never generated, which results in a smaller state space graph.

Technical Realisation

- To pick correct but optimal subset of successors is as difficult as to generate the whole state space. Hence, heuristics are used.
- The reduced state space must contain an accepting cycle if and only if the unreduced state space does so.
- LTL formula must not use X operator (subclass of LTL).

Distributed and Parallel Verification

Principle

- Employ aggregate power of multiple CPUs.
- Increased memory and computing power.

Problem of Nested DFS

- Typical implementation relies on hashing mechanism, hence, the main memory is accessed extremely randomly. Should memory demands exceeds the amount of available memory, **thrashing** occurs.
- Mimicking serial Nested DFS algorithm in a distributed-memory setting is extremely slow. (Token-based approach).
- It unknown whether the DFS post-order can be computed by a time-optimal scale-able parallel algorithm (Still an open problem.)

Parallel Algorithms for Distributed-Memory Setting

Observation

- Instead of DFS other graph procedures are used.
- Tasks such as breadth-first search, or value propagation can be efficiently computed in parallel.
- Parallel algorithms do not exhibit optimal complexity.

	Complexity	Optimal	On-The-Fly
Nested DFS	O(V+E)	Yes	Yes
OWCTY			
general Büchi automata	O(V.(V+E))	No	No
weak Büchi automata	O(V+E)	Yes	No
MAP	O(V.V.(V+E))	No	Partially
OWCTY+MAP			
general Büchi automata	O(V.(V+E))	No	Partially
weak Büchi automata	O(V+E)	Yes	Partially

Section

Model Checking – Summary

Decision Procedure and State Space Explosion

Properties Validity

- Property to be verified may be violated by a single particular (even extremely unlikely) run of the system under inspection.
- The decision procedure relies on exploration of state space graph of the system.

State Space Explosion

- Unless thee are other reasons, all system runs have to be considered.
- The number of states, that system can reach is up to exponentially larger than the size of the system description.
- Reasons: Data explosion, asynchronous parallelism.

Advantages of Model Checking

General Technique

- Applicable to Hardware, Software, Embedded Systems, Model-Based Development, . . .

Mathematically Rigorous Precision

- The decision procedure results with $\mathcal{M} \models \varphi$, if and only if, it is the case.

Tool for Model Checking – Model Checkers

- The so called "Push-Button" Verification.
- No human participation in the decision process.
- Provides users with witnesses and counterexamples.

Disadvantages of Model Checking

Not Suitable for Everything

- Not suitable to show that a program for computing factorial really computes $n!$ for a given n.
- Though it is perfectly fine to check that for a value of 5 it always returns the value of 120.

Often Relies on Modelling

- Need for model construction.
- Validity of a formula is guaranteed for the model, not the modelled system.

Size of the State Space

- Applicable mostly to system with finite state space.
- Due to state space explosion, practical applicability is limited.

Verifies Only What Has Been Specified

- Issues not covered with formulae need not be discovered.

Practicals and Homework – 06

Practicals

- Code-based reachability analysis with DIVINE model checker.
- Verify `ring.cpp`.
- Find error in `fifo.cpp`.

Homework

- Analysis with DIVINE model checker on a more complex example (some homework from previous course on secure coding).