How to handle large finite-state systems

- Abstractions
- Symbolic Approach
- Partial Order Reduction
- Bounded MC
- Parallelization/Distribution
How to use Clusters of Workstations for Automated Verification (Distributed LTL Model-Checking)
Parallel and Distributed Model-Checking

Common Issues

- Which platform? – platform dependent algorithms
- How to divide the problem into sub-problems?
- New (often radically different) algorithms needed.
- Tools.
Main obstacle in practice:

ENORMOUS SIZE OF THE MODEL (SYSTEM)

*state explosion problem*

- Analysis is computationally intensive and/or storage-demanding application.
- The practical limitation is often the amount of the randomly accessed memory.
- Time is “elastic”, memory not!
Main obstacle in practice:

ENORMOUS SIZE OF THE MODEL (SYSTEM)
state explosion problem

- Analysis is computationally intensive and/or storage-demanding application.
- The practical limitation is often the amount of the randomly accessed memory.
- Time is “elastic”, memory not!
Automata Approach – Basic Principle

- Both the model (of the system) and the LTL formula are represented as Büchi automata.
- Given are automaton \( A \) and formula \( \varphi \). Construct \( B_{\neg \varphi} \) and \( A \times B_{\neg \varphi} \).
- The LTL model-checking problem “\( A \models \varphi ? \)” is reduced to is the language recognized by \( A \times B_{\neg \varphi} \) empty?
- \( BA \) \( A \) can be represented as a graph \( G_A \)
- \( L(A) \) is non-empty iff \( G_A \) has a reachable accepting cycle

Graph problem: Existence (detection) of a reachable accepting cycle in a graph
Automata Approach – Basic Principle

- Both the model (of the system) and the LTL formula are represented as Büchi automata.
- Given are automaton $A$ and formula $\varphi$. Construct $B_{\neg \varphi}$ and $A \times B_{\neg \varphi}$.
- The LTL model-checking problem “$A \models \varphi$?” is reduced to is the language recognized by $A \times B_{\neg \varphi}$ empty?
- BA $A$ can be represented as a graph $G_A$
- $L(A)$ is non-empty iff $G_A$ has a reachable accepting cycle

Graph problem: Existence (detection) of a reachable accepting cycle in a graph
The depth-first search (DFS) strategy provides suitable time (and space) efficient algorithms.

Nested DFS, SCC based (Tarjan)

**Nested DFS:**
- 1st procedure generates reachable states and discovers accepting states.
- 2nd (nested) procedure checks for accepting cycles.
- The nested procedure is started when the first one backtracks.
- The first search is suspended during the nested search.
- Only 2 additional bits needed.
- The time complexity is linear in the size of the graph \( O(n + m) \).
**Platform**
- Network of workstations (NOWs).
- No shared memory (combined memory).
- Communication by message passing.

**Problem Division**
- Enumerative approach (not symbolic).
- Graph given implicitly by \((F_{\text{init}}, F_{\text{successor}})\)
- Distributed data – **partition function** assigns states to workstations
Partition function assigns workstations to states

\[ \text{partition} : S \rightarrow N \]

Detection of (accepting) cycles is
- easy for cycles placed on one workstation and
- more difficult for cycles splitted among workstations.

Also important: spatial balance, fraction of cross-edges, temporal balance.

Two techniques
- Hash-based partition
- Property-driven partition
Partition function assigns workstations to states

\[ \text{partition} : S \rightarrow \mathcal{N} \]

Detection of (accepting) cycles is
- easy for cycles placed on one workstation and
- more difficult for cycles splitted among workstations.

Also important: spatial balance, fraction of cross-edges, temporal balance.

Two techniques
- Hash-based partition
- Property-driven partition
Partition function assigns workstations to states

\[ \text{partition} : S \rightarrow \mathcal{N} \]

Detection of (accepting) cycles is
- easy for cycles placed on one workstation and
- more difficult for cycles split among workstations.

**Also important:** spatial balance, fraction of cross-edges, temporal balance.

**Two techniques**
- Hash-based partition
- Property-driven partition
Hash-Based Partition

\[
P_1 \mid P_2 \mid \ldots \mid P_n \mid \neg \Phi
\]

state descriptor

\[
\begin{array}{ccccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & & & \\
\end{array}
\]
Hash-Based Partition

[Lerda, Sisto – SPIN 1999]

\[ P_1 \mid P_2 \mid \ldots \mid P_n \mid \neg \Phi \]

state

\begin{array}{cccccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0
\end{array}

window

state descriptor
Hash-Based Partition

[Lerda, Sisto – SPIN 1999]

\[ P_1 | P_2 | \ldots | P_n | \neg \Phi \]

\[
\begin{array}{cccccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\]

window

state descriptor

\[
\begin{array}{cccccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\]

hash+modulo

\[
\begin{array}{cccccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\]
Idea

- Product is a *synchronous product* of two automata \((A \times B_{\neg \varphi})\).
- Each state has two parts: the one given by the modeled system and the other one given by the negative claim automaton (representing negation of the verified formula).
- Each cycle in the product is determined by a cycle in the negative claim automaton (and system automaton).
- Use the decomposition of the negative claim automaton into maximal SCCs as a heuristic to partition the state space – eliminate division of accepting cycles.
Property-Driven Partition

[Barnat, Brim, Černá – VCL 2002]

**Idea**

- Product is a *synchronous product* of two automata \((A \times B_{-\varphi})\).
- Each state has two parts: the one given by the modeled system and the other one given by the negative claim automaton (representing negation of the verified formula).
- Each cycle in the product is determined by a cycle in the negative claim automaton (and system automaton).
- **Use the decomposition of the negative claim automaton into maximal SCCs as a heuristic to partition the state space – eliminate division of accepting cycles.**
$GFa \ldots$ infinitely often $a$

BA for $\neg GFa$

![Diagram showing states 1 and 2 with transitions and negated symbols $\neg a$.]
Property-Driven Partition

[Barnat, Brim, Černá – VCL 2002]

\[ \text{GF}a \ldots \text{infinitelly often } a \]

BA for \( \neg \text{GF}a \)

System

\begin{align*}
\text{System} & \quad \text{System} \\
1 & \xrightarrow{\neg a} 2 & c & \xrightarrow{\quad} \quad \\
2 & \xrightarrow{\neg a} & & b \\
& & a & \\
\end{align*}
Property-Driven Partition

[Barnat, Brim, Černá – VCL 2002]

$GFa \ldots$ infinitely often $a$

BA for $\neg GFa$

System

\begin{tikzpicture}
  \node[circle,draw] (1) at (0,0) {1};
  \node[circle,draw] (2) at (1,0) {2};
  \node[circle,draw] (a) at (2,1) {a};
  \node[circle,draw] (b) at (2,-1) {b};
  \node[circle,draw] (c) at (1,2) {c};

  \draw[->] (1) edge node {$\neg a$} (2);
  \draw[->] (2) edge node {$\neg a$} (1);
  \draw[->] (2) edge node {} (a);
  \draw[->] (2) edge node {} (b);
  \draw[->] (a) edge node {} (c);
  \draw[->] (b) edge node {} (c);
  \draw[->] (c) edge node {} (a);
\end{tikzpicture}
Property-Driven Partition

[Barnat, Brim, Černá – VCL 2002]

\[ GFa \ldots \text{infinitely often } a \]

BA for \( \neg GFa \)

System

\begin{align*}
\text{System} & \quad \text{BA for } \neg GFa \\
1 & \quad \neg a & 2 & \quad \neg a
\end{align*}
### Three types of sets of SCCs

- **F** – any cycle within the component contains at least one accepting state
- **P** – there is at least one accepting cycle and one non-accepting cycle within the component
- **N** – there is no accepting cycle within the component

- **N** – reachability (distributed arbitrarily)
- **F** – can be detected *without* using the nested search (place each component on a separate workstation)
- **P** – distributed detection of accepting cycles

Components of type **P** are rare in real applications.
Three types of sets of SCCs

- **F** – any cycle within the component contains at least one accepting state
- **P** – there is at least one accepting cycle and one non-accepting cycle within the component
- **N** – there is no accepting cycle within the component

- **N** – reachability (distributed arbitrarily)
- **F** – can be detected *without* using the nested search (place each component on a separate workstation)
- **P** – distributed detection of accepting cycles
- components of type **P** are *rare in real applications*
Property-Driven Partition

\[ GFa \ldots \text{infinitely often } a \]

BA for \( \neg GFa \)

System

![Diagram of a system with nodes and arrows representing state transitions and properties.]

\[ \neg a \]

\[ a \rightarrow \neg a \rightarrow a \]

\[ 1 \rightarrow \neg a \rightarrow 2 \]

\[ \text{N} \quad 1c \rightarrow 1a \rightarrow 1b \rightarrow N \]

\[ \text{F} \quad 2c \rightarrow 2a \rightarrow 2b \rightarrow F \]
Property-Driven Partition

$GFa \ldots$ infinitely often $a$

BA for $\neg GFa$

System

\[
\begin{array}{c}
1 \quad \neg a \quad 2 \quad \neg a \\
\end{array}
\]

\[
\begin{array}{c}
a \\
\end{array}
\]

\[
\begin{array}{c}
b \\
\end{array}
\]
Parallel Nested DFS?

- Searching for *reachable accepting cycles* in a graph.
- *Nested DFS* – memory efficient linear algorithm.
- Accepting cycles are recognized using DFS *postorder*.
- Postorder problem is P-complete – difficult to parallelize.
- Alternative algorithms needed.

Be Ready to Pay

The aim is to achieve higher degree of asynchronous parallelism even for the price of increasing worst-case time complexity and consuming more space.
Parallel Nested DFS?

- **Searching for reachable accepting cycles in a graph.**
- Nested DFS – memory efficient linear algorithm.
- Accepting cycles are recognized using DFS *postorder*.
- Postorder problem is P-complete – difficult to parallelize.
- Alternative algorithms needed.

Be Ready to Pay

The aim is to achieve higher degree of asynchronous parallelism even for the price of increasing worst-case time complexity and consuming more space.
new algorithms needed

- sequential solution: postorder – difficult to parallelize (PTIME)
- parallel solution: reachability – efficient parallelization (NC)

travel & propagate (repeated reachability)

Four groups – Six algorithms

- BFS instead of DFS
  [Maximal Predecessors, Back-Level Edges]
- SCC-based approaches
  [Elimination of SCCs – forth and back]
- Reduction to another problem
  [Negative Cycles]
- Additional data
  [Dependency Structure]
Maximal Accepting Predecessors – MAP


Idea

Each cycle must contain a state which is its own predecessor.
Algorithm

- Storing all accepting predecessors is expensive.
- Ordering on accepting states.
- Store maximal accepting predecessor only.
- Modified SSSP algorithm (propagates accepting predecessors - ID).
- Accepting cycle is discovered if an accepting state is propagated to the state itself.
- After stabilization the algorithm is recursively called on subgraphs defined by states labeled with the same ID.
- Finishes when there are no new subgraphs or accepting cycle is found.
ordering:

4 > 2 > ⊥
ordering:

$4 > 2 > \perp$
ordering:

\[ 4 > 2 > \bot \]
ordering:

$4 > 2 > \bot$
ordering:

4 > 2 > ⊥
ordering:

4 > 2 > ⊥
ordering:

2 > 4 > ⊥
ordering:

2 > 4 > ⊥
ordering:

$2 > 4 > \bot$
ordering:

\[ 2 > 4 > \bot \]
ordering:

\[ 2 \succ 4 \succ \bot \]
MAP – Example 2

ordering:

2 > 4 > ⊥

unmark and recompute
ordering:

\[ 2 > 4 > \perp \]
ordering:

2 > 4 > ⊥
ordering:

2 > 4 > ⊥
ordering:

\[ 2 > 4 > \bot \]
ordering:

2 > 4 > ⊥
Maximal Accepting Predecessors

**Comments**

- An accepting cycle in $G$ can be formed from vertices with the same maximal accepting predecessor only.
- A graph induced by the set of vertices having the same maximal accepting predecessor is called **predecessor subgraph**.
- Every cycle in the graph is completely included in one of the predecessor subgraphs.
- Re-computing the MAP function can be done in parallel for every predecessor subgraph.
- DFS gives optimal ordering – heuristics for “good” ordering.
Back-Level Edges Algorithm – BL

[Barnat, Brim, Chaloupka – ASE 2003]

- DFS: detection of cycles
- BFS: computing distances

**Idea**
Each cycle must contain a back-level edge, i.e. an edge with destination state having the same or smaller distance from source vertex than its source state.

**Algorithm**
- Discover all back-level edges (primary procedure) – level synchronized BFS
- Check if there is an edge that is part of a cycle (nested procedure)
Back-Level Edges Algorithm – BL

[Barnat, Brim, Chaloupka – ASE 2003]

- DFS: detection of cycles
- BFS: computing distances

Idea
Each cycle must contain a back-level edge, i.e. an edge with destination state having the same or smaller distance from source vertex than its source state.

Algorithm
- Discover all back-level edges (primary procedure) – level synchronized BFS
- Check if there is an edge that is part of a cycle (nested procedure)
Back-Level Edges Algorithm – BL

[Barnat, Brim, Chaloupka – ASE 2003]

- DFS: detection of cycles
- BFS: computing distances

**Idea**
Each cycle must contain a *back-level edge*, i.e. an edge with destination state having the same or smaller distance from source vertex than its source state.

**Algorithm**
- Discover all back-level edges (primary procedure) – level synchronized BFS
- Check if there is an edge that is part of a cycle (nested procedure)
Back-Level Edges Algorithm – Example
Back-Level Edges Algorithm – Example

level 0

level 1

level 2

level 3
Back-Level Edges Algorithm – Example
Back-Level Edges Algorithm – Example

Level 0

Level 1

Node A

Node B

Nodes 1, 2, 3, 4, 5, 6
Back-Level Edges Algorithm – Example

level 0

level 1

level 2
Back-Level Edges Algorithm – Example

level 0

level 1

level 2
Back-Level Edges Algorithm – Example
Back-Level Edges Algorithm – Example
Back-Level Edges Algorithm – Example

level 0

level 1

level 2

level 3

A

B

Jiří Barnat, Luboš Brim, Ivana Černá
Back-Level Edges Algorithm – Example

- - - - - - - - level 0

- - - - - - - - level 1

- - - - - - - - level 2

- - - - - - - - level 3

A

B

Jiří Barnat, Luboš Brim, Ivana Černá

Distributed Analysis of Large Systems
Back-Level Edges Algorithm – Example

stops after #bl edges ✓
Back-Level Edges Algorithm

Comments
- Accepting cycle detection (additional bit required)
- Partial Order Reduction

Partial Order Reduction
- Exploring subsets of successors of states (ample sets)
- Conditions ensuring correctness: C0 – C3
- C3-DFS: at least one fully explored state on each cycle
- C3-BFS: Full expansion of source states of back-level edges
Back-Level Edges Algorithm

Comments
- Accepting cycle detection (additional bit required)
- Partial Order Reduction

Partial Order Reduction
- Exploring subsets of successors of states (ample sets)
- Conditions ensuring correctness: C0 – C3
- C3-DFS: at least one fully explored state on each cycle
- C3-BFS: Full expansion of source states of back-level edges
SCC-Based Algorithm

Idea
Each accepting cycle is contained in a nontrivial strongly connected component which is reachable from the source vertex and contains an accepting vertex.

Algorithm
Remove all vertices without required properties.
- remove vertices which are not reachable from the source
- remove vertices which are not reachable from accepting vertices
- remove vertices which are not contained in any cycle
Idea

Each accepting cycle is contained in a nontrivial strongly connected component which is reachable from the source vertex and contains an accepting vertex.

Algorithm

Remove all vertices without required properties.

- remove vertices which are not reachable from the source
- remove vertices which are not reachable from accepting vertices
- remove vertices which are not contained in any cycle
SCC-Based Algorithm

Idea
Each accepting cycle is contained in a nontrivial strongly connected component which is reachable from the source vertex and contains an accepting vertex.

Algorithm
Remove all vertices \textit{without} required properties.
- remove vertices which are not reachable from the source
- remove vertices which are not reachable from accepting vertices
- remove vertices which are not contained in any cycle
SCC-Based Algorithm – Example

reachability from accepting states
SCC-Based Algorithm – Example

reachability from accepting states
not on any cycle
SCC-Based Algorithm – Example

not on any cycle
SCC-Based Algorithm – Example

reachability from accepting states
SCC-Based Algorithm – Example

reachability from accepting states
SCC-Based Algorithm – Example

not on any cycle
Idea

Main idea is the same: Each accepting cycle is contained in a nontrivial strongly connected component which is reachable from the source vertex and contains an accepting vertex.

Computing successors may be expensive. Store edges and check symmetric conditions.

Algorithm

Remove all vertices without required properties.
- remove vertices with out-degree 0
- remove vertices from which no accepting vertex is reachable
SCC-Based Algorithm Reversed – Example
SCC-Based Algorithm Reversed – Example

reachability of accepting states
SCC-Based Algorithm Reversed – Example

reachability of accepting states
SCC-Based Algorithm Reversed – Example

no output edges
SCC-Based Algorithms

Discussion

- Time complexity is $O(h \cdot (n + m))$
  - $n$ - number of vertices
  - $m$ - number of edges
  - $h$ - height of SCC quotient graph
- Almost linear complexity
- Only one external iteration for weak BA graphs
- Algorithm does not work on-the-fly
Negative Cycles Algorithm

Idea

- Reduce BA emptiness problem to another one which can be distributed more easily.
- Detecting negative cycles in the SSSP problem.

Negative cycles coincide with accepting cycles.
**Negative Cycles Algorithm**

Idea

- Reduce BA emptiness problem to another one which can be distributed more easily.
- **Detecting negative cycles in the SSSP problem.**

**Negative cycles coincide with accepting cycles.**
Negative Cycles – Example
Negative Cycles – Example
Negative Cycles – Example
Negative Cycles – Example

B

A

1 0

2 0

3 -1

5 -1

4 -1

6 -1

0 0 0 0

0 0 0 0

0 0 0 0

0 0 0 0

0 0 0 0
Negative Cycles – Example
Negative Cycles – Example
Negative Cycles – Example

Cycle in $G$
Negative Cycles – Example

walk to root

B

A

1

2

3

4

5

6
Negative Cycles Algorithm

Comments

- Strategies to detect feasibility
  - time out
  - walk to root (WTR)
  - subtree traversal
  - amortized search
- time complexity is $O(n^3/P)$
  - $P$ - number of processors, $n$ number of vertices
+ algorithm is comparable with nested-DFS algorithm on all graphs
+ algorithm is significantly better on graphs without negative cycles
Dependency Structure Algorithm – DepS

[Barnat, Brim, Stříbrná - SPIN 2001]

Idea

- Replace the graph by another smaller one:
  - Border states and accepting states only
  - Edges represent reachability (dependency) among these states.
- There is an accepting cycle in dependency graph iff there is an accepting cycle in the original graph.
- Dependency graph is distributed as well.
Dependency Structure Algorithm – Example

A

B

1

2

3

4

5

6
Dependency Structure Algorithm – Example
Dependency Structure Algorithm – Example

A

2 → 4
3 → 4

B

4 → 3
5 → 3
2
Dependency Structure:
- Each workstation maintains its own local dependency structure.
- Dynamic – vertices are added and removed.

Additional memory required:
\( O(n.r) \) on average, where \( r \) is the maximal out-degree and \( n \) is the number of states.

Any distributed cycle detection algorithm can be used.
Typical Result of Experiments

Speedup experiment for Lift 14 (SCC-Based Algorithm)

states: 34 291 712, transitions: 517 955 584

![Graph showing speedup time against number of workstations]
DiVinE – Distributed Verification Environment

**DiVinE Developer**
- Development and experimental evaluation of enumerative parallel and distributed model checking algorithms.
- Credible evaluation of existing enumerative algorithms with regard to their performance and characteristics under controlled conditions.

**DiVinE ToolSet**
- Ready-to-use distributed LTL model checker.
DiVinE – Architecture

User

DiVinE ToolSet

DiVinE Graphical Interface

Tool1  Tool2  Tool3

DiVinE Developer (library)

Model+Property  Output - Log Files

State Gen.  Algorithm  Reporter

Storage  Network  HW Monitor

Cluster GRID

Jiří Barnat, Luboš Brim, Ivana Černá  Distributed Analysis of Large Systems
Core problem of automata based LTL model-checking is the detection of reachable accepting cycles in the state space.

Classical depth first strategy provides a suitable approach to cycle detection in a sequential case.

The depth first search approach is difficult to distribute.

Other approaches to distributing LTL Model-Checking required.

Could allow to verify larger systems.
Some History of Distributed Verification

Distributed Reachability


- Caselli, ... (1994, 1995, ...) – reachability in PN (SIMD).
- Allmaier, ... (1997, 1999) – state space generation PN (shared memory).
- Nicol and Ciardo (1997) – stochastic models
- Stern and Dill (1997) – parallel version of the verifier Murφ.
- Haverkort, Bohnenkamp, and Bell (1998) – stochastic PN.
Distributed Reachability

- Lerda and Sisto (1999) – distributed reachability in SPIN.

Beyond the reachability

- Behrmann, Hune and Vaandrager (2000) – distributed Uppaal
- Barnat, Brim and Stříbrná (2001) – distributed LTL in SPIN.
Some History (cont.)

Distributed Reachability

- Lerda and Sisto (1999) – distributed reachability in SPIN.

Beyond the reachability

- Barnat, Brim and Stříbrná (2001) – distributed LTL in SPIN.
Beyond the reachability

- Haverkort and Bell (2002) – CTL
- Barnat, Brim, Chaloupka (2003) – back-level edges – LTL.
- ...