The MONA Project
Logic, Automata, and Program Verification

Anders Møller

http://www.brics.dk/~amoller/talks/dresden.pdf
The MONA Tool

The MONA tool

- transforms *formulas* into *finite-state automata*
- decides *validity* / provides *counterexamples* for the formulas by analyzing the automata
Example: A Mutual Exclusion Protocol

Hyman’s mutual exclusion algorithm:

```plaintext
while true do begin
  1  < noncritical section >
  2  b_i := true
  3  while ( k ≠ i ) do begin
  4    while ( b_{1:i} ) do skip
  5    k := i
  end
  6  < critical section >
  7  b_i := false
end
```

- Two processes executing (i=0 and i=1)
- Hyman’s claim: only one can be in the critical section at any time
Example: A Mutual Exclusion Protocol

Encoding the state:

```plaintext
var2 PC0', PC0'', PC0''', PC1', PC1'', PC1''', b0, b1, k;

pred p0_at_line_1(var1 t) = t \not\in PC0' \land t \not\in PC0'' \land t \not\in PC0''';
pred p0_at_line_2(var1 t) = t \not\in PC0' \land t \not\in PC0'' \land t \in PC0''';
...

pred b0_false(var1 t) = t \notin b0;
pred b0_true(var1 t) = t \in b0;
...

pred k_is_0(var1 t) = t \in k;
pred k_is_1(var1 t) = t \notin k;
```

declares variables that range over sets of natural numbers

encodes program counters

encodes state
Example: A Mutual Exclusion Protocol

Encoding the dynamics:

\[
\text{pred } p0\text{-proc\_step}(\text{var1 } t) = \\
(p0\_at\_line\_1(t) \Rightarrow p0\_at\_line\_2(t+1) \land \text{unchanged\_vars}(t)) \land \\
(p0\_at\_line\_2(t) \Rightarrow p0\_at\_line\_3(t+1) \land b0\_true(t+1) \land \\
\text{unchanged\_k}(t) \land \text{unchanged\_b1}(t)) \land \\
(p0\_at\_line\_3(t) \Rightarrow (\text{unchanged\_vars}(t) \land \\
(k\_is\_0(t) \Rightarrow p0\_at\_line\_6(t+1)) \land \\
(k\_is\_1(t) \Rightarrow p0\_at\_line\_4(t+1)))) \land \\
\ldots \\
(p0\_at\_line\_7(t) \Rightarrow p0\_at\_line\_1(t+1) \land b0\_false(t+1) \land \\
\text{unchanged\_k}(t) \land \text{unchanged\_b1}(t));
\]

\[
\text{pred Valid()} = p0\_at\_line\_1(0) \land p1\_at\_line\_1(0) \land \\
b0\_false(0) \land b1\_false(0) \land k\_is\_1(0) \land \\
(\forall t: ((p0\_proc\_step(t) \land \text{unchanged\_PC1}(t)) \\
\mid (p1\_proc\_step(t) \land \text{unchanged\_PC0}(t))));
\]
Example: A Mutual Exclusion Protocol

Checking mutual exclusion:

\[ \text{Valid()} \Rightarrow \forall t: \neg (p0_{\text{at\_line\_6}}(t) \land p1_{\text{at\_line\_6}}(t)) \]
Example: A Mutual Exclusion Protocol

After 0.5 seconds, MONA returns an automaton with 137 states

Reply from MONA automaton analysis:

A counterexample of least length (10) is:

- \( PC0' \)
- \( PC0'' \)
- \( PC0''' \)
- \( PC1' \)
- \( PC1'' \)
- \( PC1''' \)
- \( b0 \)
- \( b1 \)
- \( k \)

\[
\begin{align*}
PC0' & : 0 0 0 0 0 1 1 1 1 0 1 \\
PC0'' & : 0 0 0 1 1 0 0 0 1 0 \\
PC0''' & : 0 0 1 0 1 0 0 0 0 1 \\
PC1' & : 0 0 0 0 0 0 0 1 1 1 \\
PC1'' & : 0 0 0 0 0 0 1 0 0 0 \\
PC1''' & : 0 1 1 1 1 1 0 1 1 1 \\
b0 & : 0 0 0 1 1 1 1 1 1 1 \\
b1 & : 0 0 0 0 0 0 1 1 1 1 \\
k & : 0 0 0 0 0 0 0 0 1 1 \\
\end{align*}
\]

This **counterexample** shows the encoding of a valid run which violates the mutual-exclusion property!
Overview

• Introduction: verifying Hyman’s mutual exclusion algorithm

• **Mona**dic 2nd-order Logic on finite Strings (M2L-Str) / Weak monadic Second-order theory of 1 Successor (WS1S)

• Logic → Automata

• Complexity

• Tree logics (M2L-Tree / WS2S)

• Implementation issues

• Applications

• Example: *Pointer Assertion Logic*

• Conclusion
Monadic 2nd-order Logic on Strings

\[ \Phi ::= \neg \Phi \mid \Phi \lor \Phi \mid \Phi \land \Phi \mid \Phi \Rightarrow \Phi \mid \Phi \Leftrightarrow \Phi \mid \forall^{1}x. \Phi \mid \exists^{1}x. \Phi \mid \forall^{2}X. \Phi \mid \exists^{2}X. \Phi \]  

(formulas)

\[ t=t \mid t \in T \mid T=T \mid T \subseteq T \mid \ldots \]  

(set terms)

\[ X \mid T \cup T \mid T \cap T \mid T \setminus T \mid \emptyset \]  

(set terms)

\[ x \mid 0 \mid t+1 \]  

(position terms)

- **Weak Monadic 2nd-order logic** = quantification over finite sets
- Two choices for interpretation:
  - **WS1S**: the natural numbers
  - **M2L-Str**: positions in a finite string
- Typical use: as linear temporal logic
Related Logics

- M2L-Tree \sim WS2S
- M2L-Str \sim WS1S \approx S1S
- S2S
- CTL*
- LTL
- CT1

- Tree automata (finite trees)
- String automata (finite strings)
- Tree automata (infinite trees)
- String automata (infinite strings)
Logic → Automata

**Assignment** $A$ of values to $\text{FV}(\Phi)$

⇔

**String** $w_A$ over the alphabet $\Sigma = \{0,1\}^k$ where $k = |\text{FV}(\Phi)|$

Example:
The assignment $A = [P \mapsto \{2,3\}, Q \mapsto \emptyset, R \mapsto 0]$ corresponds to the string:

$$w_A = \begin{pmatrix}
0 & 0 & 1 & 1 & \text{P} \\
0 & 0 & 0 & 0 & \text{Q} \\
1 & 0 & 0 & 0 & \text{R} \\
0 & 1 & 2 & 3
\end{pmatrix}$$

Define the *language* of $\Phi$ : $L(\Phi) = \{ w_A \mid A \models \Phi \}$
Logic → Automata

Simplified syntax:

\[ \Phi ::= \neg \Phi \mid \Phi \land \Phi \mid \exists^2 X. \Phi \]
\[ \mid X_1 \subseteq X_2 \mid X_1 = X_2 \setminus X_3 \mid X_1 = X_2 + 1 \]

Translation of \( \Phi \) into automaton \( A_\Phi \) such that \( L(\Phi) = L(A_\Phi) \):

<table>
<thead>
<tr>
<th>formula ( \Phi )</th>
<th>automaton ( A_\Phi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic formulas</td>
<td>basic automata</td>
</tr>
<tr>
<td>negation ( \neg )</td>
<td>complement ( \mathcal{C} )</td>
</tr>
<tr>
<td>conjunction ( \land )</td>
<td>intersection ( \cap )</td>
</tr>
<tr>
<td>existential quantification ( \exists )</td>
<td>projection+determinization</td>
</tr>
</tbody>
</table>

– we work with deterministic minimal automata
Example 1:

The atomic formula:

$$\Phi = P \subseteq Q$$

corresponds to the basic automaton:

where $P$ corresponds to the first component, and $Q$ to the second...
Example 2:

The composite formula:

\[ \Phi = \exists^2 P. \Psi \]

corresponds to a projection where the P track is removed.

Consider \( \Phi = \exists^2 P. 2 \in P \land 1 \in Q \)

This is for M2L-Str, WS1S also needs a quotient operation after projection.
Automaton Analysis

1. Given a formula $\Phi$, construct the corresponding minimal finite-state automaton $A_{\Phi}$

2. Look at $A_{\Phi}$:
   - If $L(A_{\Phi}) = \Sigma^*$, then $\Phi$ is valid
   - Otherwise, generate a (minimal) counter-example by finding a (minimal) path in $A_{\Phi}$ from the initial state to a non-accepting state
Logic ← Automata

• Every automaton can be encoded as an M2L-Str formula

– but this direction is not relevant for MONA
Complexity

Practical problems:

- The alphabet size is \textit{exponential} in the number of free variables: \( \Sigma = \{0,1\}^k \)

- A single determinization can cause an exponential increase in state-space size

  Worst case: \( 2^2 \cdots 2 \) \#alternating quantifiers

And it is inevitable: The decision problem for WS1S has a \textit{non-elementary} lower bound

Meyer 1972
Only a madman would implement that!

Nils Klarlund
Monadic 2nd-order Logic on Trees

Generalize the structural primitives:
\[ t \ ::= \ x \ | \ 0 \ | \ t+1 \ | \ \varepsilon \ | \ \text{succ}_0(t) \ | \ \text{succ}_1(t) \]

Interpretation: now over tree structures

Again, two choices of models:

WS2S: the infinite binary tree
M2L-Tree: a finite binary tree

Thatcher/Wright 1968
Tree Automata

WS2S / M2L-Tree are also decidable using finite-state automata:

A *bottom-up tree automaton* has a transition function of the form
\[ \delta: Q \times Q \rightarrow \Sigma \rightarrow Q \]
and assigns a state to each tree node starting from the leaves

- All standard automaton operations (product, minimization, subset construction, ...) generalize elegantly to tree automata
- Extra complexity: a *quadratic* blow-up in the transition function

(Later: an example application encoding *tree-shaped data structures* in tree logic...)
Guided Tree Automata (GTA)

– making tree automata practically useful

A GTA *factorizes* the state space:

- A user-defined *guide* assigns a *state space* to each position in the infinite binary tree

- Each state space has its own transition function
  \[ \delta_a : Q_b \times Q_c \rightarrow \Sigma \rightarrow Q_a \]

- This can give an indispensable *exponential* improvement (but it requires a good guide to be defined)
Binary Decision Diagrams (BDDs)

– working with automata with huge alphabets

How to represent transition functions

\[ \delta: Q \rightarrow \Sigma \rightarrow Q \]

when \( \Sigma = \{0,1\}^k \) and \( k \) is large?

The MONA solution:

use **Binary Decision Diagrams** (BDDs)

Worst case: no improvement - Typical case: indispensable!
A **Binary Decision Diagram** is a canonical graph representation for boolean functions \( \{0,1\}^k \rightarrow \{0,1\} \).

Example:

\[[A=0, B=1, C=0, D=1]\] is mapped to 1
We use **Shared Multi-terminal** BDDs:

- *Shared*: each node represents a function
- *Multi-terminal*: the leaves are \{q_1, q_2, ..., q_n\} (not just \{0,1\})

\[ \delta: Q \rightarrow \{0,1\}^k \rightarrow Q \]
Use BDD properties to reuse computations:

Example:

In the formula $\exists X_7: X_1 \subseteq X_7 \land X_7 \subseteq X_9$
the automata for $X_1 \subseteq X_7$ and $X_7 \subseteq X_9$ are isomorphic

DAGification:

- Collapse the formula parse tree to a DAG where the edges are labeled with renaming information
- Build only one automaton for each DAG node
  - gives a factor 2–5 speed-up
Formula Reduction

– optimize the formulas before translating into automata

• Simple reductions:

  \[ \text{true} \lor \Phi \rightarrow \text{true} , \quad \neg \neg \Phi \rightarrow \Phi , \quad \text{etc.} \]

• Quantifier reductions: (can give exponential speed-up!)

  \[ \exists X : \Phi \rightarrow \Phi[T/X] \quad \text{if } \Phi \Rightarrow X = T \text{ and } \text{FV}(T) \subseteq \text{FV}(\Phi) \]

• Conjunction reductions:

  \[ \Phi_1 \land \Phi_2 \rightarrow \Phi_1 \quad \text{if } \Phi_2 \text{ is “contained in” } \Phi_1 \]

– gives a factor 2–4 speed-up
Applications

- Hardware verification [CAV’95, ISMVL’99, FMCAD’00]
- Controller synthesis [FASE’98, FASE’00]
- Trace abstractions [PODC’96]
- Computational linguistics [LACL’97]
- Protocol verification [TACAS’95, FORTE’00]
- Duration calculus
- Parser generation [DLT’99]
- Software engineering [OOPSLA’96]
- Model checking [TACAS’00]
- Theorem proving [CAV’00, FROCOS]
- **Program verification** [PLDI’97, ESOP’00, PLDI’01]
- ...

Logic, Automata, and Program Verification
Pointer Assertion Logic  [PLDI’01]

Consider an imperative **programming language** for data-type implementations, based on **pointers**

Correctness requirements are specified with **assertions** and **pre/post-conditions**

If
- the assertion language (“Pointer Assertion Logic”) is based on **M2L-Tree**,  
- the data-types are restricted to certain **tree-like** structures (“graph types” [POPL’93]), and  
- the program is sufficiently **annotated**
then correctness can be encoded as MONA formulas!
Red-Black Search Trees

Example: A red-black search tree is
1. a binary tree whose records are red or black and have parent pointers
2. a red record cannot have a red successor
3. the root is black
4. the number of black records is the same for all direct paths from the root to a leaf

- 1) is a graph type 😊
- 2) and 3) can be captured as PAL formulas 😊
- 4) cannot be expressed 😞
The redblackinsert procedure

```plaintext
proc redblackinsert(data t, root:Node):Node
{ t.left=null & t.right=null & inv(root) }

pointer y,x:Node;
    x = t;
    root = treeinsert(x,root) [treeinsert.Z=x & treeinsert.Q=root];
    x.color = false;
while [x!=null & root<(left+right)*>x & almostinv1(root,x) & (black(root) | x=root) & (x!=root & red(x.p) => red(x))]
    (x!=root & x.p.color=false) {
        if (x.p=x.p.p.left) {
            y = x.p.p.right;
            if (y!=null & y.color=false) {
                x.p.color = true; y.color = true;
                x.p.p.color = false;
                x = x.p.p;
            } else {
                if (x=x.p.right) {
                    x = x.p;
                    root = leftrotate(x,root) [leftrotate.X=x & root<(left+right)*>x & red(leftrotate.Y)];
                } x.p.color = true;
                x.p.p.color = false;
                root = rightrotate(x.p.p,root) [rightrotate.Y.left=x & root<(left+right)*>x &
                    red(rightrotate.X) & rightrotate.Q=root & x!=null];
                root.color = true;
            } }
        else { ... } }
    root.color = true;
    return root;
} [inv(return)]
```

+ auxiliary procedures leftrotate, rightrotate, and treeinsert (total ~135 lines of program code)
Hoare Logic

1. Require **invariants** at all while-loops and procedure calls (extra assertions are also allowed)

2. Split the program into **Hoare triples**: \( \{ \Phi_{\text{pre}} \} \ \text{stm} \ \{ \Phi_{\text{post}} \} \)

3. Verify each triple separately (only loop-free code left)
   - including check for null-pointer dereferences and other memory errors

Note: highly modular, no fixed-point iteration, but requires invariants!
Verifying the Hoare triples

Use a technique of *transductions* [CAAP’94] to encode loop-free code:

- A collection of M2L-Tree **store predicates** describes a set of stores at a given program point, e.g:
  - `succ_T_d(v,w)` is true if `v` denotes a record of type `T` with a pointer field `d` pointing to the record `w`
  - `ptr_p(v)` is true if `v` denotes the record pointed to by the program variable `p`

- Each statement is simulated by **predicate transformation**, e.g:
  - `p = q.next;` is simulated by updating the `ptr_p(v)` predicate to `ptr_p'(v) = \exists w. ptr_q(w) \land succ_T_next(w,v)`

- A **verification condition** is constructed by expressing the pre- and post-condition using store predicates from end points

  This technique is sound *and complete* for individual Hoare triples!
Logic, Automata, and Program Verification

**PALE**: The Pointer Assertion Logic Engine
- an implementation of this program verification technique

`redblackinsert` ~ 800K formulas

Result of running PALE on `redblackinsert`:
After ~4000 tree automaton operations and 40 seconds,
PALE replies that
- all assertions are valid
- there can be no null-pointer dereferences or memory leaks
- the graph type is wellformed and valid at all cut-points

If verification fails, a **counterexample** initial store is returned
Conclusion

**MONA v1.4:**
- implementation of classical logic/automaton theories
- orders of magnitude more **efficient** than the first implementation due to BDDs, formula reductions, etc.

Future plans:
- heuristic optimizations
- high-level language extensions
- more applications

More information:
- The MONA Project: [http://www.brics.dk/mona/](http://www.brics.dk/mona/)
- Pointer Assertion Logic: [http://www.brics.dk/PALE/](http://www.brics.dk/PALE/)
- (Open Source implementations, full documentation, papers, ...)

Logic, Automata, and Program Verification 34