Mona Tutorial

Automata-based Symbolic Computation

Nils Klarlund

klarlund@research.att.com

AT&T Labs-Research

BRICS, University of Aarhus

Goals

Practical introduction to crunching formulas in an almost undecidable logic: the monadic second-order theory of numbers

A little theory; comparison to BDDs and model checking

Applications: verification, program analysis, parsing

Sense of possibilities and limitations

What we're going to do

- simple Mona examples
- features of the Mona tool
- indicate how does it work
- larger examples
  - pointer verification
  - sliding window protocol verification
  - application to parsing
- projects using Mona

- comparison to model checking
- future work

Mona Essentials

Mona

- is a logic notation
- is a decision procedure (deciding the validity status of formulas)
- calculates for any formula an automaton that describes the possibly infinite set of all satisfying interpretations
- to *decide* a formula means producing this automaton

Note: we just revealed an exciting property: the set of *all* models is finitely describable and the description is computable!!!

BDD Essentials

- BDD = Binary Decision Diagram
- A special kind of automaton that accepts a language of strings of bounded length
- Quantified Boolean Logic
- Crunching BDDs constitutes a decision procedure
- It calculates for any formula an BDD that describes the finite set of all satisfying interpretations

So,...

This talk is about exploring symbolic computations based on finite-state automata, all of them, not only the BDD ones.

Our first program

MONA logic is about natural numbers and sets of natural numbers.

Example: `simple.mona` is

```
var2 P,Q;
P\Q = {0,4} union {1,2};
```

Formula is not always true, but interpretation assigning $\{0, 1, 2, 4\}$ to `P` and $\{\}$ to `Q` makes it true.

At the heart of the matter I

This interpretation $[\mathtt{P} \mapsto \{0, 1, 2, 4\}, \mathtt{Q} \mapsto \{\}]$, can also be represented by 0s and 1s in a *string*

$$\begin{array}{cc} \mathtt{P} \\ \mathtt{Q} \end{array} \quad \begin{array}{cccccc} \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}0\\0\end{pmatrix} & \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}0\\0\end{pmatrix} \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array}$$

where *letters* are bit-vectors.

- P-track is 111010
- Q-track is 000000

At the heart of the matter II

We define the *language* associated to `simple.mona` as the set of such finite strings that define satisfying interpretations.

MONA is able to analyze `simple.mona` automatically by translating it into the minimum automaton recognizing the set of satisfying interpretations. The command

```
  mona simple.mona
```

produces the automaton, analyzes it, and generates the following output:

Output from simple.mona

```
A counter-example of least length (0) is:
P                X
Q                X

P = {}
Q = {}

A satisfying example of least length (5) is:
P                X 11101
Q                X 000X0

P = {0,1,2,4}
Q = {}
```

For the logicians: WS1S

Weak **S**econd-order theory of **1 S**uccessor

- Monadic second-order logic with successor function “+1” and “0”.
- Note: no addition.
- First-terms interpreted as natural numbers.
- Monadic means: only unary predicates may occur as second-order variables.
- Thus second-order terms denote sets of natural numbers.
- “Weak” means these sets are finite.

Syntax summary

- First-order terms t
  - constant 0, variables p
  - successor terms $t' + k$, where t' is a term, k number
- Second-order terms T
  - constant $\emptyset$, variables P
  - $T' \cup T'$, $T' \cap T''$, $T' \backslash T''$
  - no complement operation
- Formulas ϕ
  - $t = t'$, $t \leq t'$, $T = T'$, $t \in T$
  - $\exists p : \phi'$, $\forall p : \phi'$
  - $\exists P : \phi'$, $\forall P : \phi'$

Example explaining it all: `even.mona`

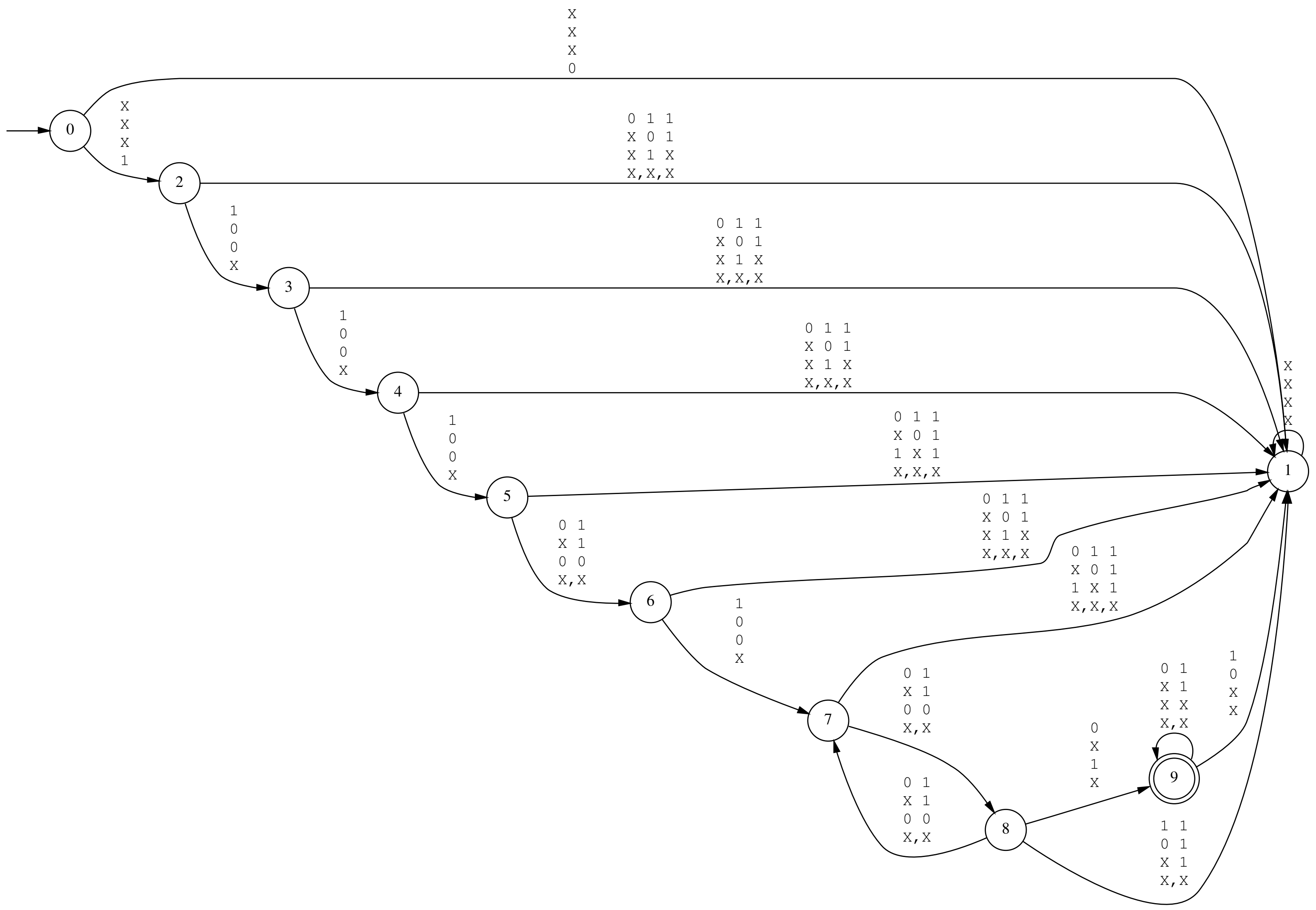
```
var2 P,Q;
P\Q = {0,4} union {1,2};

var1 x;
var0 A;

ex2 Q: x in Q
  & (all1 q:
      (0 < q & q <= x) =>
          (q in Q => q - 1 notin Q)
        & (q notin Q => q - 1 in Q))
  & 0 in Q;

A & x notin P;
```

Output: even.mona

```
A counter-example of least length (1) is:
P                X X
Q                X X
x                X 1
A                0 X

P = {}, Q = {}, x = 0, A = false

A satisfying example of least length (7) is:
P                X 1110100
Q                X 000X0XX
x                X 0000001
A                1 XXXXXXX

P = {0,1,2,4}, Q = {}, x = 6, A = true
```

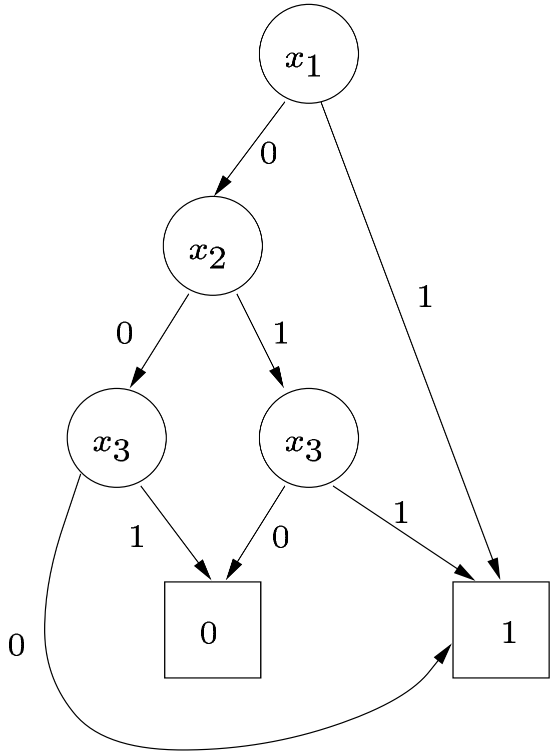
`even.mona`: what is actually going on?

Let us look at the automaton that represents all satisfying interpretations

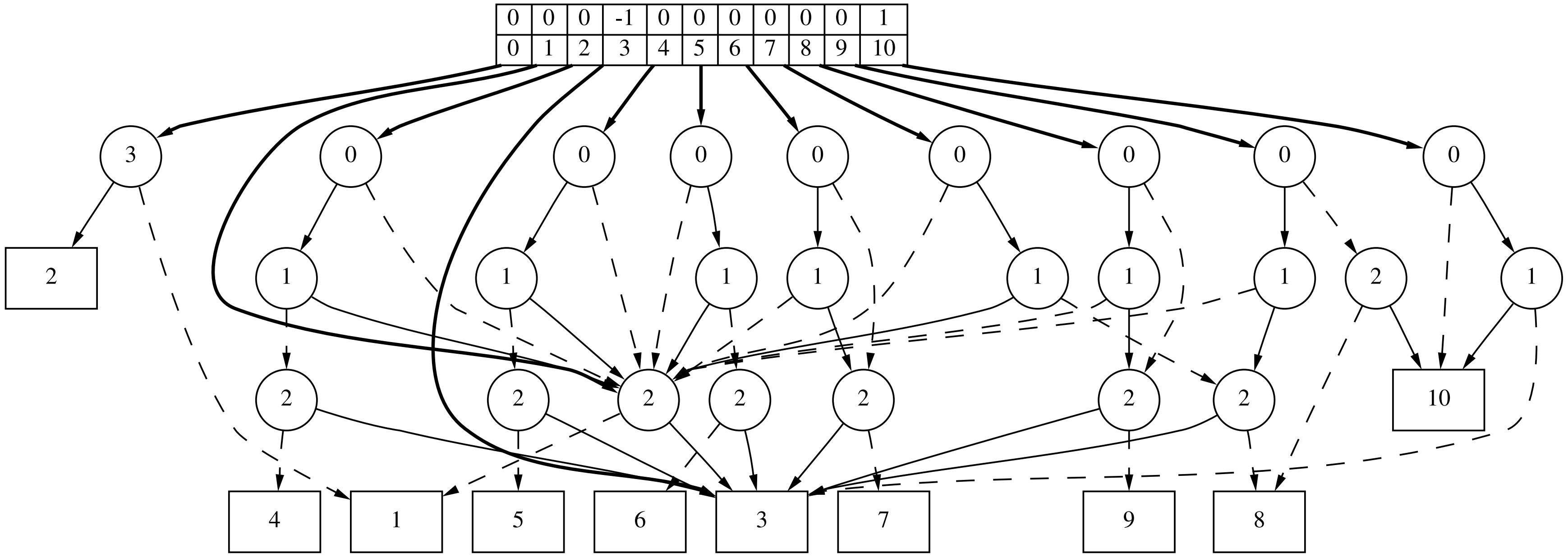

X X X 0
X X X 1
0 1 1 / X 0 1 / X 1 X / X,X,X
1 0 0 X
0 1 1 / X 0 1 / X 1 X / X,X,X
1 0 0 X
0 1 1 / X 0 1 / X 1 X / X,X,X
1 0 0 X
0 1 1 / X 0 1 / 1 X 1 / X,X,X
0 1 / X 1 / 0 0 / X,X
0 1 1 / X 0 1 / X 1 X / X,X,X
1 0 0 X
0 1 1 / X 0 1 / 1 X 1 / X,X,X
0 1 / X 1 / 0 0 / X,X
0 1 / X 1 / 0 0 / X,X
0 X 1 X
0 1 / X 1 / X X / X,X
1 0 X X
1 1 / 0 1 / X 1 / X,X
X X X X
0
1
2
3
4
5
6
7
8
9

Remark: what is a BDD?

Consider $\phi = x_1 \vee (x_2 \Leftrightarrow x_3)$. Variable order is x_1, x_2, x_3.



even.mona: the BDD-based represenation



This automaton is different from the one just shown. It has been formatted from the internal representation.

We forgot predicates

```
var2 P,Q;
P\Q = {0,4} union {1,2};

pred even(var1 p) =
  ex2 Q: p in Q
    & (all1 q:
        (0 < q & q <= p) =>
            (q in Q => q - 1 notin Q)
          & (q notin Q => q - 1 in Q))
    & 0 in Q;

var1 x;
var0 A;

A & even(x) & x notin P;
```

How the decision procedure works

- $w \models \phi$ means the string w, viewed as an interpretation, satisfies ϕ.
- Let $L(\phi) = \{w \mid w \models \phi\}$.
- Prove by induction that all $L(\phi)$ are represented by DFAs.
  - atomic formulas by direct construction of small automaton
  - $\neg$ by complementation
  - $\wedge$ by product construction
  - $\exists$ by projection, a slightly subtle trick, and subset construction

Classical results

$\boxed{L(\phi) \text{ is regular}}$

Büchi, Elgot, 1960

Consider any translation $\phi \longrightarrow A$ with $L(\phi) = L(A)$. Then:

$\left.2^{2^{\cdot^{\cdot^{\cdot^{2^{c \cdot n}}}}}}\right\} c \cdot n$ is a lower bound for A's state space

Meyer and Stockmeyer, 1974

In practice?

- Unfortunate fact: n free variables means alphabet has size 2^n; seems to preclude any practical applications.

- Think of automata as representing a language over $\mathbb{B}$ instead of $\mathbb{B}^k$. That helps, but still large overhead.

- BDDs for compressing states? Easier said than done!

- For example, one single BDD code could encode the set $\{(r, a, s) \mid r \xrightarrow{a} s\}$, where r and s are states and $a \in \mathbb{B}^k$ is a letter. But how to minimize?

- Thus, choose explicit state representation, but encode alphabet.

Decision procedure works for trees, too

- WS2S, **W**eak **S**econd-order theory of **2 S**uccessors, deals with elements and finite subsets of the infinite, binary tree

$$(\mathbf{0}+\mathbf{1})^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$$

- The transition function of a tree automaton determines for each pair (r, s) of states and each letter a what the next state is.

- Tree automata are at least quadratically more difficult to work with in practice.

Related work

So, somebody must have noticed these BDD-represented automata?

Yes, **Fisher and Gupta, 1993** who suggested *linear inductive functions* as a notation for regular languages on large alphabets. They showed that these functions could be represented as automata very similar to ours, but they did not exploit the connection to logic.

Automata on large alphabets are also implicit in the work on the relationship between *p-adic numbers* and circuits by Vuillemin, 1995.

Many experimental systems exist for automata calculations with modest-sized alphabets.

End of digression, back to Mona tool

Sometimes things don't go as fast. Let us try a formula that is 65,000 characters long.

```
mona -s t60.mona
```

The Mona tool

- 36,000 lines of C, C++
- downloaded directly from BRICS +500 times
- heavily optimized
  - BDD algorithms: minimizing cache miss complexity
  - preprocessing: sophisticated analysis and manipulation of code tree
- three-valued logic to avoid spurious state-space explosions
- along with other features...

Feature: visualization

Output to the AT&T Labs graphviz tool.

```
mona -gw even.mona > even.dot
dot -Tps even.dot -o even.ps
```

Feature: separate compilation

MONA can be used so that all automata corresponding to predicate applications are stored in an automaton library, and automatically reused in subsequent executions of MONA.

Feature: restrictions

A *where restriction* `where` ρ can be added to a variable declaration for p. Whenever a subformula contains p as a free variable and an interpretation does not satisfy ρ, the value of the subformula is deemed $\perp$. So, MONA is in fact three-valued!

Example:

```
var1 $ where $ <= 5;
var1 p where p <= $;
p > 5;
```

The status of $\phi_0 = \texttt{p} > 5$ under the interpretation $[\texttt{p} \mapsto i]$ is $\perp$ if $i > 5$ and 0 for $i \leq 5$. In particular, `p > 5` is not satisfiable under the conjunction of the restrictions for p and $.

This relativization technique works thanks to some nice congruence-theoretic arguments about regular languages.

Feature: Exporting and importing automata

- "`.dfa`"-format
- Export:

  `export("`$filename$`", `ϕ`)`

- Import:

  `import("`$filename$`", `n_1`->`n'_1`, `n_2`->`n'_2`, ... , `n_k`->`n'_k`)`

Feature: Presburger arithmetic

- Presburger arithmetic has formulas built of natural number constants and variables, equality, first-order logical connectives and addition.
- Presburger arithmetic can be encoded by letting numbers be expressed as sets in WS1S according to their binary presentation.
- `pconst(`I`)`

  encodes the natural number I as a second-order value using least-significant-bit-first encoding.
- All operations encoded as MONA predicates.

Comparisons between Mona and alternative implementations of decision procedures for Presburger arithmetic show no clear winner.

Feature: Reusing intermediate results

MONA code is stored in a DAG (Directed Acyclic Graph), not a tree. The atomic formulas are located in the leaves and the composite constructs are in the internal nodes. We DAGify according to:

Two formulas are *syntax-equivalent* if their code trees are identical.

Two formulas ϕ and ϕ' are *signature-equivalent* if there is an order-preserving renaming of the variables in ϕ (i.e., increasing w.r.t. the indices of the variables) such that ϕ and ϕ' become syntax-equivalent.

This DAG-representation means huge gain in practice when machine-generated MONA code is translated.

Example: reasoning about queues

Model queues of arbitrary length that contain elements in $\{0, 1, 2, 3\}$. Idea: use three second-order variables:

- `Qe` denotes the used positions, so we assume it is an initial subset of the natural numbers.
- The four possible membership status combinations that a position `p` has relative to `Q1` and `Q2` encode the value stored at position `p`.

Goal: do reasoning about insertion and deletion of elements in lossy queues, which may drop elements.

Queues: basics

```
pred is1(...)
...
pred is2(var1 p, var2 Qe,Q1,Q2) = p in Qe & p in Q1 & p notin Q2;
...
# lt compares the elements at positions p and q of a queue
pred lt(var1 p, q, var2 Qe, Q1, Q2) =
   (is0(p, Qe, Q1, Q2) & ~is0(q, Qe, Q1, Q2))
|  (is1(p, Qe, Q1, Q2) & (is2(q, Qe, Q1, Q2) | is3(q, Qe, Q1, Q2)))
|  (is2(p, Qe, Q1, Q2) & (is3(q, Qe, Q1, Q2)));
...
pred isLast(var1 p, var2 Qe) =
  p in Qe & (all1 q': q' in Qe => q' <= p);
```

etc.!

Queues: playing with definitions

Is there an ordered queue `Q` of length 4 and a queue `Q'` such that `Q'` contains the value 3 and is the same as `Q` with one element removed?

```
var2 Qe, Q1, Q2;      # the queue Q
var2 Qe', Q1', Q2';   # the queue Q'

assert isWfQueue(Qe);

Queue(Qe, Q1, Q2, 4);    # the queue Q is a queue of length 3
                         # containing the elements 0, 1, 2, 3

LooseOne(Qe, Q1, Q2, Qe', Q1', Q2'); # Q' is Q except for one

ex1 p: is3(p, Qe', Q1', Q2'); # Q' does contain the element 3
```

Queues: solution

MONA says:

```
A satisfying example (for assertion & main) of least length (4) is:
Qe                X 1111
Q1                X 0011
Q2                X 0101
Qe'               X 1110
Q1'               X 001X
Q2'               X 011X

Qe = {0,1,2,3}, Q1 = {2,3}, Q2 = {1,3}, Qe' = {0,1,2}, Q1' = {2}, Q2' = {1,2}
```

So, $Q = \langle 0, 1, 2, 3\rangle$ and $Q' = \langle 0, 1, 3\rangle$ is a solution.

What is clear: WS1S is not a convenient specification language by itself. We need a higher-level notation.

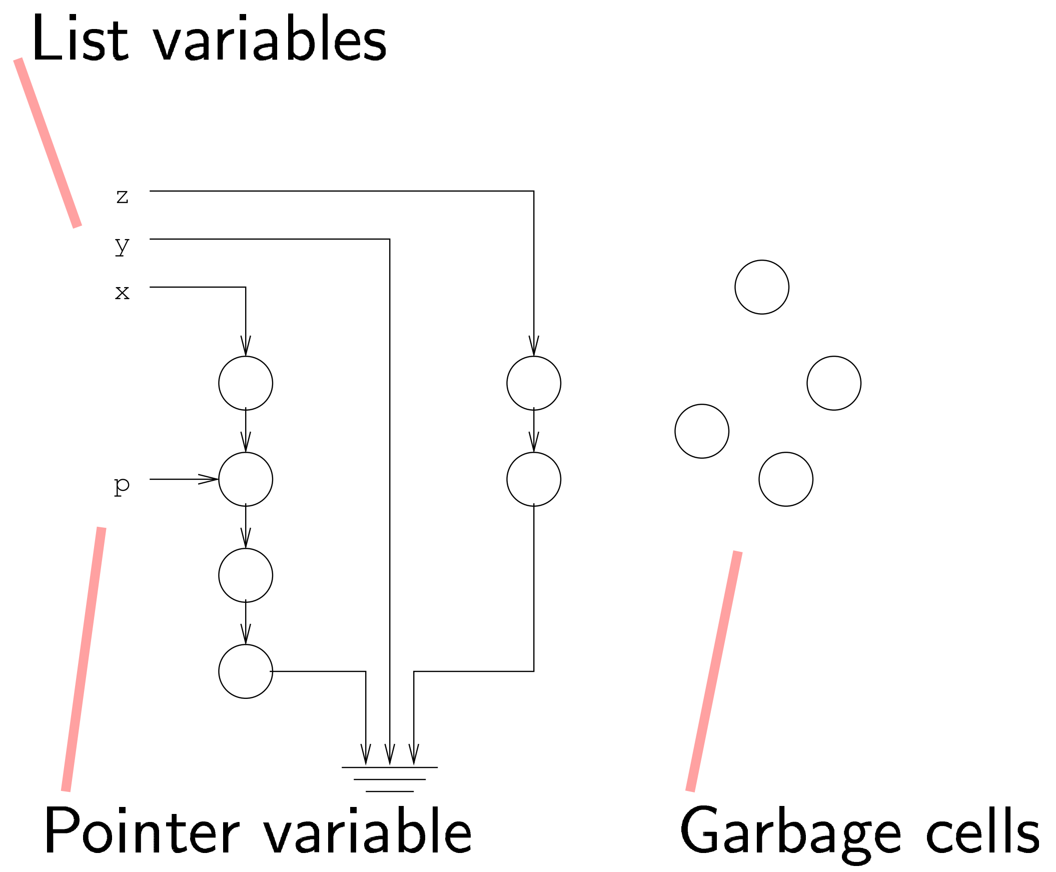
Decidable program logic for pointers

FIDO

- Can model list and pointer variables (as long as list elements are bounded).
- Can represent precise semantics of basic blocks (loop-free code).
- Can be used to express assertions in a first-order language.

Thus, we have a decidable fragment of Floyd/Hoare program logic.

Stores



No general graphs, of course!

An example in Pascal

Let us declare a type `Item` of list elements that contains a `tag` field with value `red` or `blue`:

```
type Color = (red,blue);

List = ^Item;

Item = record
case tag: Color of
   red,blue: (next: List)
end;
```

Verifying a list zipping program

```
program zip;
{data}var x,y,z: List; {pointer}var p,t: List;
begin
    if x=nil then
    begin
       t:=x; x:=y; y:=t
    end;
    z:=nil; p:=nil;
    while x<>nil do
    begin
       if z=nil then
       begin
          z:=x; p:=x;
       end
       else
       begin
          p^.next:=x;
          p:=p^.next
       end;
       x:=x^.next;
```

```
      p^.next:=nil;
      if y<>nil then
      begin
         t:=x; x:=y; y:=t
      end
   end
end.
```

We need an invariant

Invariant: `x` is only empty if `y` is empty, and `p` points to the last element of `z`.

So, we annotate `while`-loop with assertion

```
{ (x=nil => y=nil)
 & z<next*>p & (z<>nil => p^.next=nil)}
```

Then, the program is verified by the push of a button.

Predicate transformer semantics I

- a predicate $\mathit{points\text{-}to}(\alpha, \beta)$ holds iff the cell at α contains a pointer that points to β
- this predicate is definable for a well-formed store
- $\{P\}S\{Q\}$
- S is a piece of straight-line code, e.g. `p^.next:=x;`
- the store after is the same as before except that the predicate $\mathit{points\text{-}to}(\alpha, \beta)$ has changed for $\alpha = \mathtt{p}$
- let Q' be Q, where we use modified $\mathit{points\text{-}to}(\alpha, \beta)$

Predicate transformer semantics II

- WF describes a well-formed store
- this property can be formulated as a fixed-point formula based on the $points\text{-}to(\alpha, \beta)$ predicate
- the sets of cells that can be reached from the root of the data structures
  - are disjoint, and
  - their union is everything that is allocated.

Predicate transformer semantics III

- a fixed-point formula can be encoded using second-order quantifier.
- WF is built into the assumptions of $\mathit{points\text{-}to}(\alpha, \beta)$
- so all we need to do is to verify:

$$P \;\rightarrow\; Q' \wedge WF'$$

- which is interpreted over the well-formed initial stores

Fido formulation (15k)

```
type Label = Null,Lim,Item_red,Item_blue;
type Elm = Label(next: Elm) | empty;

func Type_Item(tree s_0:Elm; pos mu_a:Elm): formula;
mu_a=Item_red or mu_a=Item_blue
end;

...

func WfStore'(tree s_0:Elm): formula;
exists set x_Cells: s_0.(
List_x_After(s_0,x_Cells);
exists set y_Cells: s_0.(
List_y_After(s_0,y_Cells);
RecordCells_After(s_0,x_Cells + y_Cells);
NullSet(s_0,x_Cells * y_Cells)))
end;

func Check_Validity(tree s_0:Elm): formula;
...
```

Mona formulation (50k)

```
var1 $;
defaultwhere1(p) = p<=$;
# Type environment

var2 G0;
var2 S0,S1;
...
pred FUNC_DeAllocCheck_IF1_L_5(var2 G0_s_0,var2 S0_s_0,S1_s_0) =
((ex2 G0_445,S0_445,S1_445: (((G0_445=G0_s_0) & (S0_445=S0_s_0) & (S1_445=S1_s_0)))
& FUNC_DeAllocCheck_IF1_L_4(G0_445,S0_445,S1_445)) & (ex1  POS_mu_0: (TYPE_Elm(POS_mu
& ((ex1  POS448: ex2 G0_449,S0_449,S1_449: (((G0_449=G0_s_0) & (S0_449=S0_s_0) &
(S1_449=S1_s_0))) & (POS448=POS_mu_0) & FUNC_Root_y_IF1_L_1(G0_449,S0_449,S1_449,POS4
& (~(ex1  POS452: ex2 G0_453,S0_453,S1_453: (((G0_453=G0_s_0) & (S0_453=S0_s_0) &
(S1_453=S1_s_0))) & (POS452=POS_mu_0) & FUNC_DeAlloc_0(G0_453,S0_453,S1_453,POS452)))

pred FUNC_Succ_next_IF1_L_1(var2 G0_s_0,var2 S0_s_0,S1_s_0,var1 POS_mu_a,var1 POS_mu_
```

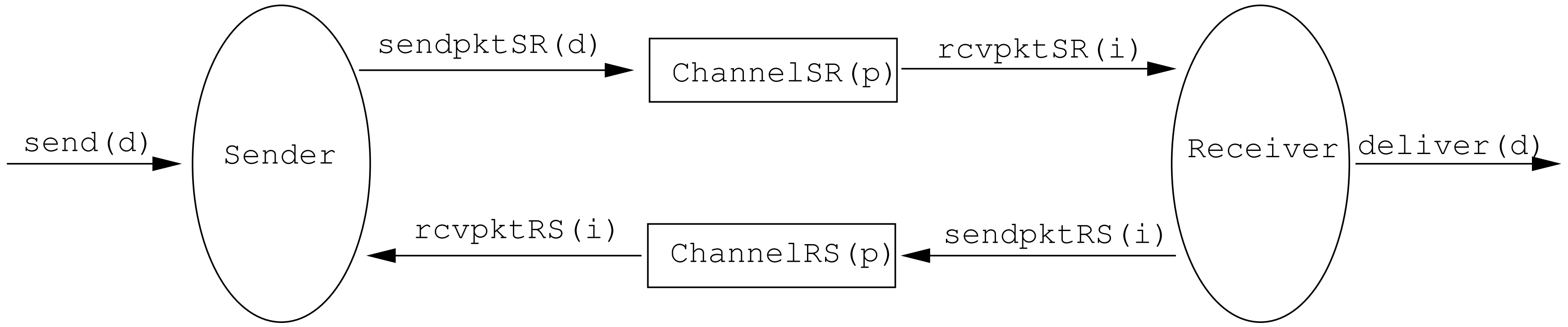
Sliding window protocol

- The classic protocol studied in verification is the *alternating bit protocol*.
- A bit is used to acknowledge the receipt of a message.
- In the *sliding window protocols*, a sequence number is used as an acknowledgment.
- The *window size* is the number of messages that can be in transit.

Sliding window protocol modeling

- We'll use IOA (Input/Output Automata) notation
- Reasonable level of abstraction:
  - not a program
  - but close
- We'll model
  - unbounded queues,
  - unbounded channels, and
  - dynamic window size.

Sliding window protocol in a picture



Proving safety property "What goes out is what goes in"

- IOA description of general protocol
- two uses of abstract interpretation techniques
- IOA description of abstract protocol
- whose transition relation $\rightarrow$ is WS1S (at least intuitively)
- our IOA to MONA translator proves that part
- extend safety property to a presumed invariant ϕ
- verify by "push of a button" $\phi(S) \ \wedge S \rightarrow S' \ \wedge \phi(S')$.

Applications of tree logic: parsing with logical side constraints

Syntax = context-free grammar + side constraints
The full grammar may be:

- not context-free;
- outside tractable grammar classes;
- too large; or
- unintutive and hard to maintain.

A simple HTML0 grammar

```
H : H E
  | E
;
E : <a href=url> H </a>
  | <b> H </b>
  | <i> H </i>
  | <ul> L </ul>
  | word
;
L : /* empty */
  | L <li> H
;
```

No nested anchor tags

```
H : H E
  | E
;
E : <a href=url> H' </a>
  | <b> H </b>
  | <i> H </i>
  | <ul> L </ul>
  | word
;
L : /* empty */
  | L <li> H
;
```

```
H' : H' E'
   | E'
;
E' : <b> H' </b>
   | <i> H' </i>
   | <ul> L' </ul>
   | word
;

L' : /* empty */
   | L' <li> H'
;
```

Constraints yield large grammars

- No nested anchor tags: 6 nonterminals.
- List are nested to depth at most three: 12 nonterminals.
- If any part of an anchor text is in boldface, then all anchor texts must be so in full: 16 nonterminals.

If all constraints are imposed simultaneously, we get more than 100 nonterminals.

Checking constraints

- context-free grammars: *difficult, not modular*;
- tree-walking procedures: *programming, parse trees*;
- attribute grammars: *explicit flow*; but
- logical side constraints: *expressive, modular, efficient!*

Parse tree logic

Terms:

t :	$\$\$$	the root
	$\mid t.i$	the i'th child node of t
	$\mid \alpha$	a first-order variable

Term types:

τ :	N	any production of nonterminal N
	$\mid N[j]$	the j'th production of nonterminal N

Parse tree logic

Formulas:

$\phi : t_1 < t_2$	ancestor relation
$\mid t_1 = t_2$	equality
$\mid \neg\phi$	negation
$\mid \phi_1 \Rightarrow \phi_2$	implication
$\mid \phi_1 \wedge \phi_2$	conjunction
$\mid \phi_1 \vee \phi_2$	disjunction
$\mid \exists\alpha : \tau . \phi$	existential quantification
$\mid \forall\alpha : \tau . \phi$	universal quantification

Example formulas

- No nested anchor tags:

$$\forall a : E[1].\ \neg\exists b : E[1].a < b$$

- List are nested to depth at most three:

$$\neg\exists\, a, b, c, d : E[4].a < b \wedge b < c \wedge c < d$$

- If any part of an anchor text is in boldface, then all anchor texts must be so in full:

$$\exists a : E[1].\exists b : E[2].\exists w : E[5].a < w \wedge b < w$$

$$\Downarrow$$

$$\forall w : E[5].(\exists a : E[1].a < w) \Rightarrow (\exists b : E[2].b < w)$$

The rest: encoding

Idea: encode parse trees as binary trees. Interpret logic over these trees. Calculate automata that represent synthesized attributes. Generate Bison (Yacc) code for these.

It works and parsers become very fast.

Some projects using Mona

- The PVS Specification and Verification System
- PALE - The Pointer Assertion Logic Engine.

See Web-site for more

Comparison with model checking

Is one stronger than the other?

- Model checking: given a (compact) representation of a finite-state system and a formula ϕ, decide whether all *infinite* computations satisfy ϕ.
- Logic-automaton connection: determine validity of a formula ϕ on all *finite* computations.

Some problems, such as determining whether a safety property holds of a finite-state system, can be solved using either approach; but in general, the approaches are incomparable.

Future work

Lots of interesting theoretical and practical problems left to study:

- finish a final high-level language (based on FMona and Fido)
- automata-based analysis of XML grammars
- much reasoning about programs boils down to nitty-gritty arguments that can be carried out through automated decision procedures; how to apply abstract interpretation without being entangled by theorem proving
- reformulate advanced regular expressions (Perl) in a more readily understandable declarative manner—and translate them into finite-state automata guaranteed to run fast
- integrate such techniques with parser generators
- study the problem of making automata-algorithms such that the logic-automaton connection is presented here subsumes BDD to within a constant factor
- generalize the algorithm presented here to the ω-theoretic framework—thus extending the techniques of Vardi & Wolper for converting temporal logic to automata

Joint work with

Michael I. Schwartzbach (programming language applications, FIDO)

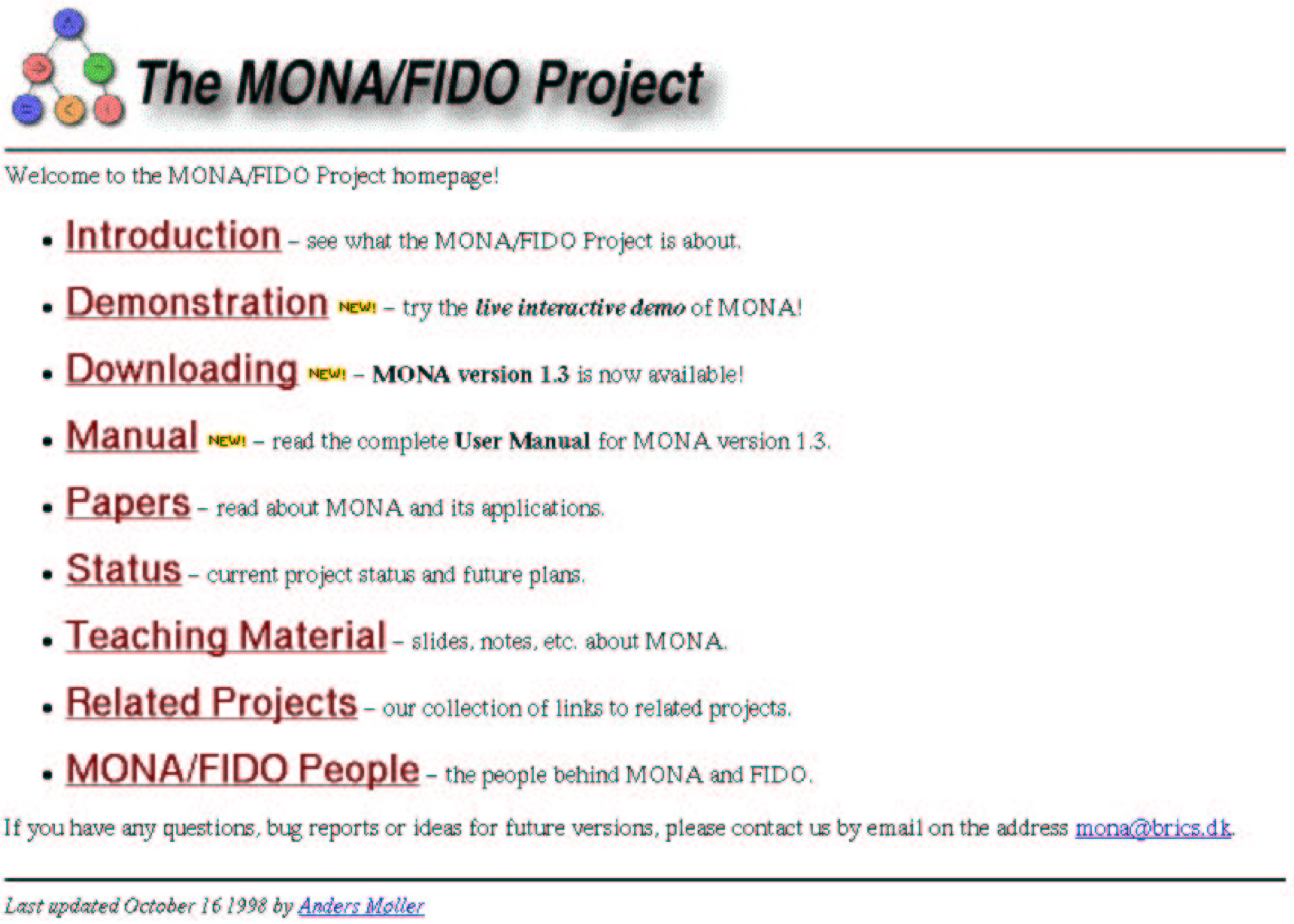
Anders Møller (Mona development and implementation)

Niels Damgaard

Mark Smith (protocol verification)

Jackob Elgaard, Jacob Jensen, Michael Jørgensen (pointer verification)

The MONA homepage `http://www.brics.dk/mona/`



The MONA/FIDO Project

Welcome to the MONA/FIDO Project homepage!

- Introduction – see what the MONA/FIDO Project is about.
- Demonstration NEW! – try the ***live interactive demo*** of MONA!
- Downloading NEW! – **MONA version 1.3** is now available!
- Manual NEW! – read the complete **User Manual** for MONA version 1.3.
- Papers – read about MONA and its applications.
- Status – current project status and future plans.
- Teaching Material – slides, notes, etc. about MONA.
- Related Projects – our collection of links to related projects.
- MONA/FIDO People – the people behind MONA and FIDO.

If you have any questions, bug reports or ideas for future versions, please contact us by email on the address mona@brics.dk.

Last updated October 16 1998 by Anders Møller