MONA - MSO in practice

Martin Emrich
Tutors: Tim Priesnitz, Thomas Klöcker, Christian Klein

PS Lab
Gert Smolka
Saarland University

October 20, 2003

Abstract

This presentation is about MONA, an MSO theorem prover using BDDs (binary decision diagrams) and finite-state automata. We will show what it can do and shed some light on its internal optimization techniques.

1 Introduction

MONA is a theorem prover for WS1S/WS2S formulas written in C/C++. MONA takes a program consisting of formulas and variable definitions and generates a finite state tuple automaton that recognizes the language of the conjunction of all formulas in the program.

MONA is/has been used in numerous practical applications, including

- Hardware verification
- Controller synthesis
- Parser generation (YakYak, an extension to `yacc`)
- Protocol verification
- Program verification

[KM01]

```
var1 x;
var2 X,Y;
x in X => x+1 in Y;
```

Figure 1: A simple MONA Program

2 Programming MONA

MONA programs primarily consist of variable definitions

`var0 b;` | `var1 p;` | `var2 P;`

(where $\mathtt{b} \in \mathbb{B}, \mathtt{p} \in \mathbb{N}$ and $\mathtt{P} \subset \mathbb{N}$) and formulas:

t	::=	0	$\mid p$	$\mid t+1$	
T	::=	$\emptyset$	$\mid P$	$\mid T$ `sub` T	
	$\mid$	T `union` T	$\mid T \setminus T$		
ϕ	::=	t `<=` t	$\mid t$ `=` t	$\mid T$ `=` T	
	$\mid$	t `in` T	$\mid$ `b`	$\mid$`~`ϕ	
	$\mid$	ϕ `&` ϕ	$\mid \phi$ `|` ϕ	$\mid \phi$ `=>` ϕ	$\mid \phi$ `<=>` ϕ
	$\mid$	`ex1 p :` ϕ	$\mid$ `all1 p :` ϕ	$\mid$ `ex2 P :` ϕ	$\mid$ `all2 P :` ϕ

More syntax elements such as macro definitions, predicates (which should rather be seen as macros with precompiled automata), etc. can be found in [KM01]. But these constructs deserve a special explanation:

- No formula is allowed to construct infinite sets, something like

  ```
  var 2 G;
  0 in G & all1 p: p in G <=> p+1 notin G;
  ```

  is not allowed.

- If `t` a first-order variable and c an integer constant, $\mathtt{t} - c := 0$ if $c > \mathtt{t}$
- `min(T)` and `max(T)` are 0 when $\mathtt{T} = \emptyset$.

Figure 1 shows a small MONA program for $x \in X \Rightarrow x1 \in Y$.

3 Automaton Language

The automata generated by MONA generally read a language $L \subset (\{\mathtt{0}, \mathtt{1}, \mathtt{X}\}^k)^*$, where k is the number of free variables in the corresponding MONA program,

and `X` being a dummy character denoting a situation where both `0` and `1` have the same effect.

The first tuple of a word contains the truth value for boolean (0th order) variables, or is filled up with an `X` for first- and second-order variables. This enables the automaton to take boolean variables into account as early as possible.

The remaining string contains the first/second-order variables in the usual sense, except that a first order value p is not encoded as a singleton set, but as a non-empty set P with $p = \min(P)$. This way, the automaton can completely ignore all characters in p's fingerprint after reading the first `1`.

An Example:

```
var0 b;     b = true     1XXXXXXXXX
var1 p;       p = 4      X00001XXXX
var2 P;  P = {2,4,5,8}   X001011001
```

Note that there is neither a $\perp$-Symbol nor a shape variable denoting the end of the word. Automata generated by MONA are always in an accepting state if the input string satisfies the corresponding program, and vice versa. If one wants to use a MONA-generated automaton in own projects with finite input strings, one has to take care of the string's end by oneself.

4 Language minimization & Automaton generation

Before starting to build the automaton, MONA simplifies the program:

- Trivial boolean expressions are reduced:

  $T = T \rightsquigarrow$ `true` $\qquad \phi\&\phi \rightsquigarrow \phi$

  `true` $\&\phi \rightsquigarrow \phi \qquad \tilde{}\tilde{}\phi \rightsquigarrow \phi$

  `true` $\&\phi \rightsquigarrow$ `false` $\qquad \tilde{}false \rightsquigarrow$ `true`

  Of course these constructs do not appear very often, but the reduction yields small improvements especially with machine-generated MONA code.

- Second-order terms are "flattened" by introducing new variables for sub-terms. Ex.:

  $A = (B$ `union` $C)$ `inter` $D \quad \rightsquigarrow$
  $\exists V : (A = V$ `inter` $D)\&(V = B$ `union` $C)$

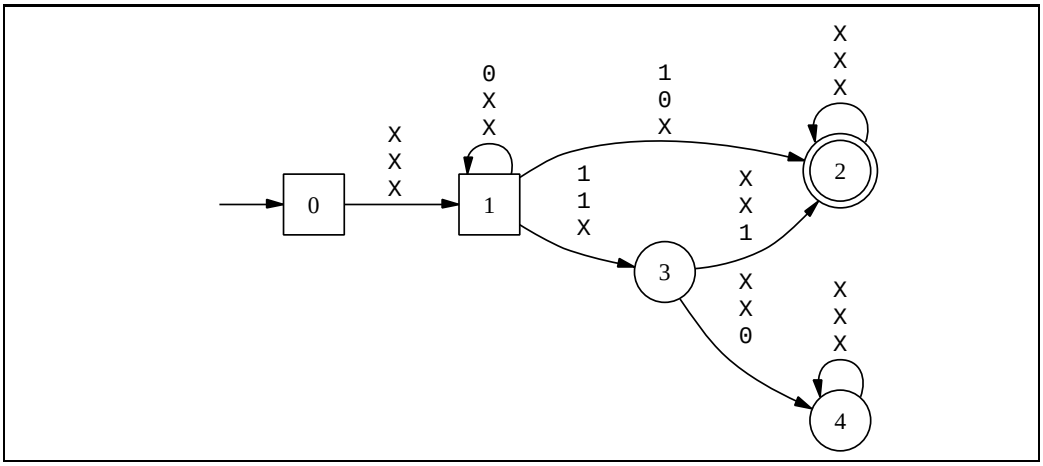


Figure 2: The automaton for the example

- Redundant operations are removed, e.g.:

$$\forall \rightsquigarrow \neg\exists\neg \qquad\qquad \phi \vee \phi' \rightsquigarrow \neg(\neg\phi \wedge \neg\phi')$$

The result is the minimal syntax

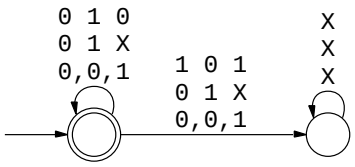
$$\begin{aligned}\phi ::= \ & \neg\phi \quad | \ \phi \wedge \phi' \quad | \ \exists X : \phi \\ & | \ X \subseteq X' | \ X = X' + 1 | \ X = X' \backslash X''\end{aligned}$$

Now, MONA calculates a minimum DFA for a program which accepts all strings with interpretations that satisfy the conjunction of all formulas in the program. The automaton for our example (Figure 1) is shown in figure 2.

Atomic formulas are translated into basic automata, e.g.

$$\phi = (P_1 = P_2 \setminus P_2)$$

is translated to the automaton



Composite formulas are translated by combining the basic automata (e.g. $\phi = \tilde{\phi'}$ corresponds to automaton complementation). After every operation, the automaton is minimized to keep the memory usage low all the time. Of

course, existential quantification results in a non-elementary complex decision procedure (n nested alternating quantifiers):

$$2^{2^{\cdot^{\cdot^{2^{c \cdot n}}}}} \Big\} c \cdot n$$

[Kla98]

The cause for this are the projection operations used to translate quantifications, which produce an additional 2^n state increase caused by the determinization. To avoid this and the following costly normalization, MONA tries to rewrite quantifications:

$$\texttt{ex2}\ X_i : \phi \quad \rightsquigarrow \quad \phi[X_j/X_i]$$

when ϕ is of the form $(\cdots\ \texttt{\&}\ X_i = X_j\ \texttt{\&}\ \cdots)$ and $i \neq j$.

5 Three-valued semantics

By viewing a first-order term t as a singleton second-order term T, we face the problem that this semantics is not closed under complementation. Consider the formula $\phi \ =; p = 2$ (p is first-order), which is handled as $\phi' \ = \ P = \{2\}$ where P is second order. So, the complement of ϕ' is $\tilde{}(P = \{0\})$, something different from the representation of $\tilde{}(p = 0)$, namely $\tilde{}(P = \{0\})$ `&` $\texttt{singleton}(P)$, where the restriction $\texttt{singleton}(P)$ holds when P has exactly one element.[KM01]

To be on the safe side, we would have to carry the restriction with us all the time (*conjunctive semantics*), resulting in numerous additional automaton product and minimization procedures.

Another type of restrictions often needed is limiting variables for M2L-Str, where all first and second-order terms have to be element/subset of a given finite string `$`. To enforce this, we would have to cojoin the restriction $p \in \$$ with every atomic formula, which may easily result in doubly-exponential blow-ups[Kla99].

WS1S-R First, we make the restrictions explicit in syntax. A variable P is associated with a *restriction* ρ:

$$\phi ::= \texttt{ex2}\ P\ \texttt{where}\ \rho : \phi'$$

For the time being, this is equivalent to $\texttt{ex2}\ P : \phi'\ \&\ \rho$, but ρ is cojoined to any sub-formula mentioning P. We assume that every variable P_i is restricted, even if the restriction is "empty", by just being $(P_i = P_i)$.

The ternary semantics Until now, a formula ϕ is either true ($+$) or false ($-$). The MONA semantics adds a third possibility[KM01][Kla99]: The formula ϕ is *don't care* ($\perp$) if the restriction(s) of a free variable in ϕ is not fulfilled.

Let X is an expression. We define

$$\boldsymbol{\rho}^*(X) = \underset{P_i \in \mathcal{P}}{\&} \rho(P_i)$$

where $\mathcal{P}$ contains all free variables in X and all free variables appearing in restrictions on free variables in X. $\boldsymbol{\rho}^*(\phi)$ is the *induced restriction* on ϕ, the conjunction of all restrictions appearing in ϕ.

Conjunction and negation are redefined in a way that if $\|\phi\|w = \perp$ or $\|\phi'\|w = \perp$ (ϕ on the word w evaluates to $\perp$), so do $\|\phi \,\&\, \phi'\|w = \perp$ and $\|\tilde{}\phi\|w = \perp$.

Now, the three-valued semantics is defined as

$$\begin{aligned}
\|\tilde{}\phi\|w &= \tilde{}\|\phi\|w \\
\|\phi \;\&\; \phi'\|w &= \|\phi\|w \;\&\; \|\phi'\|w \\
\|\texttt{ex2}\ P_i\ \texttt{where}\ \rho : \phi'\|w &= \begin{cases} + & \text{if } \exists M : \|\phi'\|w\,[P_i \mapsto M] = + \\ - & \text{if } \forall M : \|\phi'\|w\,[P_i \mapsto M] \neq + \\ & \quad \text{and } \exists M : \|\phi'\|w\,[P_i \mapsto M] = - \\ \perp & \text{if } \forall M : \|\phi'\|w\,[P_i \mapsto M] = \perp \end{cases} \\
\|P_i\ \texttt{sub}\ P_j\|w &= \begin{cases} + & \text{if } w \models P_i\ \texttt{sub}\ P_j \text{ and } \|\boldsymbol{\rho}^*(P_i)\ \&\ \boldsymbol{\rho}^*(P_j)\|w = + \\ - & \text{if } w \not\models P_i\ \texttt{sub}\ P_j \text{ and } \|\boldsymbol{\rho}^*(P_i)\ \&\ \boldsymbol{\rho}^*(P_j)\|w = + \\ \perp & \text{if } \|\boldsymbol{\rho}^*(P_i)\ \&\ \boldsymbol{\rho}^*(P_j)\|w \neq + \end{cases}
\end{aligned}$$

[KM01]

Consequently, the automata now have a three kinds of states, *accepting*, *rejecting* and *don't-care*-states (in fig. 2, the square states), that are reached if a restriction is violated. Consider the string w

$$w = \begin{array}{ll} \texttt{x = 3} & \texttt{X0001X} \\ \texttt{X = \{3,4\}} & \texttt{X00011} \\ \texttt{Y = \{4\}} & \texttt{X00001} \end{array}$$

which satisfies the example in Figure 1. As long as there is no `1` read in x's fingerprint, it clearly violates the `singleton`() restriction, and the automaton stays in the *don't-care* state 1.

Equivalence of binary and three-valued semantics [KM01]

The "classic" semantics and the ternary semantics defined above are equivalent in the following sense:

$$\begin{aligned} w \not\models \boldsymbol{\rho}^*(\phi) &\Leftrightarrow \|\phi\|w = \bot \\ w \models \phi \ \texttt{\&} \ \boldsymbol{\rho}^*(\phi) &\Leftrightarrow \|\phi\|w = 1 \\ w \models \texttt{~}\phi \ \texttt{\&} \ \boldsymbol{\rho}^*(\phi) &\Leftrightarrow \|\phi\|w = 0 \end{aligned}$$

In fact, MONA can be forced to generate two-valued automata, the *don't-care* states are replaced by *rejecting* states.

6 Finite state automata as BDDs

The transition table for even a small automaton (e.g. 5 free variables, 20 states) is usually very big (640 entries). To cut down the memory footprint of automata, MONA stores the automata as *multi-terminal, shared BDDs* (SMBDD). [KMS00]

A generic BDD is a directed, acyclic graph consisting of $k \geq 2$ nodes; the leaves $\{0, 1\}$ and variable nodes. Furthermore, there is a transition function γ with

$$\exists n \in \mathbb{N} : \gamma \in \{0, \ldots, n\} \rightarrow Var \times \{0, \ldots, n\} \times \{0, \ldots, n\}$$

giving for an index n a node v, its low successor (reached when v's variable is 0) and its high successor. [Smo03] If the BDD is *ordered*:

$$\forall (n, (X, n_0, n_1) \in \gamma : n > n_0 \wedge n > n_1$$

we need at most n transitions to reach a leaf. BDDs are more space saving than decision trees, the tree for the example in Figure 3 would have 15 nodes instead of 5.

While a generic BDD has only 2 final nodes (0 and 1), a MONA BDD for an automaton has q leaves, one for every automaton state, each one additionally marked with the node type (*accepting*=1, *rejecting*=-1 or *don't-care*=0). The variable nodes are labeled according to the free variables in the input string, thus in the example, the variable nodes 0 corresponds to x, 1 to X, and 2 to Y. So, the SMBDD can be seen as many BDDs joined together, sharing as many variable nodes as possible. It is no longer acyclic, as there is one cycle per input character.

For every character the automaton reads a k-tuple, which is processed top-down through the SMBDD. Starting at the top in the current state, walking down along the way the k-tuple lays out, and one reaches the next state. This takes at most k steps, instead of looking up in 2^k decision table entries.

A BDD for $(a \wedge \neg b \wedge c)$:

$V = \{a, b, c\}$

$\gamma = \{(4, (a, 0, 3)), (3, (b, 2, 0)), (2, (c, 0, 1))\}$

Ex.: Read γ as "In node 4, read a, when $a = 0$ then enter node 0, otherwise, enter node 3".

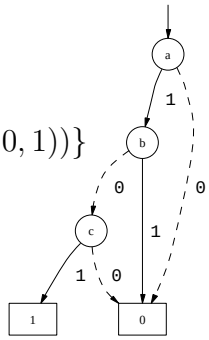


Figure 3: A generic BDD

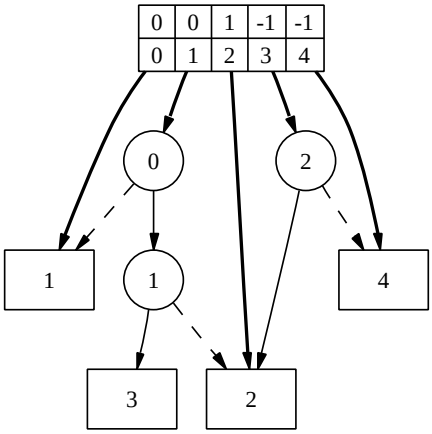


Figure 4: The BBD for the example in Figure 1

7 Model checking & satisfiability

When MONA is run on a program, it also checks if the formula is valid, satisfiable or unsatisfiable.

This is done by searching the shortest path to an accepting/rejecting state. If a path to an accepting state is found, the formula is satisfiable. If no path to a rejecting state is found, too, the formula is valid. If no path to an accepting state is found, the formula is unsatisfiable.

MONA prints a shortest example and counter-example, too, which is generated by concatenating all transitions on the shortest path to the accepting state and rejecting state, respectively. For the example in Figure 1, the examples are:

```
ANALYSIS
```

```
A counter-example of least length (1) is:
x               X 1
X               X 1
Y               X X

x = 0
X = {0}
Y = {}

A satisfying example of least length (1) is:
x               X 1
X               X 0
Y               X X

x = 0
X = {}
Y = {}
```

To get "larger" examples, one can force it by adding a new free first-order variable v and including `v =` *length* in the program. But keep in mind that the automata become very large.

MONA can be used to check if a formula holds for a given input by assigning all free variables a fixed value. For the example, this could be:

```
x = 3; X = {2,3}; Y = {4};
```

If MONA's shortest example is again this interpretation, it satisfies the formula.

References

[Kla98] Nils Klarlund. Mona & fido: The logic-automaton connection in practice. In *Computer Science Logic, CLS'97*, volume 1414. LNCS, 1998.

[Kla99] Nils Klarlund. A theory of restrictions for logics and automata. In *Computer Aided Verification, CAV'99*, volume 1633. LNCS, 1999.

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade 540, DK-8000 Aarhus C, Denmark, 1997-2001.

[KMS00] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. Mona implementation secrets. In *Fifth International Conference on Implementation and Application of Automata, CIAA'00*, 2000.

[Smo03] Gert Smolka. *Vorlesungsskript zu Logik, Semantik und Verifikation.* PS Lab, Saarland University, Saarbrücken, Germany, 2003.