International Journal of Foundations of Computer Science
© World Scientific Publishing Company

MONA Implementation Secrets

Nils Klarlund
AT&T Labs Research
180 Park Ave.
Florham Park, NJ 07932, USA
klarlund@research.att.com

Anders Møller & Michael I. Schwartzbach
BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark
{amoeller,mis}@brics.dk

Received
Revised
Communicated by

ABSTRACT

The MONA tool provides an implementation of automaton-based decision procedures for the logics WS1S and WS2S. It has been used for numerous applications, and it is remarkably efficient in practice, even though it faces a theoretically non-elementary worst-case complexity. The implementation has matured over a period of six years. Compared to the first naive version, the present tool is faster by several orders of magnitude. This speedup is obtained from many different contributions working on all levels of the compilation and execution of formulas. We present an overview of MONA and a selection of implementation "secrets" that have been discovered and tested over the years, including formula reductions, DAGification, guided tree automata, three-valued logic, eager minimization, BDD-based automata representations, and cache-conscious data structures. We describe these techniques and quantify their respective effects by experimenting with separate versions of the MONA tool that in turn omit each of them.

Keywords: monadic second-order logic, finite-state automata, the MONA tool

1. Introduction

MONA [20, 29, 37, 25] is an implementation of decision procedures for the logics WS1S and WS2S (Weak monadic Second-order theory of 1 or 2 Successors) [48]. They have long been known to be decidable [11, 17], but with a non-elementary lower bound [36]. For many years it was assumed that this discouraging complexity precluded any useful implementations.

MONA has been developed at BRICS since 1994, when our initial attempt at automatic pointer analysis through automata calculations took four hours to complete. Today MONA has matured into an efficient and popular tool on which the same analysis is performed in a couple of seconds. Through those years, many different approaches have been tried out, and a good number of implementation "secrets" have been discovered. This paper describes the most important tricks we have learned, and it tries to quantify their relative merits on a number of benchmark formulas.

Of course, the resulting tool still has a non-elementary worst-case complexity. Perhaps surprisingly, this complexity also contributes to successful applications, since it is provably linked to the succinctness of the logics. If we want to describe a particular regular set, then a WS1S formula may be non-elementarily more succinct that a regular expression or a transition table.

The niche for MONA applications contains those structures that are too large and complicated to describe by other means, yet not so large as to require infeasible computations. Happily, many interesting projects fit into this niche, including hardware verification [4, 1], pointer analysis [22, 16, 38], controller synthesis [44, 21], natural languages [39], parsing tools [13], software design descriptions [28], Presburger arithmetic [45], and verification of concurrent systems [31, 30, 23, 42, 46].

There are a number of tools resembling MONA. Independent of the MONA project, the first implementation of automata represented with BDDs was that of Gupta and Fischer from 1993 [19]. However, they used "linear inductive functions" instead of the automaton–logic connection. MOSEL (see `http://sunshine.cs.uni-dortmund.de/projects/mosel/`) implements the automaton based decision procedure for M2L-Str using BDDs like MONA. In [24], MOSEL is described and compared with MONA 0.2, which provided inspiration for the MOSEL project. Apparently, there have been only few applications of MOSEL. AMORE [34] (see `http://www.informatik.uni-kiel.de/inf/Thomas/amore.html`) is a library of automata theoretic algorithms, resembling those used in MONA. AMORE also provides functionality for regular expressions and monoids, but is not tied to the automaton–logic connection. Glenn and Gasarch [18] have in 1997—apparently independently of MONA and MOSEL—implemented a decision procedure for WS1S, basically as the one in MONA, but without using BDDs or other sophisticated techniques. The SHASTA tool from 1998 is based upon the same ideas as MONA. It is used as an engine for Presburger Arithmetic [45].

Furthermore, MONA has provided the foundation of or been integrated into a range of other tools: FIDO [33], LISA [2], DCVALID [42], FMONA [8], STTOOLS [43], PEN [40], PAX [5], PVS [41], and ISABELLE [3].

2. The Automaton–Logic Connection

Being a variation of first-order logic, WS1S is a formalism with quantifiers and boolean connectives. First-order terms denote natural numbers, which can be compared and subjected to addition with constants. Also, WS1S allows second-order terms, which are interpreted as finite sets of numbers. The actual MONA syntax is a rich notation with constructs such as set constants, predicates and macros, modulo operations, and let-bindings. If all such syntactic sugar is peeled off, the formulas are "flattened" (so that there are no nested terms), and first-order terms are encoded as second-order terms, the logic reduces to a simple "core" language:

$$\begin{array}{rclclcl} \phi & ::= & \texttt{\~{}}\phi' & \mid & \phi' \;\texttt{\&}\; \phi'' & \mid & \texttt{ex2}\ X_i : \phi' \\ & \mid & X_i\ \texttt{sub}\ X_j & \mid & X_i \;\texttt{=}X_j \setminus X_k & \mid & X_i \;\texttt{=}X_j\ \texttt{+1} \end{array}$$

where X ranges over a set of second-order variables.

Given a fixed main formula ϕ_0, we define its semantics inductively relative to a string w over the alphabet $\mathbb{B}^k$, where $\mathbb{B} = \{0,1\}$ and k is the number of variables in ϕ_0. We assume every variable of ϕ_0 is assigned a unique index in the range $1, 2, .., k$, and that X_i denotes the variable with index i. The projection of a string w onto the i'th component is called the X_i-track of w. A string w determines an interpretation $w(X_i)$ of X_i defined as the finite set $\{j \mid \text{the } j\text{th bit in the } X_i\text{-track is } 1\}$.

The semantics of a formula ϕ in the core language can now be defined inductively relative to an interpretation w. We use the notation $w \vDash \phi$ (which is read: w satisfies ϕ) if the interpretation defined by w makes ϕ true:

$$\begin{array}{lll}
w \vDash \texttt{~}\phi' & \text{iff} & w \nvDash \phi' \\
w \vDash \phi' \ \texttt{\&}\ \phi'' & \text{iff} & w \vDash \phi' \text{ and } w \vDash \phi'' \\
w \vDash \texttt{ex2}\ X_i : \phi' & \text{iff} & \exists \text{ finite } M \subseteq \mathbb{N} : w[X_i \mapsto M] \vDash \phi' \\
w \vDash X_i \ \texttt{sub}\ X_j & \text{iff} & w(X_i) \subseteq w(X_j) \\
w \vDash X_i = X_j \backslash X_k & \text{iff} & w(X_i) = w(X_j) \backslash w(X_k) \\
w \vDash X_i = X_j \ \texttt{+1} & \text{iff} & w(X_i) = \{j+1 \mid j \in w(X_j)\}
\end{array}$$

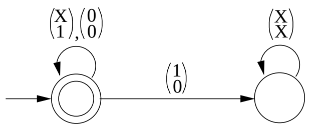
The notation $w[X_i \mapsto M]$ is used for the shortest string that interprets all variables X_j where $j \neq i$ as w does, but interprets X_i as M.

The language $\mathcal{L}(\phi)$ of a formula ϕ can be defined as the set of satisfying strings: $\mathcal{L}(\phi) = \{w \mid w \vDash \phi\}$. By induction in the formula, we can now construct a minimal deterministic finite-state automaton (DFA) A such that $\mathcal{L}(A) = \mathcal{L}(\phi)$, where $\mathcal{L}(A)$ is the language recognized by A.

For the atomic formulas, we show just one example: the automaton for the formula $\phi = X_i \ \texttt{sub}\ X_j$ in the case where $i = 1$ and $j = 2$. The automaton must recognize the language

$$\mathcal{L}(X_1 \ \texttt{sub}\ X_2) = \{w \in (\mathbb{B}^k)^* \mid \text{for all letters in } w\text{: if the first component is 1, then so is the second }\}$$

Such an automaton is:



The other atomic formulas are treated similarly. The composite formulas are translated as follows:

$\phi = \texttt{~}\phi'$ Negation of a formula corresponds to automaton complementation. In Mona, this is implemented trivially by flipping accepting and rejecting states.

$\phi = \phi' \ \texttt{\&}\ \phi''$ Conjunction corresponds to language intersection. In Mona, this is implemented with a standard automaton product construction generating only the reachable product states. The resulting automaton is minimized.

$\phi = \texttt{ex2}\ X_i \texttt{:}\ \phi'$ Existential quantification corresponds to a simple quotient operation followed by a projection operation. The quotient operation takes care of the problem that the only strings satisfying ϕ' may be longer than those satisfying `ex2` $X_i : \phi'$. The projection operation removes the "track" belonging to X_i, resulting in a nondeterministic automaton, which is subsequently determinized using the subset construction operation, and finally minimized.

This presentation is a simplified version of the procedure actually used in MONA. For more detail, see the MONA User Manual [29].

When the minimal automaton A_0 corresponding to ϕ_0 has been constructed, validity of ϕ_0 can be checked simply by observing whether A_0 is the one-state automaton accepting everything. If ϕ_0 is not valid, a (minimal) counter-example can be constructed by finding a (minimal) path in A_0 from the initial state to a non-accepting state.

WS2S is the generalization of WS1S from linear- to binary-tree-shaped structures [47, 14, 48]. Seen at the "core language" level, WS2S is obtained from WS1S by replacing the single successor predicate by two successor predicates, for *left* and *right* successor respectively. This logic is also decidable by the automaton–logic connection, but using tree automata instead of string automata. The MONA tool also implements this decision procedure.

There is a subtle difference between WS1S, the logic now used in MONA, and M2L-Str, the logic used in early experimental versions [48, 6, 15]. (The difference between WS2S and M2L-Tree is similar.) In WS1S, formulas are interpreted over *infinite string* models (but quantification is restricted to finite sets only). In M2L-Str, formulas are instead interpreted over *finite string* models. That is, the universe is not the whole set of naturals $\mathbb{N}$, but a bounded subset $\{0, \ldots, n-1\}$, where n is defined by the length of the string. The decision procedure for M2L-Str is almost the same as for WS1S, only slightly simpler: the quotient operation (before projection) is just omitted. From the language point of view, M2L-Str corresponds exactly to the regular languages (all formulas correspond to automata *and* vice versa), and WS1S corresponds to those regular languages that are closed under concatenation by 0's. These properties make M2L-Str preferable for some applications [4, 44]. However, the fact that not all positions have a successor often makes M2L-Str rather unnatural to use. Being more closely tied to arithmetic, the WS1S semantics is easier to understand. Also, for instance Presburger Arithmetic can easily be encoded in WS1S whereas there is no obvious encoding in M2L-Str.

Notice that the most significant source of complexity in this decision procedure is the quantifiers, or more precisely, the automaton determinization. Each quantifier can cause an exponential blow-up in the number of automaton states, so in the worst case, this decision procedure has a non-elementary complexity. Furthermore, we cannot hope for a fundamentally better decision procedure since this is also the lower bound for the WS1S decision problem [36]. However, as we will show, even constant factors of improvement can make significant differences in practice.

To make matters even worse (and the challenge the more interesting), the implementation also has to deal with automata with huge alphabets. As mentioned, if ϕ_0 has k free variables, the alphabet is $\mathbb{B}^k$. Standard automaton packages cannot handle alphabets of that size, for typical values of k.

3. Benchmark Formulas

The experiments presented in the following section are based on twelve benchmark formulas, here shown with their sizes, the logics they are using, and their time and space consumptions when processed by Mona 1.4 (on a 296MHz UltraSPARC with 1GB RAM):

Benchmark	Name	Size	Logic	Time	Space
A	`dflipflop.mona`	2 KB	WS1S (M2L-Str)	0.4 sec	3 MB
B	`euclid.mona`	6 KB	WS1S (Presburger)	33.1 sec	217 MB
C	`fischer_mutex.mona`	43 KB	WS1S	15.1 sec	13 MB
D	`html3_grammar.mona`	39 KB	WS2S (WSRT)	137.1 sec	208 MB
E	`lift_controller.mona`	36 KB	WS1S	8.0 sec	15 MB
F	`mcnc91_bbsse.mona`	9 KB	WS1S	13.2 sec	17 MB
G	`reverse_linear.mona`	11 KB	WS1S (M2L-Str)	3.2 sec	4 MB
H	`search_tree.mona`	19 KB	WS2S (WSRT)	30.4 sec	5 MB
I	`sliding_window.mona`	64 KB	WS1S	40.3 sec	59 MB
J	`szymanski_acc.mona`	144 KB	WS1S	20.6 sec	9 MB
K	`von_neumann_adder.mona`	5 KB	WS1S	139.9 sec	116 MB
L	`xbar_theory.mona`	14 KB	WS2S	136.4 sec	518 MB

The benchmarks have been picked from a large variety of Mona applications ranging from hardware verification to encoding of natural languages.

`dflipflop.mona` – a verification of a D-type flip-flop circuit [4]. Provided by Abdel Ayari.

`euclid.mona` – an encoding in Presburger arithmetic of six steps of reachability on a machine that implements Euclid's GCD algorithm [45]. Provided by Tom Shiple.

`fischer_mutex.mona` and `lift_controller.mona` – duration calculus encodings of Fischer's mutual exclusion algorithm and a mine pump controller, translated to Mona code [42]. Provided by Paritosh Pandya.

`html3_grammar.mona` – a tree-logic encoding of the HTML 3.0 grammar annotated with 10 parse-tree formulas [13]. Provided by Niels Damgaard.

`reverse_linear.mona` – verifies correctness of a C program reversing a pointer-linked list [22].

`search_tree.mona` – verifies correctness of a C program deleting a node from a search tree [16].

`sliding_window.mona` – verifies correctness of a sliding window network protocol [46]. Provided by Mark Smith.

`szymanski_acc.mona` – validation of the parameterized Szymanski problem using an accelerated iterative analysis [8]. Provided by Mamoun Filali-Amine.

`von_neumann_adder.mona` and `mcnc91_bbsse.mona` – verification of sequential hardware circuits; the first verifies that an 8-bit von Neumann adder is equivalent to a standard carry-chain adder, the second is a benchmark from MCNC91 [49]. Provided by Sebastian Mödersheim.

`xbar_theory.mona` – encodes a part of a theory of natural languages in the Chomsky tradition. It was used to verify the theory and led to the discovery of mistakes in the original formalization [39]. Provided by Frank Morawietz.

We will use these benchmarks to illustrate the effects of the various implementation "secrets" by comparing the efficiency of MONA shown in the table above with that obtained by handicapping the MONA implementation by not using the techniques.

4. Implementation Secrets

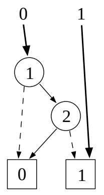
The MONA implementation has been developed and tuned over a period of six years. Many large and small ideas have contributed to a combined speedup of several orders of magnitude. Improvements have taken place at all levels, which we illustrate with the following seven examples from different phases of the compilation and execution of formulas.

To enable comparisons, we summarize the effect of each implementation "secret" by a single dimensionless number for each benchmark formula. Usually, this is simply the speedup factor, but in some cases where the numerator is not available, we argue for a more synthetic measure. If a benchmark cannot run on our machine with 1GB of memory, it is assigned time ∞.

4.1. BDD-based automata representation

The very first attempt to implement the decision procedure used a representation based on conjunctive normal forms—however this quickly showed to be very inefficient. The first actually useful version of the MONA tool was the 1995 experimental ML-version [20]. The reason for the success was the novel representation of automata based on (reduced and ordered) BDDs (Binary Decision Diagrams) [9, 10] for addressing the problem of large alphabets. In addition, the representation allows some specialized algorithms to be applied [32, 26].

A BDD is a graph representing a boolean function. The BDD representation has some extremely convenient properties, such as compactness and canonicity, and it allows efficient manipulation. BDDs have successfully been used in a long range of verification techniques (a popular one is [35]). In MONA, a special form of BDDs, called *shared multi-terminal* BDDs, or SMBDDs are used. As an example, the transition function of the tiny automaton shown in Section 2 is represented in MONA as the following SMBDD:



The roots and the leaves represent the states. Each root has an edge to the node representing its alphabet part of the transition function. For the other edges, dashed represents 0 and solid represents 1. As an example, from state 0, the transition labeled $\binom{1}{0}$ leads to state 1. In this way, states are still represented explicitly, but the transitions are represented symbolically, in a compact way.

It's reasonable to ask: "What would happen if we had simply represented the transition tables in a standard fashion, that is, a row for each state and a column for each letter?". Under this point of view, it makes sense to define a letter for each bit-pattern assignment to the free variables of a subformula (as opposed to the larger set of all variables bound by an outer quantifier). We have instrumented MONA to measure the sum of the number of entries of all such automata transition tables constructed during a run of a version of MONA without BDDs:

	Misses	Table entries	Effect
A	397,472	237,006	0.6
B	48,347,395	2,973,118	0.1
C	46,080,139	1,376,499,745,600	29,871.9
E	19,208,299	290,999,305,488	15,149.7
F	39,942,638	2,844,513,432,416,357,974,016	71,214,961,626,128.9
G	561,202	912,194	1.6
I	95,730,831	116,387,431,997,281,136	1,215,777,934.7
J	24,619,563	15,424,761,908	626.5
K	250,971,828	2,544,758,557,238,438	10,139,618.4

In Section 4.2, we describe the importance of cache awareness, which motivates the number of cache misses as a reasonable efficiency measure. "Misses" is the number of cache misses in our BDD-based implementation, and "Table entries" is the total number of table entries in the naive implementation. To roughly estimate the effect of the BDD-representation, we conservatively assume that each table entry results in just a single cache miss; thus, "Effect" compares "Table entries" to "Misses". The few instances where the effect is less than one correctly identify benchmark formulas where the BDDs are less necessary, but are also artifacts of our conservative assumption. Conversely, the extremely high effects are associated with formulas that could not possibly be decided without BDDs. Of course, the use of BDD-structures completely dominates all other optimizations, since no implementation could realistically be based on the naive table representation.

The BDD-representation was the first breakthrough of the MONA implementation, and the other "secrets" should really be viewed with this as baseline.

4.2. Cache-conscious data structures

The data structure used to represent the BDDs for transition functions has been carefully tuned to minimize the number of cache misses that occur. This effort is motivated in earlier work [32], where it is determined that the number of cache misses during unary and binary BDD apply steps totally dominates the running time.

In fact, we argued that if A1 is the number of unary apply steps and A2 is the number of binary apply steps, then there exist constant m, c_1, and c_2 such that the total running time is approximately $m(c_1 \cdot \mathrm{A1} + c_2 \cdot \mathrm{A2})$. Here, m is the machine dependent delay incurred by an L2 cache miss, and c_1 and c_2 are the average number of cache misses for unary and binary apply steps. This estimate is based the assumption that time incurred for manipulating auxiliary data structures, such as those used for describing subsets in the determinization construction, is insignificant. For the machine we have used for experiments, it is by a small C utility determined that $m = 0.43\mu s$. In our BDD implementation, explained in [32], we have estimated from algorithmic considerations that $c_1 = 1.7$ and $c_2 = 3$ (the binary apply may entail the use of unary apply steps for doubling tables that were too small—these steps are not counted towards the time for binary apply steps, and that is why we can use the figure $c_2 = 3$); we also estimated that for an earlier conventional implementation, the numbers were $c_1 = 6.7$ and $c_2 = 7.3$. The main reason for this difference is that our specialized package stores nodes directly under their hash address to minimize cache misses; traditional BDD packages store BDD nodes individually with the hash table containing pointers to them—roughly doubling the time it takes to process a node. We no longer support the conventional BDD implementation, so to measure the effect of cache-consciousness, we must use the above formula to estimate the running times that would have been obtained today.

In the following experiment, we have instrumented Mona to obtain the exact numbers of apply steps:

	Apply1	Apply2	Misses	Auto	Predicted	Conventional	Effect
A	183,949	28,253	397,472	0.2 sec	0.2 sec	0.6 sec	3.0
B	21,908,722	3,700,856	48,347,395	32.8 sec	20.8 sec	74.7 sec	3.6
C	24,585,292	1,428,381	46,080,139	14.2 sec	19.8 sec	75.2 sec	3.8
E	9,847007	822,796	19,208,299	7.7 sec	8.2 sec	30.9 sec	3.8
F	13,406,047	5,717,453	39,942,638	12.8 sec	17.2 sec	56.6 sec	3.3
G	233,566	54,814	561,504	0.5 sec	0.3 sec	0.8 sec	2.7
I	36,629,195	11,153,733	95,730,831	37.0 sec	41.2 sec	140.5 sec	3.4
J	10,497,759	2,257,791	24,619,563	11.6 sec	10.6 sec	37.3 sec	3.5
K	129,126,447	10,485,623	250,971,828	137.4 sec	107.9 sec	404.7 sec	3.8

"Apply1" is the number of unary apply steps; "Apply2" is the number of binary apply steps; "Misses" is the number of cache misses predicted by the formula above; "Auto" is the part of the actual running time involved in automata constructions; "Predicted" is the running time predicted from the cache misses alone; "Conventional" is the predicted running time for a conventional BDD implementation that was not cache-conscious; and "Effect" is "Conventional" compared to "Predicted". In most cases, the actual running time is close to the predicted one (within 25%). Note that there are instances where the actual time is about 50% larger than the estimated time: benchmark B involves a lengthy subset construction on an automaton with small BDDs—thus it violates the assumption that the time handling accessory data structures is insignificant; similarly, benchmark G also consists of many automata with few BDD nodes prone to violating the assumption.

In an independent comparison [45] it was noted that Mona was consistently twice as fast as a specially designed automaton package based on a BDD package considered efficient. In [32], the comparison to a traditional BDD package yielded a factor 5 speedup.

4.3. Eager minimization

When Mona inductively translates formulas to automata, a Myhill-Nerode minimization is performed after every product and projection operation. Naturally, it is preferable to operate with as small automata as possible, but our strategy may seem excessive since minimization often exceeds 50% of the total running time. This suspicion is strengthened by the fact that Mona automata by construction contain only reachable states; thus, minimization only collapses redundant states.

Three alternative strategies to the eager one currently used by Mona would be to perform only the very final minimization, only the ones occurring after projection operations, or only the ones occurring after product operations. Many other heuristics could of course also be considered. The following table results from such an investigation:

	Time				Effect
	Only final	After project	After product	Always	
A	∞	∞	0.6 sec	0.4 sec	1.5
B	∞	∞	∞	33.1 sec	∞
C	∞	∞	32.3 sec	15.1 sec	2.1
D	∞	∞	290.6 sec	137.1 sec	2.1
E	∞	∞	19.4 sec	8.0 sec	2.4
F	∞	∞	36.7 sec	13.2 sec	2.8
G	∞	∞	5.8 sec	3.2 sec	1.8
H	∞	∞	59.6 sec	30.4 sec	2.0
I	∞	∞	74.4 sec	40.3 sec	1.8
J	∞	∞	36.3 sec	20.6 sec	1.8
K	∞	∞	142.3 sec	139.9 sec	1.0
L	∞	∞	∞	136.4 sec	∞

"Only final" is the running time when minimization is only performed as the final step of the translation; "After project" is the running time when minimization is also performed after every projection operation; "After product" is the running time when minimization is instead performed after every product operation; "Always" is the time when minimization is performed eagerly; and "Effect" is the "After product" time compared to the "Always" time (since the other two strategies are clearly hopeless). Eager minimization is seen to be always beneficial and in some cases essential for the benchmark formulas.

4.4. Guided tree automata

Tree automata are inherently more computationally expensive because of their three-dimensional transition tables. We have used a technique of factorization of state spaces to split big tree automata into smaller ones. The basic idea, which may result in exponential savings, is explained in [7, 29]. To exploit this feature, the MONA programmer must manually specify a *guide*, which is a top-down tree automaton that assigns state spaces to the nodes of a tree. However, when using the WSRT logic, a canonical guide is automatically generated. For our two WSRT benchmarks, we measure the effect of this canonical guide:

	Without guide	With guide	Effect
D	584.0 sec	137.1 sec	4.3
H	∞	30.4 sec	∞

"Without guide" shows the running time without any guide, while "With guide" shows the running time with the canonical WSRT guide; "Effect" shows the "Without guide" time compared to the "With guide" time. We have only a small sample space here, but clearly guides are very useful. This is hardly surprising, since they may yield an asymptotic improvement in running time.

4.5. DAGification

Internally, MONA is divided into a front-end and a back-end. The front-end parses the input and builds a data structure representing the automata-theoretic operations that will calculate the resulting automaton. The back-end then inductively carries out these operations.

The generated data structure is often seen to contain many common subformulas. This is particularly true when they are compared relative to *signature equivalence*, which holds for two formulas ϕ and ϕ' if there is an order-preserving renaming of the variables in ϕ (increasing with respect to the indices of the variables) such that the representations of ϕ and ϕ' become identical.

A property of the BDD representation is that the automata corresponding to signature-equivalent trees are isomorphic in the sense that only the node indices differ. This means that intermediate results can be reused by simple exchanges of node indices. For this reason, MONA represents the formulas in a DAG (Directed Acyclic Graph), not a tree. The DAG is conceptually constructed from the tree using a bottom-up collapsing process, based on the signature equivalence relation as described in [15].

Clearly, constructing the DAG instead of the tree incurs some overhead, but the following experiments show that the benefits are significantly larger:

	Nodes		Time		Effect
	Tree	DAG	Tree	DAG	
A	2,532	296	1.7 sec	0.4 sec	4.3
B	873	259	79.2 sec	33.1 sec	2.4
C	5,432	461	40.1 sec	15.1 sec	2.7
D	3,038	270	∞	137.1 sec	∞
E	4,560	505	20.5 sec	8.0 sec	2.6
F	1,997	505	49.1 sec	13.2 sec	3.7
G	56,932	1,199	∞	3.2 sec	∞
H	8,180	743	∞	30.4 sec	∞
I	14,058	1,396	107.1 sec	40.3 sec	2.7
J	278,116	6,314	∞	20.6 sec	∞
K	777	273	284.0 sec	139.9 sec	2.0
L	1,504	388	∞	136.4 sec	∞

"Nodes" shows the number of nodes in the representation of the formula. "Tree" is the number of nodes using an explicit tree representation, while "DAG" is the number of nodes after DAGification. "Time" shows the running times for the same two cases. "Effect" shows the "Tree" running time compared to the "DAG" running time. From the differences in the number of nodes, one might expect the total effect to be larger, however DAGification is mainly effective on small formulas where the corresponding automata typically are also smaller. Nevertheless, the DAGification process is seen to provide a substantial and often essential gain in efficiency.

The effects reported sometimes benefit from the fact that the restriction technique presented in the following subsection knowingly generates redundant formulas. This explains some of the failures observed.

4.6. Three-valued logic and automata

The standard technique for dealing with first-order variables in monadic second-order logics is to encode them as second-order variables, typically as singletons. However, that raises the issue of *restrictions*: the common phenomenon that a formula ϕ makes sense, relative to some exterior conditions, only when an associated restriction holds. The restriction is also a formula, and the main issue is that

ϕ is now essentially undefined outside the restriction. In the case of first-order variables encoded as second-order variables, the restriction is that these variables are singletons. We experienced the same situation trying to emulate M2L-Str in WS1S, causing state-space explosions.

The nature of these problems is technical, but fortunately they can be solved through a theory of restriction couched in a three-valued logic [27]. Under this view, a *restricted subformula* ϕ is associated with a restriction ϕ_R. If, for some valuation, ϕ_R does not hold, then formulas containing ϕ are assigned a new third truth value "don't-care". This three-valued logic carries over to the MONA notion of automata—in addition to accept and reject states, they also have "don't-care" states. A special `restrict(`ϕ_R`)` operation is used for converting reject states to "don't-care" states for the restriction formulas, and the other automaton operations are modified to ensure that non-acceptance of restrictions is propagated properly.

This gives a cleaner semantics to the restriction phenomenon, and furthermore avoids the state-space explosions mentioned above. According to [27], we can guarantee that the WS1S framework handles all formulas written in M2L-Str, even with intermediate automata that are no bigger than when using the traditional M2L-Str decision procedure. MONA uses the same technique for the tree logics, WS2S and M2L-Tree.

We refer to [27] for the full theory of three-valued logic and automata. Unfortunately, there is no way of disabling this feature to provide a quantitative comparison.

4.7. Formula reductions

Formula reduction is a means of "optimizing" the formulas in the DAG before translating them into automata. The reductions are based on a syntactic analysis that attempts to identify valid subformulas and equivalences among subformulas.

There are some non-obvious choices here. How should computation resources be apportioned to the reduction phase and to the automata calculation phase? Must reductions guarantee that automata calculations become faster? Should the two phases interact? Our answers are based on some trial and error along with some provisions to cope with subtle interactions with other of our optimization secrets.

MONA 1.4 performs three kinds of formula reductions: 1) simple equality and boolean reductions, 2) special quantifier reductions, and 3) special conjunction reductions. The first kind can be described by simple rewrite rules (only some typical ones are shown):

$$\begin{array}{rcl@{\qquad\qquad}rcl}
X_i \texttt{ = } X_i & \rightsquigarrow & \texttt{true} & \phi \texttt{ \& } \phi & \rightsquigarrow & \phi \\
\texttt{true \& } \phi & \rightsquigarrow & \phi & \texttt{\~{}\~{}}\phi & \rightsquigarrow & \phi \\
\texttt{false \& } \phi & \rightsquigarrow & \texttt{false} & \texttt{\~{}false} & \rightsquigarrow & \texttt{true}
\end{array}$$

These rewrite steps are guaranteed to reduce complexity, but will not cause significant improvements in running time, since they all either deal with constant size automata or rarely apply in realistic situations. Nevertheless, they are extremely cheap, and they may yield small improvements, in particular on machine generated MONA code.

The second kind of reduction can potentially cause tremendous improvements. As mentioned, the non-elementary complexity of the decision procedure is caused by the automaton projection operations, which stem from quantifiers. The accompanying determinization construction may cause an exponential blow-up in automaton size. Our basic idea is to apply a rewrite step resembling *let*-reduction, which removes quantifiers:

$$\texttt{ex2}\ X_i : \phi \quad \rightsquigarrow \quad \phi[T/X_i] \quad \text{provided that } \phi \texttt{ => } X_i \texttt{ = } T \text{ is valid, and } T \text{ is some term satisfying } FV(T) \subseteq FV(\phi)$$

where $FV(\cdot)$ denotes the set of free variables. For several reasons, this is not the way to proceed in practice. First of all, finding terms T satisfying the side condition can be an expensive task, in worst case non-elementary. Secondly, the translation into automata requires the formulas to be "flattened" by introduction of quantifiers such that there are no nested terms. So, if the substitution $\phi[T/X]$ generates nested terms, then the removed quantifier is recreated by the translation. Thirdly, when the rewrite rule applies in practice, ϕ usually has a particular structure as reflected in the following more restrictive rewrite rule chosen in MONA:

$$\texttt{ex2}\ X_i : \phi \quad \rightsquigarrow \quad \phi[X_j/X_i] \quad \text{provided that } \phi \equiv \cdots \texttt{ \& } X_i \texttt{ = } X_j \texttt{ \& } \cdots \text{ and } X_j \text{ is some variable other than } X_i$$

In contrast to equality and boolean reductions, this rule is not guaranteed to improve performance, since substitutions may cause the DAG reuse degree to decrease.

The third kind of reduction applies to conjunctions, of which there are two special sources. One is the formula flattening just mentioned; the other is the formula restriction technique mentioned in Section 4.6. Both typically introduce many new conjunctions. Studies of a graphical representation of the formula DAGs (MONA can create such graphs automatically) led us to believe that many of these new conjunctions are redundant. A typical rewrite rule addressing such redundant conjunctions is the following:

$$\phi_1 \texttt{ \& } \phi_2 \quad \rightsquigarrow \quad \phi_1 \quad \text{provided that } \mathit{nonrestr}(\phi_2) \subseteq \mathit{nonrestr}(\phi_1) \cup \mathit{restr}(\phi_1) \text{ and } \mathit{restr}(\phi_2) \subseteq \mathit{restr}(\phi_1)$$

Here, $\mathit{restr}(\phi)$ is the set of `restrict(·)` conjuncts in ϕ, and $\mathit{nonrestr}(\phi)$ is the set of non-`restrict(·)` conjuncts in ϕ. This reduction states that it is sufficient to assert ϕ_1 when ϕ_1& ϕ_2 was originally asserted in situations where the non-restricted conjuncts of ϕ_2 are already conjuncts of ϕ_1—whether restricted or not—and the restricted conjuncts of ϕ_2 are also restricted conjuncts of ϕ_1.

All rewrite rules mentioned here have the property that they cannot "do any harm", that is, have a negative impact on the automaton sizes. (They can however damage the reuse factor obtained by the DAGification, but this is rarely a problem in practice.) A different kind of rewrite rules could be obtained using heuristics—this will be investigated in the future.

With the DAG representation of formulas, the reductions just described can be implemented relatively easily in MONA. The table below shows the effects of performing the reductions on the benchmark formulas:

	Hits			Time					Effect
	Simple	Quant.	Conj.	None	Simple	Quant.	Conj.	All	
A	12	8	22	0.8 sec	0.7 sec	0.7 sec	0.7 sec	0.4 sec	2.0
B	10	45	0	58.2 sec	58.8 sec	56.2 sec	56.8 sec	33.1 sec	1.8
C	9	13	8	43.7 sec	41.9 sec	37.1 sec	42.9 sec	15.1 sec	2.9
D	4	28	27	542.7 sec	536.1 sec	296.0 sec	404.7 sec	137.1 sec	4.0
E	5	6	19	22.6 sec	23.4 sec	16.6 sec	22.7 sec	8.0 sec	2.8
F	3	1	1	28.3 sec	29.9 sec	27.0 sec	27.2 sec	13.2 sec	2.1
G	65	318	191	6.1 sec	5.9 sec	6.1 sec	5.9 sec	3.2 sec	1.9
H	35	32	81	104.1 sec	102.6 sec	71.0 sec	98.5 sec	30.4 sec	3.4
I	102	218	7	76.2 sec	76.5 sec	75.0 sec	76.0 sec	40.3 sec	1.9
J	91	0	1	37.3 sec	37.9 sec	37.6 sec	37.0 sec	20.6 sec	1.9
K	9	4	1	313.7 sec	267.9 sec	240.3 sec	302.6 sec	139.9 sec	2.3
L	4	4	18	∞	∞	∞	∞	136.4 sec	∞

"Hits" shows the number of times each of the three kinds of reduction is performed; "Time" shows the total running time in the cases where no reductions are performed, only the first kind of reductions are performed, only the second, only the third, and all of them together. "Effect" shows the "None" times compared to the "All" times. All benchmarks gain from formula reductions, and in a single example this technique is even necessary. Note that most often all three kinds of reductions must act in unison to obtain significant effects.

A general benefit from formula reductions is that tools generating MONA formulas from other formalisms may generate naive and voluminous output while leaving optimizations to MONA. In particular, tools may use existential quantifiers to bind terms to fresh variables, knowing that MONA will take care of the required optimization.

5. Future Developments

Several of the techniques described in the previous section can be further refined of course. The most promising ideas seem however to concentrate on the BDD representation. In the following, we describe three such ideas.

It is a well-known fact [9] that the ordering of variables in the BDD automata representation has a strong influence on the number of BDD nodes required. The impact of choosing a good ordering can be an exponential improvement in running times. Finding the optimal ordering is an NP-complete problem, but we plan to experiment with the heuristics that have been suggested [12].

We have sometimes been asked: "Why don't you encode the states of the automata in BDDs, since that is a central technique in model checking?". The reason is that there is no obvious structure to the state space in most cases that would lend itself towards an efficient BDD representation. For example, consider the consequences of a subset construction or a minimization construction, where similar states are collapsed; in either case, it is not obvious how to represent the new state. However, the ideas are worth investigating.

For our tree automata, we have experimentally observed that the use of guides produce a large number of component automata many of which are almost identical. We will study how to compress this representation using a BDD-like global structure.

6. Conclusion

The presented techniques reflect a lengthy Darwinian development process of the MONA tool in which only robust and useful ideas have survived. We have not mentioned here the many ideas that failed or were surpassed by other techniques. Our experiences confirm the maxim that optimizations must be carried out at all levels and that no single silver bullet is sufficient. We are confident that further improvements are still possible and that other interesting applications will be made.

Acknowledgments

Many people have contributed to the development of MONA, in particular we are grateful to David Basin, Morten Biehl, Jacob Elgaard, Jesper Gulmann, Jacob Jensen, Michael Jørgensen, Bob Paige, Theis Rauhe, and Anders Sandholm. We also thank the MONA users who kindly provided the benchmark formulas.

References

1. Abdelwaheb Ayari, David Basin, and Stefan Friedrich. Structural and behavioral modeling with monadic logics. In *The Twenty-Ninth IEEE International Symposium on Multiple-Valued Logic*. IEEE Computer Society, 1999.
2. Abdelwaheb Ayari, David Basin, and Andreas Podelski. LISA: A specification language based on WS2S. In *CSL '97*, LNCS, 1998.
3. David Basin and Stefan Friedrich. Combining WS1S and HOL. In *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*. Research Studies Press/Wiley, 2000.
4. David Basin and Nils Klarlund. Automata based symbolic reasoning in hardware verification. *Formal Methods In System Design*, 13:255–288, 1998. Extended version of: "Hardware verification using monadic second-order logic," *CAV '95*, LNCS 939.
5. Kai Baukus, Karsten Stahl, Saddek Bensalem, and Yassine Lakhnech. Abstracting WS1S systems to verify parameterized networks. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS '00*, volume 1785 of *LNCS*, 2000.
6. Morten Biehl, Nils Klarlund, and Theis Rauhe. Mona: decidable arithmetic in practice (demo). In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium*, volume 1135 of *LNCS*, 1996.
7. Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, WIA '96*, volume 1260 of *LNCS*, 1997.
8. Jean-Paul Bodeveix and Mamoun Filali. FMona: a tool for expressing validation techniques over infinite state systems. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS '00*, volume 1785 of *LNCS*, 2000.
9. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.
10. R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing surveys*, 24(3):293–318, September 1992.
11. J.R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.
12. K.M. Butler, D.E. Ross, R. Kapur, and M.R. Mercer. Heuristics to compute variable

orderings for efficient manipulation of ordered binary decision diagrams. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1991.

13. Niels Damgaard, Nils Klarlund, and Michael I. Schwartzbach. YakYak: Parsing with logical side constraints. In *Proceedings of DLT'99*, 1999.

14. J. Doner. Tree acceptors and some of their applications. *J. Comput. System Sci.*, 4:406–451, 1970.

15. Jacob Elgaard, Nils Klarlund, and Anders Møller. Mona 1.x: new techniques for WS1S and WS2S. In *Computer Aided Verification, CAV '98*, volume 1427 of *LNCS*, 1998.

16. Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proceedings of European Symposium on Programming Languages and Systems, ESOP '00*, volume 1782 of *LNCS*, 2000.

17. C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98:21–52, 1961.

18. J. Glenn and W. Gasarch. Implementing WS1S via finite automata. In *Automata Implementation, WIA '96*, volume 1260 of *LNCS*, 1997.

19. Aarti Gupta and Allan L. Fisher. Representation and symbolic manipulation of linearly inductive boolean functions. In *Proceedings of the IEEE International Conference on Computer-Aided Design.* IEEE Computer Society Press, 1993.

20. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS '95*, volume 1019 of *LNCS*, 1996.

21. Thomas Hune and Anders Sandholm. A case study on using automata in control synthesis. In *Fundamental Approaches to Software Engineering, FASE'00*, volume 1783 of *LNCS*, 2000.

22. Jacob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *PLDI '97*, 1997.

23. Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS '00*, volume 1785 of *LNCS*, 2000.

24. P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. MOSEL: a flexible toolset for Monadic Second-order Logic. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '97*, volume 1217 of *LNCS*, 1997.

25. Nils Klarlund. Mona & Fido: The logic-automaton connection in practice. In *Computer Science Logic, CSL '97*, volume 1414 of *LNCS*, 1998.

26. Nils Klarlund. An $n \log n$ algorithm for online BDD refinement. *Journal of Algorithms*, 32:133–154, 1999. Abbreviated version in CAV '97, LNCS 1254.

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

28. Nils Klarlund, Jari Koistinen, and Michael I. Schwartzbach. Formal design constraints. In *Proceedings of OOPSLA '96*, 1996.

29. Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual.* BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

30. Nils Klarlund, Mogens Nielsen, and Kim Sunesen. Automated logical verification based on trace abstraction. In *Proceedings of PODC '96*, 1996.

31. Nils Klarlund, Mogens Nielsen, and Kim Sunesen. A case study in automated verification based on trace abstractions. In M. Broy, S. Merz, and K. Spies, editors, *Formal System Specification, The RPC-Memory Specification Case Study*, volume 1169 of *LNCS*, pages 341–374. Springer Verlag, 1996.

32. Nils Klarlund and Theis Rauhe. BDD algorithms and cache misses. Technical report, BRICS Report Series RS-96-5, Department of Computer Science, University of Aarhus, 1996.

33. Nils Klarlund and Michael I. Schwartzbach. A domain-specific language for regular sets of strings and trees. *IEEE Transactions On Software Engineering*, 25(3):378–386, 1999.

34. O. Matz, A. Miller, A. Potthoff, W. Thomas, and E. Valkema. Report on the program AMoRE. Technical report, Report 9507, Inst. für Informatik u. Prakt. Mathematik, CAU Kiel, 1995.

35. Ken McMillan. *Symbolic Model Checking.* Kluwer, 1993.

36. A.R. Meyer. Weak monadic second-order theory of successor is not elementary recursive. In R. Parikh, editor, *Logic Colloquium, (Proc. Symposium on Logic, Boston, 1972)*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154, 1975.

37. Anders Møller. MONA project home page. `http://www.brics.dk/mona/`.

38. Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of Conference on Programming Language Design and Implementation, PLDI '01.* ACM, 2001.

39. Frank Morawietz and Tom Cornell. The logic-automaton connection in linguistics. In *Proceedings of LACL '97*, number 1582 in LNAI, 1997.

40. Marcus Nilsson. Analyzing parameterized distributed algorithms. Master's thesis, Department of Computer Systems at Uppsala University, Sweden, 1999.

41. Sam Owre and Harald Ruess. Integrating WS1S with PVS. In *Conference on Computer-Aided Verification, CAV '00*, LNCS, 2000.

42. Paritosh K. Pandya. DCVALID 1.3: The user manual. Technical report, Tata Institute of Fundamental Research, STCS-99/1, 1999.

43. Anders Steen Rasmussen. Symbolic model checking using monadic second order logic as requirement language. Master's thesis, Technical University of Denmark (DTU), 1999. IT-E 821.

44. Anders Sandholm and Michael I. Schwartzbach. Distributed safety controllers for Web services. In *Fundamental Approaches to Software Engineering, FASE'98*, volume 1382 of *LNCS*, 1998.

45. Thomas R. Shiple, James H. Kukula, and Rajeev K. Ranjan. A comparison of Presburger engines for EFSM reachability. In *Computer Aided Verification, CAV '98*, volume 1427 of *LNCS*, 1998.

46. Mark A. Smith and Nils Klarlund. Verification of a sliding window protocol using IOA and Mona, 1999.

47. J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Math. Systems Theory*, 2:57–82, 1968.

48. Wolfgang Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

49. S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. In *Tech. Rep. MCNC*, 1991.