MONA 1.x: New Techniques for WS1S and WS2S

Jacob Elgaard[1], Nils Klarlund[2], and Anders Møller[3]

[1] BRICS, University of Aarhus (elgaard@brics.dk)
[2] AT&T Labs–Research (klarlund@research.att.com)
[3] BRICS, University of Aarhus (amoeller@brics.dk)

Abstract. In this note, we present the first version of the MONA tool to be released in its entirety. The tool now offers decision procedures for both WS1S and WS2S and a completely rewritten front-end. Here, we present some of our techniques, which make calculations couched in WS1S run up to five times faster than with our pre-release tool based on M2L(Str). This suggests that WS1S—with its better semantic properties—is preferable to M2L(Str).

1 Introduction

It has been known for a couple of years that Monadic Second-order Logic interpreted relative to finite strings (M2L(Str)) is an attractive formal and practical vehicle for a variety of verification problems. The formalism is generally easy to use, since it provides Boolean connectives, first and second-order quantifiers and no syntactic restrictions, say, to clausal forms. However, the semantics of the formalism is the source of definitional and practical problems. For example, the concept of a first-order term doesn't even make sense for the empty string since such terms denote positions.

So, it is natural to investigate whether the related logic WS1S (Weak Second-order theory of 1 Succesor) can be used instead. This logic is stronger in that it captures a fragment of arithmetic, and its decision procedure is very similar to that of M2L(Str). Similarly, we would like to explore the practical feasibility of WS2S (Weak Second-order theory of 2 Successors).

In this note, we present some new techniques that we have incorporated into the first full release of the MONA tool. The MONA tool consists of a front-end and two back-ends, one for WS1S and one for WS2S. The front-end parses the MONA program, which consists of predicates (subroutines that are compiled separately), macros, and a main formula. Each back-end implements the automata-theoretic operations that are carried out to decide the formula corresponding to the program.

Since our earlier presentation of the MONA tool [1], we have completely rewritten the front-end, this time in C++ (the earlier version was written in ML). In the old version, the front-end produces a *code tree*, whose internal nodes each describe an automata-theoretic operation—such as a product or subset construction—and whose leaves describe automata corresponding to basic formulas. We implemented optimization techniques (unpublished) based on rewriting of formulas according to logical laws. In this note, we report on an alternative optimization technique, based on building a code DAG instead of a code tree. (A DAG is a directed, acyclic graph.) Experiments show that this technique together with a more efficient handling of predicates yields up to five-fold improvements in compilation time over the old tool.

We also briefly discuss how a M2L(Str) formula can be translated into an essentially equivalent WS1S formula, and we discuss important problems to be addressed.

2 M2L(Str) and WS1S

M2L(Str) A formula of the logic M2L(Str) is interpreted relative to a number $n \geq 0$, which is best thought of as defining the set of positions $\{0, \ldots, n-1\}$ in a string of length n. The core logic consists of first-order terms, second-order terms, and formulas. A *first-order term* t is a variable p, a constant 0 (denoting the position 0, which is the first position in w) or \$ (denoting $n-1$, which is the last position in the string), or of the form $t' \oplus 1$ (denoting $i+1 \mod n$ when t' is a first-order term denoting i). A *second-order term* is either a variable P or of the form $T' \cup T''$. A formula ϕ is either a basic formula of the form $t \in T$ or $T \subset T'$, or of the form $\psi \wedge \chi$, $\neg\psi$, $\exists p : \psi$ (first-order quantification), or $\exists P : \psi$ (second-order quantification). In addition, we allow formulas involving $=$ (between first-order or second-order terms); $<, \leq, >, \geq$ (between first-order terms); Boolean connectives $\Rightarrow, \Leftrightarrow$ and $\vee$; set operations $\cap$, $\setminus$, and $\complement$; $\forall$ quantifiers; etc.

The automaton-logic connection (see [5]) allows us to associate a regular language over $\mathbb{B}^k$, for some $k \geq 0$, to each formula ϕ as follows. We assume that there are k variables that are ordered and that include the free variables in ϕ. Now, a string w of length n over the alphabet $\mathbb{B}^k$ can be viewed as consisting of k *tracks* (or rows), each of length n. The kth track is a bit-pattern that defines the interpretation of the kth variable, assumed to be second-order, as the set of positions i for which the ith bit is 1. Note that a first-order variable can be regarded as a second-order variable restricted to singleton values, so the assumption just made that variables are second-order is not a serious one. The *language* associated with formula ϕ is now the set of all strings that correspond to a satisfying interpretation of the formula. As an example, the formula $P \subseteq Q$ is associated with the regular language

$$(\binom{0}{0} + \binom{0}{1} + \binom{1}{1})^*$$

where the upper track of a string denotes the value of P and the lower track denotes the value of Q. Any language corresponding to a formula is regular, since the languages corresponding to basic formula can be represented by automata, and $\wedge$, $\neg$, and $\exists$ correspond to the automata-theoretic operations of product, complementation, and projection. In the case of a closed formula with no free variables, the regular language degenerates to a set of strings over a unit alphabet. Thus a closed formula essentially denotes a set of numbers.

The proof of regularity just hinted at forms the basis for the decision procedure: each subformula is compiled into a minimum deterministic automaton, see [5]. An automaton representation based on BDDs is at the core of the Mona implementation as discussed in [5]. For each state p in the state space S, a multi-terminal BDD whose leaves are states represents the transition function $a \mapsto \delta(p, a) : \mathbb{B}^k \rightarrow S$ out of p. Each BDD variable corresponds to a first or second-order WS1S variable, and the BBDs are shared among the states. Thus the resulting data structure is a DAG with multiple sources.

The automaton-logic connection (see [5]) allows us to associate a regular language over $\mathbb{B}^k$ to each formula ϕ that has k variables.

WS1S WS1S has the same syntax as M2L(Str) except that there is no $\complement$ operator and $\oplus 1$ is replaced with $+1$. This logic is interpreted in a simpler manner: first-order terms denote natural numbers, and second-order terms denote finite sets of numbers. The automata-theoretic calculations are similar to that of M2L(Str) except for the existential quantifier (see [5]).

From M2L(Str) to WS1S In principle, it is easy to translate a quantifier free M2L(Str) formula ϕ to a formula ϕ' in WS1S with essentially the same meaning: ϕ' is gotten from ϕ by the following steps.

- A conjunct $p \leq \$$, where \$ now is a variable, must be added to any subformula of ϕ containing a first-order variable p.
- Each second-order variable P is left untouched, so that the translated formula will not depend on whether P has any elements greater than \$. However, occurrences of $\emptyset$ must be taken into account; for example, the formula $P = 0$ is translated into $\forall p \leq \$: \neg(p \in P)$ so that the translated formula does not depend on the membership status of numbers in P that are greater than \$. Any use of set complement operator $\complement$ must also be carefully replaced.
- Any occurrence of a subformula involving $\oplus$ such as $p = q \oplus 1$ must be replaced by something that captures the modulo semantics (here: $q < \$ \Rightarrow p = q + 1 \wedge p = \$ \Rightarrow p = 0$).

With such a scheme it can be shown that $\mathcal{I}$ for length $n > 0$ satisfies ϕ if and only if $\mathcal{I}$, augmented by interpreting \$ as $n - 1$, satisfies ϕ'. Unfortunately, in order to preserve this property for all subformulas, we need to conjoin extraneous conditions onto every original subformula. A simpler solution is to conjoin them only for certain strategic places, such as for all basic formulas and all formulas that are directly under a quantifier. We have implemented such heuristics in a tool, s2n, that automatically translates M2L(Str) formulas to WS1S formulas.

3 DAGs for compilation

Code trees can be of the form (among others) `mk-basic-less(`i`,`j`)`, `mk-product` `(`C`,`C'`,`op`)`, or `mk-project(`i`,`C`)`, where i and j are BDD variable indices, op is a Boolean function of two variables, and C and C' are code trees. For example, consider the formula $\exists q : p < q \wedge q < r$. If variable p has index

1, i.e., if it is the 1st variable in the variable ordering, variable q has index 2, and variable r has index 3, then this formula is parsed into a code tree `mk-project(2,mk-product(mk-less(1,2), mk-less(2,3), ∧))`. This tree contains a situation that we would like to avoid: essentially isomorphic subtrees are calculated more than once. In fact, the automaton A for `mk-less(1,2)` is identical to the automaton A' for `mk-less(2,3)` modulo a renaming of variables. In general, we would like to rename the indices in A whenever we need A', since this is a linear operation (whereas building A or A' from the code tree is often not a linear operation).

So, we say that a code tree C is *equivalent* to C' if there is an order-preserving (i.e., increasing), renaming of variables in C' such that C' becomes C. Our goal is produce the DAG that arises naturally from the code tree by collapsing equivalent subtrees. Unfortunately, it takes linear time to calculate the equivalence class of any subtree, and so the total running time becomes quadratic. Therefore, the collapsing process is limited to subtrees for which the number of variable occurrences is less than a user definable parameter ℓ.

MONA offers both pre-compiled subroutines, called *predicates*, and typed macros. A use *name*($\boldsymbol{X}$) of a predicate, where $\boldsymbol{X}$ is a sequence of actual parameters, is translated to a special node of the form `mk-call(`*`name`*`,` $\boldsymbol{X}$`)`. The predicate is then compiled separately given the signature of the call node. The actual parameters are bound to the resulting automaton using a standard binding mechanism: introduction of temporary variables and projection. Additional call nodes with the same signature can then reuse the separately compiled automaton. Call nodes act as leaves with respect to DAGification.

4 Experimental results

We have run a MONA formula, `reverse`, of size 50KB (an automatically generated formula from [3]) through our old MONA (using optimizations) and our new WS1S version with and without DAGification ($\ell = 200$). We also did the experiment on `reverse2`, a version of the formula where all defined predicates were replaced by macros. And, we have run a comparison on a formula representing a parameterized hardware verification problem. The results are (in seconds):

Program	Old MONA	MONA 1.1	w. DAGs	DAG Hits	DAG Misses
`reverse`	17	8.5	3.0	20513	2725
`reverse2`	51	90	45	327328	14320
`hardware`	6.6	5.4	4.7	3284	633

In some cases (like in `reverse2`), the old Mona tool is faster than the new one run without DAGification, since the figures reported for the old apply to the version that carries out formula simplification. The experiments support our claim that WS1S can be as an efficient formalism as M2L(Str). (The underlying BDD-package in the two tools is the same.) Moreover, our DAGs and predicate uses offer substantial benefits, up to a factor five. The hardware example runs only slightly faster, and the improvement is due to the new front-end being quicker.

5 Related and Future Work

There are at least three similar tools reported in the literature: [2] reports on an implementation of WS1S that is not based on BDDs and that therefore is likely not to be as efficient as our tool. The tool in [4] implements M2L(Str) using a different BDD representation, and the tool in [6] implements a decision procedure for WS2S (in Prolog and without BDDs).

There are still several problems and challenges not addressed in the current MONA tool: 1) the semantics of formulas with first-order terms is not appealing, for example, the MONA formula $x1 < x2 \wedge \ldots \wedge x_{n-1} < x_n$ is translated in linear time whereas its negation, $x1 \geq x_2 \vee \ldots \vee x_{n-1} \geq x_n$, is translated in exponential time; 2) there is no reuse of intermediate results from one automaton operation to the next (a general solution to this problem seems to require identification of isomorphic subgraphs, a problem that appears computationally expensive); 3) the automatic translation from M2L(Str) to WS1S by S2N sometimes makes formulas unrunnable for reasons similar to 1), namely that the restrictions a formula is translated under are wrapped into subformulas in unfortunate ways unless the restrictions are reapplied for each intermediate result; 4) the use of formula rewriting (as we did in the earlier MONA version) should be combined with our DAG techniques.

The MONA tool, currently in version 1.2, can be retrieved from `http://www.brics.dk/~mona`, along with further information.

References

1. M. Biehl, N. Klarlund, and T. Rauhe. Mona: decidable arithmetic in practice (short contribution). In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium, LNCS 1135*. Springer Verlag, 1996.
2. J. Glenn and W. Gasarch. Implementing WS1S via finite automata. In *Automata Implementation, WIA '96, Proceedings*, volume 1260 of *LNCS*, 1997.
3. J.L. Jensen, M.E. Jørgensen, N. Klarlund, and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *SIGPLAN '97 Conference on Programming Language Design and Implementation,*, pages 226–234. SIGPLAN, 1997.
4. P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. Mosel: a flexible toolset for Monadic Second-order Logic. In *Computer Aided Verification, CAV '97, Proceedings*, LNCS 1217, 1997.
5. N. Klarlund. Mona & Fido: the logic-automaton connection in practice. In *CSL '97 Proceedings*, 1998. To appear in LNCS.
6. F. Morawietz and T. Cornell. On the recognizability of relations over a tree definable in a monadic second order tree description language. Technical Report SFB 340, Seminar für Sprachwissenschaft Eberhard-Karls-Universität Tübingen, 1997.