

Implementing WS1S via Finite Automata

James Glenn*
Univ. of Maryland

William Gasarch†
Univ. of Maryland

Abstract

A previous paper described the implementation of the decision procedure for the weak second-order theory of one successor (*WS1S*), especially those parts critical to its overall performance. This paper explores further some of the topics described in that paper, including minimization, representation of automata, subset construction, and other aspects of the decision procedure. New data will be presented to illustrate how various approaches to the above procedure affect the algorithm's performance.

1 Introduction

1.1 Definitions

1.1.1 *WS1S*

For any $m, n \in N$, let the following be called terms: n , x_n , and $x_n + m$. Define the language L_{S1S} as follows:

- (1) If t_1 and t_2 are terms and n is a natural number then $t_1 \leq t_2$ and $t_1 \in X_n$ are both in L_{S1S} .
- (2) If $n \in N$ and $f_1, f_2 \in L_{S1S}$ (such that no variable quantified in either f_1 or f_2 is free in the other) then the following are also in L_{S1S} : $(f_1 \wedge f_2)$, $(f_1 \vee f_2)$, $\neg f_1$, $(\exists x_n)f_1$, $(\exists X_n)f_1$, $(\forall x_n)f_1$, and $(\forall X_n)f_1$.

WS1S is the set of all sentences in L_{S1S} that are true under the standard interpretation (except quantifiers range over only *finite* sets or natural numbers). For example, of the following sentences, the first two are in *WS1S* but the second two are not:

$$\begin{aligned} & (\exists x_1)(\forall x_2)(\neg x_1 < x_2) \\ & (\forall X_1)(\exists x_2)(\forall x_3)(\neg x_2 < x_3 \vee \neg x_3 \in X_1) \\ & (\exists x_1)(\forall x_2)(x_2 < x_1) \\ & (\exists X_1)(\text{forall } x_2)(x_2 \in X_1) \end{aligned}$$

Büchi showed that *WS1S* is decidable through the use of finite automata. In a previous paper [1], we presented his proof again, making comments about those areas important to the performance of the implied decision procedure. In this paper we conduct a further exploration of some of those areas.

*Dept. of Computer Science, Univ. of MD., College Park, MD 20742, U.S.A. (email: glennj@cs.umd.edu).

†Dept. of C.S. and Inst. for Adv. Comp. Stud., Univ. of MD., College Park, MD 20742, U.S.A. Supported in part by NSF grants CCR-8803641 and CCR-9020079 (email: gasarch@cs.umd.edu).

1.1.2 $S2S$

To define the (strong) second order theory of two successors ($S2S$), recursively define the following as terms (for $n \in N$ and some term t): n , x_n , $tconcat0$, $tconcat1$. We define L_{S2S} recursively just as we did L_{S1S} , with the following addition:

(1.5) If t_1 and t_2 are both terms then $t_1prefixt_2$ is in L_{S2S} .

Now \leq refers to the standard lexicographical ordering of $\{0, 1\}^*$ and $t_1prefixt_2$ should be read “ t_1 is a prefix of t_2 .” $S2S$ is then the set of true sentences of L_{S2S} , and is decidable through the use of automata on infinite trees [4].

1.2 Motivation

Hilbert desired a procedure to decide all mathematics. In light of Gödel’s Incompleteness Theorem, that is of course impossible. Still, we can ask whether it is reasonable to expect computers to decide those areas of mathematics to which they can be applied. In particular there is a hierarchy (in terms of expressibility) of theories from $WS1S$ to $S2S$ which are decidable through the use of various forms of automata. $S2S$ could have been relevant to descriptive set theory: there is a theorem in that area that can be phrased in L_{S2S} . As it is, humans proved that theorem over a decade before Rabin showed that $S2S$ is decidable, but if the order of events had been different, we could have implemented the decision procedure (which no one has yet done), input the L_{S2S} sentence, and waited for the computer to tell us whether or not it was true. However, we most likely would be waiting for the answer still: even $WS1S$, the weakest theory in the hierarchy, has a provably terrible worst case running time. It has been our goal to see if the average case time for $WS1S$ is remotely as bad as the worst case. If it is, the certainly there is no hope for a practical procedure to decide $S2S$, and even a modified version of Hilbert’s program has practically no chance of success.

2 The Decision Procedure for $WS1S$

The decision procedure for $WS1S$ works by associating with each n -ary formula $f \in L_{S1S}$ a deterministic finite automaton M that operates on an n -track tape (we can also think of M as working on a singly tracked tape with alphabet $\{0, 1\}^n$, which will be called Σ_n . By use of these intermediary automata, we can build an M that works on a 0-track tape for any sentence f whose membership in $WS1S$ we wish to test. If M accepts any tape then $f \in WS1S$, otherwise $f \notin WS1S$. However, a formula f works on n -tuples of natural numbers and sets of natural numbers, while our machines work on n -track tapes. We therefore need a way of translating from one to the other and back.

3 Encoding

In the prior paper we described a way to go back and forth between tapes and tuples. The translation from n -tuples to n -track tapes is simple. First, we encode each component of the n -tuple on its own track. A natural number $n > 0$ will be encoded as $0^{n-1}1$; zero is encoded as the empty string. A finite set $S \subset N$ is encoded as the string $\chi_S(0)\chi_S(1)\dots\chi_S(\max\{x \in S\})$. If the tracks have different lengths, we pad the shorter tracks with zeros. To translate in the other direction, we assume that we know whether we want the result to be a natural number or a set. If we

want a number from a track, we find the location of the first 1 on the track and add one (regarding the first position on the tape as location zero). If there is no 1 then we interpret the track as zero. If we want the translation to result in a set, we read off the prefix of the characteristic function and assume the set is empty beyond the last entry recorded. The automata we build later depend on that encoding we use, and this particular encoding results in rather cumbersome automata.

We now present a more elegant solution that improves performance as well. A finite set $S \subset N$ we still encode in the same way as before (i.e. we record a sufficiently long prefix of its characteristic function). A natural number n we will now encode as 0^n1 . Decoding tracks as sets is the same as before, but decoding numbers is somewhat different. The number represented by a track is the location of the first 1 on the track (we do not add one to that location as we did before). If there is no 1 on the track, we pretend that we found one in the location just beyond the end of the tape (in effect, the number the track represents in this case is the same as the length of the tape). Now the empty tape and the tape containing the single symbol 1 both mean 0, 0 and 01 both mean one, 00 and 001 mean two, and so forth. This new encoding results in simpler automata. For example, there is a simple n -ary formula that results in an automaton with 2^n states using our original encoding. Using the new encoding we get an automaton with only one state. Most improvements are less extreme but still very substantial. Preliminary runs over a set of 1000 moderately sized, randomly generated formulas take almost two hours to decide using the old encoding but only about twenty minutes with the new encoding. We speculate that most of the improvement is because our new encoding leads to less nondeterminism.

3.1 Algorithm

To decide whether or not a sentence f from L_{S1S} is in $WS1S$, we first build automata corresponding to the atomic formulas in f . We then combine those automata in a manner prescribed by the structure of the subformulas of f . The automaton for $f_1 \wedge f_2$ is generated by using the standard procedure for automata intersection on the machines M_1 and M_2 that we constructed for f_1 and f_2 . Similarly, to build the automata for $f_1 \vee f_2$ or $\neg f_1$ we use the standard procedures for automata union or complementation. For $(\exists x_n)f_1$ we must introduce some nondeterminism into the automaton for f_1 ; the nondeterminism must be removed later (so the procedure for complementation will work). Once we have built the machine M corresponding to the sentence f , it is easy to check whether or not $L(M) = \emptyset$. If so, f is false and hence not in $WS1S$, otherwise f is true and is in $WS1S$.

4 Internal Representation of Automata

An n -track automaton has an alphabet with 2^n symbols. We store the complete transition function in an array for each state, which would normally require 2^n entries. Many states, however, ignore several tracks. For example, if a machine is driven to state q by an input that has a 1 on some track representing a number, then the transition function for q and all of q 's successors cannot depend on the symbol on that track. We take advantage of this to reduce the storage space for the transition function. In particular, if q ignores m tracks, then the transition function at q requires only 2^{n-m} units of storage.

This space saving measure makes it more time consuming to compute the transition function. Since the transition function is accessed over and over again, the performance of the decision procedure is degraded greatly. When the straightforward method is used, it is easy to find $\delta(q, \sigma)$ by simply treating σ as the binary representation of the index into the array holding the transition function. With the above space improvement, we must mask out certain tracks before converting

to a number. If we do this with elementary bitwise operations, we still end up with a number in the range 0 to $2^n - 1$, when what we want is a number in the range 0 to $2^{n-m} - 1$. For example if σ is 10110110 and we are interested in only the second, fourth, fifth, and eighth bits (with the most significant bit numbered one), we mask out the neglected bits to get `x0x10xx0`, which we read as simply 0100, which is the binary representation of 4, which we use as the index into our array. A slow way to do this follows:

```
result := 0
value := 1
for i := 31 to 0 do
  if bit i of the mask is set then
    if bit i of the symbol is set then
      result = result + value
    end if
    value := value * 2
  end if
end for
```

We do a little precomputation and cut the number of times through the loop to four:

```
result := 0
value := 1
for i := 0 to 3
  x := mask & 0xff           // strip off the last 8 bits in the mask
  y := symbol & 0xff        // do the same for the symbol
  add := table[x][y] * value // table[x][y] holds precomputed values for
                             // symbol y and mask x from the algorithm
                             // above
  result := result + add
  value := value * 2^bits[x] // bits[x] = number of bits set in x
  mask := mask / 2^8        // shift mask over to work on the next 8 bits
  symbol := symbol / 2^8    // do the same for symbol
end for
```

Preliminary data shows that the second algorithm allows us to cut the time for our 1000 inputs from 67 minutes to 37 minutes, an improvement of about 45 percent.

5 Quantifiers

Each quantifier in a formula f results in a nondeterministic automaton that is constructed by allowing the machine to “guess” that either a 0 or a 1 is on the track corresponding to the quantified variable. Formally, if $M_1 = (K, \Sigma_n, \delta, s, F)$ is the automaton for an n -ary formula f_1 and f_2 is the formula $(\exists x_n)f_1$ then the (nondeterministic) automaton for f_2 is

$$M_2 = (K, \Sigma_{n-1}, \Delta, s, F')$$

where the new transition relation is

$$\Delta = \{(q, \sigma, q') \mid \delta(q, (\sigma, 0)) = q' \vee \delta(q, (\sigma, 1)) = q'\},$$

and by $(\sigma, 0)$ we mean the n -tuple obtained from the $(n - 1)$ -tuple σ by inserting 0 as the last component, and similarly for $(\sigma, 1)$. Also, the new set of final states F' consists of those states of K that either are in F or can reach a state $f \in F$ using transitions only on a certain symbol that is prescribed by the types of the variables in f_1 .

The reason for modifying the set of final states is described in the previous paper; the change to the encoding described earlier necessitates only a very slight change in this process. We actually have two options as to how to perform the computation of the new final states. We can do the computation before determinization as implied above, or we can create a deterministic automaton without changing the final states of the original machine and then carry out the alteration on the resulting machine. Our preliminary data show that it is very slightly faster (the improvement is under ten percent) to modify the set of final states before performing the determinization. In the full paper we will provide more precise data, as well as a proof that either procedure is valid.

It was also mentioned in the previous paper that n consecutive quantifiers of the same type may be handled in one fell swoop rather than in n discrete steps. To do so, however, one must check 2^n values of the transition function of the old machine when computing one value of the transition function at a state in the new machine. We will present data to support the assertion that, on average, it is to our advantage to handle quantifiers in groups (at least for the formulas we have tested so far).

5.1 Reachable state set maintenance

Each nondeterministic machine that is created must be determinized. Profiling has confirmed the common sense feeling that the determinization subroutine is where the algorithm spends most of its time. We compute only those state sets that are reachable from the start state of the nondeterministic machine, hence one of our major concerns is what Wood and Johnson call reachable-state-set maintenance [3]. Within this problem lie at least two subproblems: that of determining if a state set has been seen before, and that of representing the state sets.

5.1.1 Finding a state set

We have implemented two solutions to the former subproblem. State sets can be stored in either a standard binary search tree, or in a tRIe (the latter is a binary tree structure in which the key values are kept only in leaves; the internal structure is such that with each internal node we associate a state of the original *NFA*, and all of the state sets stored below the left branch of the node will not contain that state, while those under the right branch will). The major problem with the binary search tree is that when we are searching for a particular set, determining which branch to take at a certain node can easily require testing membership of several states. With a tRIe structure, we need only test membership of one state to determine which branch to take. Our initial data show that the tRIe is better by twenty to seventy percent, depending on which structure is used to solve the second subproblem. (It should be noted that in practice we use a hash table to store the state sets and only use the above structures to handle collisions.)

5.1.2 Storing a state set

The state set representation subproblem is essentially that of how to store an m -element subset of $\{0, \dots, n - 1\}$. We have tried several approaches, summarized in the table below.

strcuture	insertion	membership	enumeration	space	time
bit vector	$O(1)$	$O(1)$	$O(n)$	n	9.8
linked list	$O(1)$	$O(m)$	$O(m)$	$64m$	15.2
sorted linked list	$O(m)$	$O(m)$	$O(m)$	$64m$	10.4
binary tree	$O(\log m)$	$O(\log m)$	$O(m)$	$160m$	9.6
hybrid	$O(\log n)$	$O(1)$	$O(m \log n)$	$2n$ to $4n$	11.3 to 11.7
red-black tree	$O(\log m)$	$O(\log m)$	$O(m)$	$193m$	9.9
sorted array	$O(\log m)$	$O(\log m)$	$O(m)$	$32m$	9.7

Here space is measured in bits and times are in minutes and represent running times on our set of 1000 test inputs. Time bounds listed for insertion, membership, and enumeration are generally for the theoretical average case (as opposed to averages we have observed in our study).

The entry labelled “hybrid” actually represents three very similar structures that are the result of an attempt to combine the best aspects of the bit vector structure with the those of the tree structures, and the time bounds listed are very rough. Essentially these structures consist of $\log n$ layers of bit vectors. The bottom layer contains n bits and records for each $i \in \{0, 1, \dots, n - 1\}$ whether i is in the set. The next layer up contains $\frac{n}{2}$ bits recording, for $i \in \{0, 2, \dots, n - 1\}$, if one of i or $i + 1$ is in the set. The layers proceed in this manner, with the top layer containing one bit that records if the set is empty or not. The hybrid structures can test set membership as quickly as bit vectors (by simply looking at the appropriate bit in the last level). A single insertion can now take $\log n$ time since we may have to set a bit in each level, but the total time for n insertions is still $O(n)$. The primary reason we constructed these structures was to improve on the slow enumeration of the bit vector structure. With the hybrid structure, we have the potential to look at many more bits at a time than with the bit vector (which examines at most eight bits at a time to search for the next set member). If we are looking for the next member of the set after x we will first look at bit $x + 1$ in the last level. If that is zero, we move up a level and check a bit, which has the effect of testing two bits in the level below. If we find another zero, we move up a level and check four bits with the next test, and so on. If our sets are large and sparse then this will be more efficient than the bit vector, despite the very low constant factors associated with the bit vector’s enumeration procedure. However, for large sparse sets the tree structures will beat the hybrids and the bit vector. We are looking for some middle ground or perhaps some other applications to which we can apply our new structure.

Red-black trees are an “approximately balanced” binary tree structure. We decided to implement them because we felt the performance of the standard binary tree structure may have been degraded by the order in which the insertions are performed. We suspected that the insertions frequently were done almost in order, which would result in long, skinny trees and near worst-case (linear) running times. Apparently the overhead that comes with balanced trees negated any gains they could have made.

The sorted array structure takes advantage of the fact that all the insertions are done before either of the other two functions are utilized. In our implementation we first insert elements in constant time into an unsorted array (using a bit vector to avoid inserting duplicates). Once the last insertion is made we can discard the bit vector and sort the array in $O(m \log m)$ time, for an amortized time of $O(\log m)$ time for each of the m insertions. Set membership can then be tested by binary search, and enumeration is simple.

Our test data show the four best performing structures in a statistical tie. Further analysis of our current data, along with test runs on larger machines, may eventually show that some of these structures are clearly better for certain size machines.

6 Minimization

Without minimization, our automata grow too large to handle very quickly. Naturally we have investigated which minimization algorithm works fastest on our machines. Brzowski's algorithm for minimization (reverse, determinize, reverse, determinize) has been championed by some for its excellent performance on small automata despite its worst-case exponential running time [5]. The machines we construct are for the most part not small, but we still wanted to see how Brzowski's algorithm would perform. The exponential nature of the algorithm showed up even for small automata. For example, the automaton we construct for the formula

$$x_2 + 4 \text{ in } X_0 \vee (2 < x_2 \wedge x_2 + 4 \notin X_1) \vee (x_2 + 2 \in X_1 \wedge X_1 \subset X_0)$$

has 26 states and takes 2.35 seconds to minimize via reversal to an equivalent 21 state machine. Hopcroft's algorithm minimizes it in less than a hundredth of a second. The great disparity in times has also been seen when feeding our automata into `grail`, one of several a general purpose toolkits for finite automata.

We have one other small update in this area since the previous paper: we had been using Hopcroft's algorithm [2] exclusively for non-layered automata (a layered automaton is one in which the only cycles are self loops; we have a specialized algorithm for minimizing them). The improvement in computing the transition function noted above means that the standard algorithm is now faster for automata up to a certain size. We must endeavor to find exactly what that size is. It should be noted that our specialized algorithm for minimizing layered automata still outperforms both the standard and Hopcroft's algorithms on layered machines of any size.

7 Data

Our previous paper listed the results of running the decision procedure on several sets of randomly generated formulas. In each set, the formulas all has the same number of variables, clauses, and the same maximum constant. We reproduce those results below (times are in milliseconds).

quantifiers clauses	2		3		4	
	8	16	8	16	8	16
<i>max</i> = 2	33	57	45	86	91	159
<i>max</i> = 3	40	70	59	111	133	226
<i>max</i> = 4	46	85	79	143	210	335
<i>max</i> = 5	56	100	99	183	289	517
<i>max</i> = 6	67	122	122	236	418	728
<i>max</i> = 7	78	140	175	297	632	1037
<i>max</i> = 8	99	177	294	547	1213	1640

Now, after a year of improvements, we have run the decision procedure on the same inputs and recorded the results below. Note the dramatic savings in time, especially for more complex

formulas.

quantifiers clauses	2		3		4	
	8	16	8	16	8	16
$max = 2$ 21	36	30	53	43	88	
$max = 3$ 24	43	36	66	60	124	
$max = 4$ 28	49	48	80	86	186	
$max = 5$ 31	56	57	101	107	204	
$max = 6$ 37	66	67	123	143	348	
$max = 7$ 41	73	89	148	189	485	
$max = 8$ 50	88	127	176	230	624	

References

- [1] J. Glenn and W. Gasarch. Implementing *Ws1s* via finite automata. In *Proceedings of the First International Workshop on Implementing Automata*, pages 87–98, August 1996.
- [2] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *Theory of Machines and Computation*, pages 189–196. Academic Press, 1976.
- [3] J. Johnson and D. Wood. Instruction computation in subset construction. In *Proceedings of the First International Workshop on Implementing Automata*, pages 1–9, August 1996.
- [4] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. AMS*, 141:1–35, July 1969.
- [5] B. W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, 1995.