Mona & Fido: The Logic-Automaton Connection in Practice

Nils Klarlund

AT&T Labs–Research
Florham Park, NJ
klarlund@research.att.com

Abstract. We discuss in this paper how connections, discovered almost forty years ago, between logics and automata can be used in practice. For such logics expressing regular sets, we have developed tools that allow efficient symbolic reasoning not attainable by theorem proving or symbolic model checking.
We explain how the logic-automaton connection is already exploited in a limited way for the case of Quantified Boolean Logic, where Binary Decision Diagrams act as automata. Next, we indicate how BDD data structures and algorithms can be extended to yield a practical decision procedure for a more general logic, namely WS1S, the Weak Second-order theory of One Successor. Finally, we mention applications of the automaton-logic connection to software engineering and program verification.

1 Introduction

It was discovered almost forty years ago that automata make a handy mathematical tool for the understanding of certain weak logics of arithmetic. Much later, automata have been used extensively for decision-theoretic problems in temporal logics. But in practice, the success of verification methods, especially as seen in the area of hardware, relies on calculations that are based on BDDs (Binary Decision Diagrams) [5], not general automata. For example, the model-theoretic approaches allow a satisfiability relation for a transition system, represented by a BDD, and a temporal logic property to be calculated efficiently [23]. In fact, BDDs allow sometimes spectacular compression of symbolic information and shortening of computational work. Therefore, this approach to model-checking is sometimes called *symbolic*.

BDDs are a special kind of deterministic automata that accept finite sets. A BDD can be reduced to a canonical or minimum representation in linear time. This property is crucial to symbolic reasoning, since even slight deviations from minimum representations in a serious of computations lead to exponential growth.

Automata in general accept a much larger class of languages than BDDs, namely the regular sets. So, the succes of symbolic model-checking, now routinely used in industry for hardware verification, does not answer the question:

$\boxed{\text{What can we do in a world where automata, not BDDs, are the principal means of representation?}}$

The aim of the MONA project at BRICS, University of Aarhus, is to shed light on this question by exploiting the original logic-automaton connection. The resulting computational framework is not a competitor to usual techniques, however, but promises to offer new and different applications of the symbolic approach.

In this paper, we give a tutorial introduction to the logic-automaton connection and to its use in practice, as carried out in the MONA project.

In Section 2, we first discuss BDDs and logic-automaton connection for Quantified Boolean Logic. Next, we introduce a more general logic, WS1S, which corresponds to the class of regular languages—while subsuming a fragment of arithmetic. We show how automata themselves can be represented using a kind of BDD. We also mention some related work, where similar automata were introduced.

In Section 3, we outline how we have put together the MONA decision procedure for BDD-represented automata. We also mention FIDO, a high-level language, which integrates many usual programming language concepts with WS1S.

In Section 4, we explain applications of MONA to program verification and software engineering.

2 The logic-automaton connection

The logic-automaton connection can be easily explained, along with the notion of BDDs, if we look at Boolean Logic. Consider a Boolean function $\phi(x_1, x_2, x_3) \equiv x_1 \vee (x_2 \Leftrightarrow x_3)$, where x_1, x_2, x_3 are propositional variables. The *BDD representation* of ϕ is an acyclic, directed graph whose nodes are labeled with variable names. The representation is contingent on a choice of variable ordering. If we choose the ordering to be x_1, x_2, x_3, then the BDD is the graph shown in Figure 1. This graph describes the function ϕ according to the following recipe. To find the value of ϕ for a given truth assignment, start in the root. For each internal node, shown as a circle and labeled with a variable x, go to the successor node along the edge marked 0 or 1 according to the value of x as given by the truth assignment. When a leaf, shown as a square, has been reached, its value (0 or 1) is the value of the function.

The BDD is a canonical structure that can be derived from a natural minimum, deterministic automaton. This automaton is defined by its language, which—for the example ϕ—we take to consist of all satisfying truth assignments regarded as strings of length 3, see Figure 2. Compared to the BDD, this graph contains two extra nodes, shown in the shaded area and labeled x_2 and x_3. These nodes are essentially equivalent to the leaf below them in the sense that they do not contribute any information as to the truth value of the function. The canonical BDD is defined from the canonical automaton by removing all such nodes. More precisely, any node whose two successors are identical is removed,

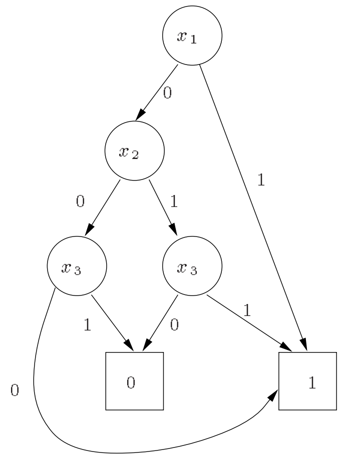


Figure 1. A simple BDD

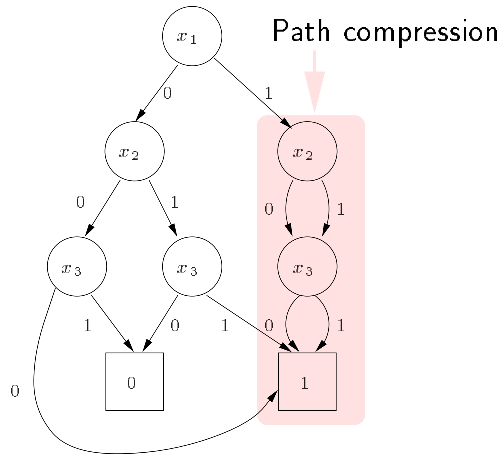


Figure 2. A simple automaton

while all its incoming edges are directed to the common successor. The technique is known as *path compression*. It can be shown that the resulting graph is independent of the order in which the nodes are removed. Thus, the BDD is itself canonical, and the operation can be summarized:

$$\boxed{\begin{array}{ll}\text{BDD} = & \text{canonical automaton} + \\ & \text{path compression}\end{array}}$$

BDD properties

Since BDDs are directed graphs, they can be stored efficiently in a computer. Each internal node is hashed to a canonical address according to the name of its variable and the addresses of its two successors.

It can be shown, see[5], that there are $O(n \cdot m)$ algorithms constructing the BDDs for Boolean operations like $\phi \wedge \psi$ or $\phi \vee \psi$, where ϕ's and ψ's BDDs have size n and m, respectively. These algorithms are formulated as a simultaneous recursive descent in the BDDs for ϕ and ψ that directly (without an additional minimization phase) constructs the canonical BDD for the resulting Boolean function. Also, existential quantification $\exists x : \phi$ can be carried out in time $O(n)$, and negation $\neg\phi$ can be done in $O(1)$ time.

Quantified Boolean Logic

The algorithms above form an exponential time decision procedure for Quantified Boolean Logic. This application of the logic-automaton connection has already found widespread use in hardware verification. Path compression turns out to be essential to the success of BDDs, although the theoretical savings over the canonical automaton representation can be shown to be only logarithmic [22].

Weak Second-order Theory of One Successor

The logic-automaton connection can be generalized to automata that accept general, infinite regular languages. The logics corresponding to regular languages are much more expressive than QBL. They contain fragments of arithmetic and allow quantification over finite sets of numbers. There are a couple of ways of formulating such a logic. Here, we will look at a very natural formalism, whose not-so-natural name is the **W**eak **S**econd-order theory of **1** **S**uccessor, or WS1S. Its syntax consists of:

- First-order terms t:
  - the constant 0, variables p; and
  - successor terms $t' + 1$, where t' is a first-order term.
- Second-order terms T:
  - the constant $\emptyset$, variables P; and
  - set terms $T' \cup T'$, $T' \cap T''$, $T' \backslash T''$, where T' and T'' are set terms.
- Formulas ϕ:
  - term comparisons $t = t'$, $t \leq t'$, $T = T'$, $t \in T$;
  - propositional combinations $\phi' \wedge \phi''$, $\phi' \vee \phi''$, $\phi' \Rightarrow \phi''$, and $\phi' \Leftrightarrow \phi''$, where ϕ' and ϕ'' are formulas;
  - first-order quantification $\exists p : \phi'$, $\forall p : \phi'$; and
  - second-order quantification $\exists P : \phi'$, $\forall P : \phi'$.

Note that there is no set-theoretic complement operation. The semantics is very simple:

- First-order terms are interpreted as natural numbers, and
- second-order terms are interpreted as *finite* sets of natural numbers.

Example: even numbers We can express that the second order-variable P denotes a finite set of all even numbers less than some unbounded constant:

$$
\begin{aligned}
& P = \emptyset \\
& \vee (0 \in P \;\wedge\; (\exists p' \in P : \\
& \qquad\qquad \forall q \in P : q \leq p' \\
& \qquad\qquad \wedge \forall q \leq p' : (q \in P \Leftrightarrow q + 1 \notin P)))
\end{aligned}
$$

Example: representing arrays Using WS1S, we can represent data structures that cannot be modeled in Boolean Logic. For example, let us consider a variable length array containing numbers in $\{0, \ldots, 3\}$. Since its length is unbounded, such an array cannot be represented by any finite set. But in WS1S, we can use two second-order variables Q_1 and Q_2 to encode values of the array and a first-order variable q to denote its length. For example, we could make a convention that position p contains

- 0 iff $p < q \;\wedge\; (p \notin Q_1 \wedge p \notin Q_2)$;
- $\ldots$
- 3 iff $p < q \;\wedge\; (p \in Q_1 \wedge p \in Q_2)$.

In this way, an interpretation of the variables q, Q_1, and Q_2 denotes an array. For example, $q = 5$, $Q_1 = \{1, 3\}$, $Q_2 = \{2, 3\}$ denotes the array $\langle 0, 1, 2, 3, 0 \rangle$.

Example: Presburger Arithmetic Since natural numbers can be viewed as finite bit-strings, that is, as finite sets, we can interpret first-order arithmetic without multiplication, but with addition, in WS1S. This theory is called Presburger Arithmetic.

Interpretations viewed as finite strings

The logic-automaton connection follows rather naturally once we understand how interpretations can be regarded as finite strings. As an example, consider WS1S formula $\phi \equiv P = Q$ and an interpretation $P = \{1, 3\}$, $Q = \{2, 3\}$. This interpretation could be represented as a string over the alphabet $\mathbb{B}^2$:

P	0	1	0	1	0	0
Q	0	0	1	1	0	0
	0	1	2	3	4	5

If this string is called w, then we say that value of P is described along the first track of w and that Q is described along the second track. Note that $w \not\models \phi$. And, note that many w' denote same interpretation as w, since an arbitrary number of the letter $\binom{0}{0}$ may be appended to w without changing the interpretation that it denotes.

The above example does not explain how first-order variables are to be handled in the setting of strings. There is a straightforward solution: rewrite the formula while treating first-order variables as second-order variables whose values are restricted to be singletons.

Now, a formula ϕ naturally defines a language $L(\phi) = \{w \mid w \vDash \phi\}$ over the alphabet $\mathbb{B}^k$, where k is the number of variables described (so k is at least the number of free variables in ϕ). It is not hard to prove by induction that all $L(\phi)$ are represented by DFAs (deterministic, finite-state automata):

- Simple formulas like $p = q + 1$ can be mapped to small automata.
- Negation is handled by automaton complementation, since $L(\neg\phi) = \complement L(\phi)$.
- Conjunction can be handled by an automaton product, since $L(\phi \wedge \psi) = L(\phi) \cap L(\psi)$.
- Existential quantification is a little harder, but it turns out that $L(\exists P : \phi) = E^i(L(\phi)/L^i)$, where
  - the variable P is described along the ith track;
  - $E^i(L)$ is the projection of L that removes the ith track from L, that is,

    $$E^i(L) = \{w \mid \exists w' \in L : w \text{ is obtained from } w' \text{ by deleting the } i\text{th track}\};$$

  - L/L' denotes the right-quotient of L by L'; and
  - $L^i = \{u \in \mathbb{B}^k \mid j\text{th track is all 0 for } j \neq i\}$.

  Projection results in a non-deterministic automaton, which can be converted back into a DFA by a subset construction. The DFA for the right-quotient is calculated by a backwards traversal from the final states.

From these observations, it follows that

$$\boxed{\begin{array}{l} L(\phi) \text{ is regular, and an automaton } A \text{ recognizing} \\ L(\phi) \text{ is computable from } \phi. \end{array}}$$

Büchi (1960), Elgot (1961) [6, 8]

Unfortunately, "computable" is bounded from below by an unbounded stack of exponentials. In fact, for any translation $\phi \longrightarrow A$ with $L(\phi) = L(A)$, the following holds:

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

Meyer (1972) [24, 30]

where n is the length of the formula and c is some constant.

This bound is discouraging! But of course, it does not by itself preclude that the logic may be useful. Logics with worse complexity—such as Turing-complete formalisms—have been used as programming languages and in theorem proving. It is worrisome, however, that the alphabet size for automata that represent formulas with k free variables necessarily is 2^k. This exponential explosion can be countered by thinking of automata as representing languages over $\mathbb{B}$, where letters in a string u over $\mathbb{B}^k$ are laid out consecutively to form a string over $\mathbb{B}$

that is k times longer than the length of u. Such a view, however, does not give us the advantages of path compression in BDDs.

Let us consider using BDDs to represent automata. Automata are finite objects, and as such are encodable. For example, a single BDD could encode the set

$$\{(r, a, s) \mid r \xrightarrow{a} s\},$$

where r and s are states and $a \in \mathbb{B}^k$ is a letter. Such a representation requires us to choose an encoding of states. Indeed, this is the traditional approach in symbolic model checking, where a state space of size N is encoded with a number M of Boolean variables logarithmic in N. Usually, each program variable is directly encoded using a small number of BDD variables. Since the transition from one state to another usually modifies only a few components of the state space description, an interleaving of the BDD nodes describing the old and new states often results in BDDs that are only linear or polynomial in size of M [23]. So, if the state space of automata in our decision procedure could be encoded similarly, a formula with k logical variables could be described by a single BDD with $2 \cdot M + k$ variables.

Unfortunately, the representation just suggested is fundamentally unsuited, it seems, for carrying out essential operations on automata such as minimization. In fact, the representation itself is far from canonical, since there are no rules prescribing the encoding of states. Moreover, it is difficult to imagine how a minimization procedure could be formulated that would directly construct a canonical encoding of the minimum state space even if such an encoding was devised.

In addition, it is also hard to imagine how a subset construction could be efficiently formulated based on the single-BDD representation.

Thus, the representation of a transition system by a single BDD is not suitable for automaton-centered calculations. By a *BDD-represented automaton*, we must think of a data structure that efficiently supports all the operations characteristic of automata.

BDD-represented automata

A rather simple generalization of BDDs results in a suitable representation: a canonical graph (up to naming of states and up to variable ordering) arises if *shared, multi-terminal* BDDs are used. These BDDs have multiple leaves, one corresponding to each state. Each state, in turn, is associated with such a BDD, and *sharing* means that isomorphic subtrees are identified. For example, the property "there are more than one element in the intersection of X and Y," or

$$\exists p, q : p \neq q \ \wedge \ p \in X \cap Y \ \wedge \ q \in X \cap Y$$

can be represented by the automaton shown in Figure 3; here each state r, s, or t is described in an array with information about whether it is final and with a pointer to a BDD node defining its transition function. This automaton accepts the regular language $(\Sigma - \binom{1}{1})^* \cdot \binom{1}{1} \cdot (\Sigma - \binom{1}{1})^* \cdot \binom{1}{1} \cdot \Sigma^*$, where $\Sigma = \mathbb{B}^2$.

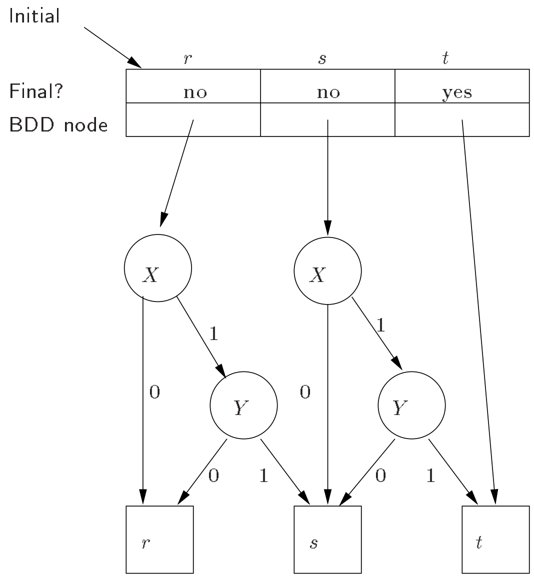


Figure 3. BDD-represented automaton

This works for trees, too

WS1S can easily be generalized to express regular sets of labeled, finite trees. The resulting logic, WS2S, **W**eak **S**econd-order theory of **2** **S**uccessors, deals with elements and finite subsets of the infinite, binary tree:

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

The logic-automaton connection generalizes to WS2S [7, 31], where the appropriate automaton concept is a *deterministic, bottom-up tree automaton.* Its transition function determines for each pair (r, s) of states and each letter a what the next state is. Tree automata are for that reason at least quadratically more difficult to work with in practice than DFAs. We have formulated data structures and algorithms for such automata in [3].

But S2S seems intractable in practice

The restriction to quantification over just finite subsets of the infinite binary tree can be removed while keeping decidability of the formalism, which is then simply named S2S. This is Rabin's result [28], which in essence boils down to a complicated question about the complementation of automata on infinite trees; see [32]. It seems unlikely that S2S can be dealt with in practice; no representation with canonicity properties is even known for S2S. For S1S, the situation is slightly brighter thanks to the existence of syntactic congruences, such as the ones discussed in [1, 17, 26, 29, 34].

Monadic second-order logic (M2L) on finite strings

The regular sets can also be described by a logic for which there is a one-to-one relationship between strings over $\mathbb{B}$ and interpretations (not a many-to-one relationship as with WS1S). Under this view, a string determines a number n, the length of the string, and the successor operation $+1$ is interpreted modulo n. In addition, second-order variables are restricted to range only over subsets of $\{0, \ldots, n-1\}$. This logic can be called the *Monadic Second-order Logic on Finite Strings* or *M2L(Str)*. (Apparently, it is not as strong as WS1S in the sense that there is no obvious way of encoding Presburger Arithmetic in it.) The earlier versions of our decision procedure were written for this logic.

Related work

Fisher and Gupta [12] were apparently the first to use BDD-represented automata as a mechanism for representing regular languages over large alphabets. Their automaton concept is slightly different from ours since it was introduced as a representation for *linear inductive functions*, a notation for regular languages. In [11], their approach is generalized to tree languages and to problems beyond the regular sets.

Automata on large alphabets are also implicit in work on the relationship between *p-adic numbers* and circuits [33].

Work at the University of Kiel [27] led to an implementation of a decision procedure for M2L on finite trees, but it did not address the problem of large alphabets. Neither did the recent decision procedure implementations for WS1S [9] and WS2S [25]. Techniques similar to ours have been used in an M2L(Str) tool [15] and in an application to languages for describing parameterized systems [16]. Also, the decision procedure for Presburger arithmetic reported in [4] is based on automata over large alphabets.

3 Mona & Fido Tools

In this section, we discuss the automata-based tools that are under development at the University of Aarhus. We have at several levels tried to optimize performance as much as possible, and as a result our Mona tool, which implements decision procedures for WS1S and WS2S, has become orders of magnitude faster since we started [13].

Data structures and algorithms in practice

Usually, BDDs are stored in a single global address space indexed by hash values as previously noted. However, it appears unlikely in the case of BDD-represented automata that there would be much sharing of nodes across different automata. This is because the state numbering is arbitrary, so isomorphic sub-automata are unlikely to be identified by the hashing. Thus, we have chosen to use a separate, contiguous address space for each automaton. Consequently, BDD nodes of an automaton can be garbage collected in constant time as soon as the automaton is not needed any longer. Also, we have chosen to represent BDDs nodes directly in the hash table, instead as separately allocated records pointed to buy entries in the hash table. This scheme creates substantially fewer hardware cache misses than traditional implementations. As a result our BDD package runs approximately six times faster than a widely used package [20].

Algorithms for DFAs A product algorithm is easy to describe and implement. Also, an $O(n^2)$ minimization algorithm is easy to implement, although its quadratic running time becomes a problem with bigger automata, where we often see that minimization may take ten times longer than the time to construct the automaton. (An $O(n \cdot \log n)$ algorithm exists [18], but has not been implemented.) The subset construction is considerably more difficult to implement efficiently. Curiously, it turns out that the subset construction is usually very well-behaved in practice, often resulting in a subset automaton with fewer states than the original automaton. Some theoretical arguments why this is the case can be found in [2].

Algorithms for tree automata To address the inherent quadratic nature of the transition table representation of tree automata, we have developed a default representation that sometimes can cut the representation from quadratic to linear. Also, we have devised a notion of partitioned tree automaton, which in principle may yield exponential savings [3].

FIDO: a high-level version of M2L

From a programmer's point of view, WS1S is a kind of assembly language about strings of bits. Therefore, we have developed a a logical formalism, FIDO [21], which combines WS1S and WS2S with programming language concepts such as records, enumerated types, recursive data types, simple polymorphims, and unification.

A recursive data type is a convenient way to express a class of labeled trees. For example, binary trees, each of whose nodes is red or black and contains a value in $\{1, \ldots, 10\}$, can be expressed as a type `Tree`:

```
type Tree = red,black(val:  Range,
                    left,right:  Tree) |
          leaf;
type Range = [1..10];
```

Two trees, `x` and `y`, of type `Tree` are declared as

```
tree x,y :  Tree;
```

Then a property such as "there are nodes with different colors somewhere in trees `x` and `y`" can be expressed as a formula:

```
∃ pos p, q:  x, y.
    p ≠ q ∧ read(p) ≠ read(q)
```

Here, `read` is a function that designates the color of a node, and `p, q:  x, y` denotes that `p` and `q` range over positions in trees `x` and `y`.

The FIDO compiler translates FIDO programs into the MONA syntax for WS1S or WS2S. The resulting formulas are often very big, sometimes on the order of 10^5 characters. Therefore, we have implemented symbolic simplification of WS1S formulas in the front-end of MONA. These reduction techniques yield substantial simplifications of the code before it is handed to the back-end, which executes the automata-theoretic calculations.

In summary, FIDO is a domain-specific language for the expression of regular sets of strings and trees. It is a programming lanuage based on logic, but not a traditional "logic programming" language, since in contrast to resolution-based languages, FIDO

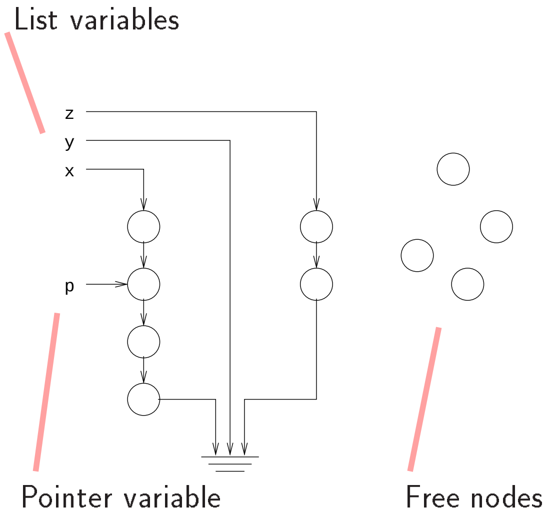
– is a quantificational logic not restrained to Horn clauses;
– but it is not a Turing-complete language.

4 Applications

We mention here two applications, where other symbolic methods are unlikely to be as efficient as the use of automata-based reasoning.

Decidable program logic for pointers

In our first application, taken from [14], we indicate how FIDO can be used as an assertional formalism about pointers and linked lists in programming languages. To us, a typical store looks like:



Here, `x`,`y`, and `z` are *list variables* that denote linked lists, and `p` is a *pointer variable*, which points to some node in the store. There are also free nodes available. So, stores consist of a fixed number of linked lists, each of arbitrary size; a fixed number of variables denoting nodes in the store; and a number of free nodes. The set of all such stores can be modeled by an infinite collection of finite sets. For example, we can in one FIDO formula describe all possible stores, where `x`,`y`, and `z` are linked lists and where `p` points to a node in a list.

Moreover, it can be shown that the precise semantics of basic blocks (loop-free code) can be encoded as predicate transformers on FIDO formulas. (This is possible only since we have restricted all values in records to be finite domain.)

FIDO extended with regular expressions also serves as a means of formulating program assertions. From putting these observations together, we may obtain a decidable fragment of Floyd-Hoare program logic.

An example in Pascal

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

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

List = ^Item;

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

We would like to verify that the program below calculates a well-formed list `z`. The program merges the lists `x` and `y`, but it is certainly far from evident that the pointer manipulations expressed really do result in a list!

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

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

To verify the program, we need to state a property about the loop. Let us use the property

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

So, we annotate the `while`-loop with the the property transcribed into a logical assertion:

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

The decision procedure verifies that this assertion is preserved under one iteration of the loop. Had we made a mistake so that the assertion was not an invariant, we would automatically have gotten a counter-example.

Design constraints for CORBA

In our second application, taken from [19], we propose to use FIDO as a formalism for restricting parse trees. Such restrictions are useful when it is known that source code must satisfy specific requirements imposed by the platform or software environment. For example, a system architect may have formalized the constraints of the CORBA interface [10]. One such constraint is that operation arguments of CORBA interfaces cannot denote object values; more precisely,

> *If* x *is an operation argument node in the syntax tree of a* CORBA *object type, then the node* `y` *below denoting its type cannot represent an object type.*

Formally, such a statement could be written as:

CONSTRAINT corba **FOR**
  ObjectTypeSpecification **IS**
  $\forall$ x: Argument. **root** $\lhd$ x $\Rightarrow$ $\exists$ y: Type.
      x $\lhd$ y $\wedge$ $\neg$ OT(y);
END

Here, ObjectTypeSpecification, Argument, and Type are production names of the syntax, and OT(y) is a predicate that evaluates to true if y is a node that denotes an object type. The expression **root** denotes the node of type ObjectTypeSpecification to which the category applies, and $x \triangleleft y$ holds when node x is the parent of node y.

The programmer can annotate object type specifications in the source code with the corba constraint. In a conformance checking phase, tree automata corresponding to the constraints are run on the syntax tree, and constraints that are not valid are flagged.

The advantage of FIDO, compared to other formalisms for specifying constraints on trees, is that there is no requirement to precisely formulate how information flows up and down the tree in terms of attributes or recursive procedures. Instead, positions can be referred to and compared as first-order variables.

Getting MONA

The first official version of MONA was released in the Fall of 1997. It can be obtained from `http://www.brics.dk/~mona`, where also additional references and papers are available.

Acknowledgments

Thanks to the referees for many helpful comments.

References

1. A. Arnold. A syntactic congruence for rational ω-languages. *Theoretical Computer Science*, 39:333–335, 1985.
2. D. Basin and N. Klarlund. Beyond the finite in hardware verification. Submitted for publication. Extended version of: "Hardware verification using monadic second-order logic," *CAV '95*, LNCS 939, 1996.
3. M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, WIA '96, Lecture Notes in Computer Science, 1260*. Springer Verlag, 1996.
4. A. Boudet and H. Comon. Diophantine equations, presburger arithmetic and finite automata. In *Trees and algebra in programming - CAAP*, volume 1059 of *LNCS*, 1995.
5. R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing surveys*, 24(3):293–318, September 1992.
6. J.R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.
7. J. Doner. Tree acceptors and some of their applications. *J. Comput. System Sci.*, 4:406–451, 1970.
8. C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–52, 1961.
9. J. Glenn and W. Gasarch. Implementing WS1S via finite automata. In *Automata Implementation, WIA '96, Proceedings*, volume 1260 of *LNCS*, 1997.

10. Object Management Group. The Common Object Request Broker: architecture and specification, 1995 July. revision 2.0.
11. A. Gupta and A.L. Fisher. Parametric circuit representation using inductive boolean functions. In *Computer Aided Verification, CAV '93, LNCS 697*, pages 15–28, 1993.
12. A. Gupta and A.L. Fisher. Representation and symbolic manipulation of linearly inductive boolean functions. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 192–199. IEEE Computer Society Press, 1993.
13. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1996.
14. 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.
15. 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.
16. Y. Kesten, O. Maler, M. Marcus, and E. Shahar A. Pnueli. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Computer Aided Verification, CAV '97*, volume 1254 of *LNCS*, 1997.
17. N. Klarlund. A homomorphism concept for ω-regularity. In *Proc. of Computer Science Logic*, 1994. To appear.
18. N. Klarlund. An $n \log n$ algorithm for online BDD refinement. In O. Grumberg, editor, *Computer Aided Verification, CAV '97*, volume 1254 of *LNCS*, 1997.
19. N. Klarlund, J. Koistinen, and M. Schwartzbach. Formal design constraints. In *Proc. OOPSLA '96*, 1996.
20. N Klarlund and T. Rauhe. Bdd algorithms and cache misses. Technical report, BRICS Report Series RS-96-5, Department of Computer Science, University of Aarhus, 1996.
21. N. Klarlund and M. Schwartzbach. A domain-specific language for regular sets of strings and trees. In *Proc. Conference on Domain Specific Languages.* Usenix, ACM SIGPLAN, 1997.
22. H-T. Liaw and C-S. Lin. On the OBDD-representation of general Boolean functions. *IEEE Trans. on Computers*, C-41(6):661–664, 1992.
23. Ken McMillan. *Symbolic Model Checking.* Kluwer, 1993.
24. 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 *LNCS*, pages 132–154, 1975.
25. 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.
26. L. Staiger O. Maler. On syntactic congruences for omega-languages. *Theoretical Computer Science*, 183:93–112, 1997.
27. A. Potthoff. Project to implement monadic-second order logic on finite trees. Unpublished., 1994.
28. M.O. Rabin. Decidability of second-order theories and automata on infinite trees. *American Mathematical Society*, 141:1–35, 1969.
29. B. Le Sac. Saturating right congruences. *Informatique Théorique et Applications*, 24:545–560, 1990.

30. L. Stockmeyer. *The complexity of decision problems in automata theory and logic.* PhD thesis, Dept. of Electrical Eng., M.I.T., Cambridge, MA, 1974. Report TR-133.
31. 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.
32. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.
33. Jean E. Vuillemin. On circuits and numbers. *IEEE Transactions on Computers*, 43(8):868–879, 1994.
34. T. Wilke. An algebraic theory for regular languages of finite and infinite words. *Int. J. of Algebra and Computation*, 4:447–489, 1993.