BRICS NS-98-3 Klarlund & Møller: MONA Version 1.2 — User Manual

MONA Version 1.2 User Manual

Nils Klarlund
Anders Møller

BRICS Notes Series NS-98-3

ISSN 0909-3206 June 1998

MONA Version 1.2
User Manual

Nils Klarlund, klarlund@research.att.com
Anders Møller, amoeller@brics.dk

June 1998

Contents

1 Introduction

It has been known since 1960 that the class of *regular languages*[1] is linked to decidability questions in formal logics. In particular, *WS1S* (*Weak Second-order Logic of One Successor*) is decidable[2] through the *automaton-logic connection*, which can be simply stated: the set of satisfying interpretations of a subformula is represented by a finite automaton [9, 12]. WS1S thus acts as a notation for regular languages, just as regular expressions do.

The automaton for a formula can be calculated by a simple induction scheme, where logical connectives correspond to classic automata-theoretic operations such as product and subset constructions. Validity and unsatisfiability of the formula can be determined and satisfying examples and counter-examples can be constructed by analyzing the associated automaton.

Despite its name, WS1S is a simple and natural notation. A variation of first-order logic, WS1S is a formalism with quantifiers and Boolean connectives. Its interpretation, however, is tied to arithmetic—somewhat weakened to keep the formalism decidable. In WS1S, first-order variables denote natural numbers, which can be compared and subjected to addition with constants. WS1S also allows second-order variables while remaining decidable; each such variable is interpreted as a finite set of numbers.

WS1S and its generalization WS2S [35, 11], which is interpreted over the infinite binary tree, can be used to express problems ranging from hardware verification to formal linguistics. Unfortunately, the space and time requirements for translating formulas to automata have been shown [29] to be non-elementary (i.e., bounded from below by a stack of exponentials whose height is proportional to the length of the formula). Thus, the decidability property has for many years been considered intractable for practical use.

The MONA tool is an efficient implementation of the decision procedures for WS1S and WS2S. "Efficient" here means that the tool is fast enough to have been used in a variety of non-trivial settings. MONA translates WS1S and WS2S formulas into minimum DFAs (Deterministic Finite Automata) and GTAs (Guided Tree Automata [5]), respectively. The automata are represented by shared, multi-terminal BDDs (Binary Decision Diagrams), see [7, 15].

1.1 Introductory example

We introduce MONA by a tiny example showing its basic functionality. MONA is run on a file that defines a WS1S or WS2S formula. The result of the compilation is an automaton which can be used in any way the MONA programmer might desire. Often, however, the interest in the compilation is the analysis that MONA can carry out on the calculated automaton.

A MONA *program* consists of a number of declarations and formulas. The two-line WS1S program below, contained in a file `simple.mona`, declares two free second-order variables, `P` and `Q`, and postulates that the set difference of `P` and `Q` is the union of the sets `{0,4}` and `{1,2}`:

```
var2 P,Q;
P\Q = {0,4} union {1,2};
```

[1] A *regular* language is a set of finite strings recognized by a finite-state automaton. We assume that the reader is familiar with automata theory as taught in undergraduate computer science courses.

[2] A logic is *decidable* if an algorithm exists that determines for any formula its truth status: valid (formula is always true) or not valid (sometimes formula is false); alternatively—and equivalently—the algorithm can classify formulas according to whether they are satisfiable (sometimes true) or unsatisfiable (always false).

Clearly, this formula is not always true, but there is an interpretation that makes it hold, for example the one which assigns $\{0, 1, 2, 4\}$ to `P` and $\{5\}$ to `Q`. This interpretation, usually written as $[\mathtt{P} \mapsto \{0, 1, 2, 4\}, \mathtt{Q} \mapsto \{5\}]$, can also be represented by 0s and 1s in a string

$$\begin{array}{ccccccc} \begin{matrix}\mathtt{P}\\ \mathtt{Q}\end{matrix} & \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}0\\0\end{pmatrix} & \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}0\\1\end{pmatrix} \\ & 0 & 1 & 2 & 3 & 4 & 5 \end{array}$$

where letters are bit-vectors. Each variable is described by a *track* along the string. Here, the `P`-track is 111010 and the `Q`-track is 000001. The positions in the string correspond to natural numbers: a "1" in a position means that the number is in the set, a "0" that it is not.

We can define the *language* associated to `simple.mona` as the set of such finite strings that define satisfying interpretations. It is a fact (see Section 4) that for WS1S, this language is regular, i.e. it can be recognized by a finite state automaton.

MONA is able to analyze `simple.mona` automatically by translating it into the minimum automaton recognizing the set of satisfying interpretations. The command

```
mona -c simple.mona
```

produces the automaton, analyzes it, and generates the following output:

```
A counter-example (for assertion => main) of least length (0) is:
P               X
Q               X

P = {}
Q = {}

A satisfying example (for assertion & main) of least length (5) is:
P               X 11101
Q               X 000X0

P = {0,1,2,4}
Q = {}

Conjunctive automaton has 8 states, 19 BDD-nodes and 14 transitions
```

This analysis tells us that a counter-example to the formula is gotten by interpreting both `P` and `Q` as the empty set. MONA has also calculated a satisfying interpretation, namely `P` $= \{0, 1, 2, 4\}$, `Q` $=\{\}$. An `X` in a string means "either `0` or `1`." The columns with the two `X`s are used for Boolean variables, of which there are none in this example.

1.2 MONA uses

The automaton-centered approach to symbolic computations has proved useful in a broad spectrum of applications:

- *Text processing*, where WS1S is a much more versatile notation for patterns than regular expressions. Section 3.6 explains how regular expressions over the ASCII alphabet can be effectively encoded in MONA.
- *Hardware verification*, where parameterized cases and time-dependent circuits are verified without induction [3].

- *Temporal property verification* in various notations such as the duration calculus [32].
- *Verification of distributed systems*, where a program, consisting of several processes, is compared to a specification, also consisting of several processes [23, 24].
- *Assertional reasoning* about programs that manipulate pointers, where comprehensive analyses of data structure shapes and storage usage can be fully automated [18].
- *Description of parse tree shapes*, where simple logical expressions capture source code constraints [22].
- *Reasoning about queues*, where the MONA representation of certain queue structures specializes to those suggested elsewhere in the literature, see Section 2.8.
- *Computational linguistics*, where WS2S can be used to encode Principles-and-Parameters based theories [31].
- *Presburger arithmetic*, where calculations expressing circuit properties can be effectively carried out in MONA [33].
- *Feature logics*, where notations with term symbols of variable arity can be encoded in WS2S [1].

1.3 How to use this manual

Read Section 2 to get started; then skim through Section 3 to see whether there are additional features such as separate compilation or visualization that you will need. Read "Mona & Fido: the logic-automaton connection in practice" [21] for a general introduction to the automaton-logic connection, including BDDs and BDD-represented automata. Or, turn to Section 4. It summarizes the classical automaton-logic connection and explains the three-valued automaton concept used in the MONA implementation. To learn about the WS2S part of Mona, turn to Section 5. If you want to extend MONA, you may consult Appendix C to find a detailed description of the MONA BDD package. Also look in the Appendices for detailed accounts of MONA syntax and command-line usage. To learn about our plans for MONA, see Section 6.

This manual describes MONA version 1.2 as released June 1998. The complete source code for MONA version 1.2 is available for educational and research purposes. Please visit the MONA home page at

`http://www.brics.dk/mona`

for further information.

All bug reports, ideas for future versions, and other comments are appreciated. Our email address is `mona@brics.dk`.

1.4 Acknowledgements

The following people have contributed to the MONA project either by helping making it or by providing us with useful feedback: Ayari Abdelwaheb, David Basin, Nikolaj Bjørner, Morten Biehl Christiansen, Rowan Davies, Jacob Elgaard, Jesper Gulmann Henriksen, Jakob Jensen, Michael Jørgensen, Doug McIllroy, Frank Morawietz, Mogens Nielsen, Robert Paige, Andreas Potthoff, Theis Rauhe, Anders Sandholm, Michael I. Schwartzbach, and Kim Sunesen.

2 Basic features

MONA syntax is essentially that of WS1S or WS2S augmented with lots of syntactic sugar and miscellaneous ways of defining the parameters of the compilation. In this Section, the basic syntax and features concerning WS1S are introduced.

A MONA *program* consists of a number of declarations and formulas, which are all terminated by semicolons. The MONA program itself is valid if the conjunction of all formulas is valid. In Figure 1 below, an extension of the program `simple.mona` from the Introduction is shown. This program imposes additional constraints on `P` and `Q`; it also introduces a first-order variable `x`, which denotes some natural number, and a Boolean variable `A`. We write `var0 A` to declare `A`, since Boolean variables in MONA are regarded as "zero'th-order." Variables introduced by `var` declarations are the free variables of the program. They are also called *global* variables. The existentially quantified formula `ex2 Q: ...` is satisfied if and only if `x` is an even number, since the formula reads: there is a finite set `Q` of numbers such that *(a)* for all numbers `q`, where $0 < \texttt{q} \leq \texttt{x}$, the membership status of `q` in `Q` is the opposite of that of $\texttt{q} - 1$ and *(b)* 0 is in `Q`. The variable `Q` declared by the existential quantifier is a *local variable* whose scope is the formula that follows the colon character.

The formula `A & x notin P` states that `A` is true and that `x` is not in `P`. The whole program is thus satisfied if $\texttt{P}\backslash\texttt{Q} = \{0, 1, 2, 4\}$, `A` is true, and `x` is even and not in `P`.

2.1 The syntax in a nutshell

The `even.mona` example illustrates the main points about MONA syntax. Note how comments are inserted using the # character. One important restriction is not obvious from the example: on the right hand side of a + or a - sign only a term denoting a constant may occur. Thus the term `x + y` is not allowed. Without this restriction, the logic would be undecidable. Other conventional mathematical syntax is rendered in MONA according to Figure 2. Precedence rules are the standard ones, see Appendix A.2.

```
var2 P,Q;
P\Q = {0,4} union {1,2}; # the formula from Section 1

var1 x;
var0 A;

ex2 Q: x in Q
  & (all1 q:
      (0 < q & q <= x) =>
          (q in Q => q - 1 notin Q)
        & (q notin Q => q - 1 in Q))
  & 0 in Q;

A & x notin P;
```

Figure 1: `even.mona`

Numbers (1st order terms)	
0	`0`
$+$	`+`
$-$	`-`

Sets (2nd order terms)	
$\emptyset$	`empty`
$\cup$	`union`
$\cap$	`inter`
$\setminus$	`\`

Formulas (0th order terms)	
0th order arguments	
$\neg$	`~`
$\wedge$	`&`
$\vee$	`|`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`
$\exists$	`ex0 ex1 ex2`
$\forall$	`all0 all1 all2`

Formulas (0th order terms)	
1st order arguments	
$<$	`<`
$>$	`>`
$\leq$	`<=`
$\geq$	`>=`
$=$	`=`
$\neq$	`~=`
2nd order arguments	
$\subseteq$	`sub`
$=$	`=`
$\neq$	`~=`
1st/2nd order arguments	
$\in$	`in`
$\notin$	`notin`

Figure 2: MONA syntax in a nutshell

Other simple constructs

Integer constants can be introduced by `const`-declarations; for example,

```
const c = 1+2*3;
```

declares a constant `c` with value 7.

Local variables can also be declared without quantification by means of a `let`-expression. For example,

```
let1 y = x+c
in y notin Q;
```

declares a first-order variable `y` with value `x+c`. The scope of `y` is the formula following `in`. Local Boolean and second-order variables can similarly be declared with `let0` and `let2` constructs.

There are `min` and `max` terms as well; for example, the value of `min` {`5,3,8`} is 3. Certain modulo calculations, but not all, can be expressed, e.g. `3 + 5 % 6`, which is 2.

2.2 Semantics in a nutshell

Since WS1S is a logic closely related to arithmetic, most constructs have a straightforward mathematical semantics, see Section 4.1. One important property of the semantics is that only *finite* sets can be expressed. For instance, one could *not* have specified the set of even numbers as `EVEN` in:

```
var2 EVEN;
0 in EVEN & all1 p: p in EVEN <=> p+1 notin EVEN;
```

A few constructs need further explanation:

- The value of t-I, where t is a first-order term and I is a constant integer expression, is 0 if the value of t is less than that of I.
- The value of t_1+ I % t_2 (and t_1- I % t_2), where t_1 and t_2 are first-order terms and I is a constant integer expression, is not defined if $t_1 > t_2$. So in that case, the MONA programmer should not assume anything about the value of the term; otherwise, the result is the expected one. (Allowing arbitrary modulo calculation would make the logic undecidable.)
- The value of `min` T and `max` T, where T is a second-order term, is 0 if T denotes the empty set.

2.3 Analyzing formulas

When MONA is run on `even.mona`, the following analysis is printed:

```
A counter-example (for assertion => main) of least length (1) is:
P               X 0
Q               X X
x               X 1
A               0 X

P = {}
Q = {}
x = 0
A = false

A satisfying example (for assertion & main) of least length (7) is:
P               X 1110100
Q               X 000X0XX
x               X 0000001
A               1 XXXXXXX

P = {0,1,2,4}
Q = {}
x = 6
A = true

Conjunctive automaton has 11 states, 35 BDD-nodes and 29 transitions
```

Let us look deeper into the issue of example generation. MONA calculates a *program automaton*, which is the minimum, deterministic automaton whose language is the set of strings that interpret the global variables such that the conjunction of all formulas in the program holds. These strings are like the ones in the introductory example, except that they begin with a bit-vector that interprets the Boolean variables. We might regard this initial letter to be in position -1. Therefore, the language associated with a formula consists of non-empty strings, even if there are no global Boolean variables in the program. Additionally, if there is a global first-order variable like `x` above, then a string must have length at least 2. For example, when `x` is 0, the first letter interprets the Boolean variables, and the second letter, in position 0, is a vector with a "1" in the `x`-track.

MONA analyzes the automaton to find a smallest counter-example by calculating a shortest path from the initial state to a rejecting state, and it prints out a string that puts the automaton in that rejecting state. In the case above, a shortest such string has length 2 (including the Boolean vector). Finding a satisfying example is done similarly, simply by searching for an accepting state instead of a rejecting state.

The minimum automaton calculated is characterized in the output by its number of states, its number of BDD nodes, and its number of transitions, which is here understood as the sum over all states of the number of possible next states.

2.4 Outputting the program automaton

If MONA is executed with option `-w`, then the program automaton is printed. For example,

```
mona -w even.mona
```

generates the following output (somewhat abbreviated here):

```
DFA for formula with free variables: P Q x A
Initial state: 0
Accepting states: 9
Rejecting states: 0 1 2 3 4 5 6 7 8

Conjunctive automaton has 10 states, 33 BDD-nodes and 20 transitions
Transitions:
State 0: XXX0 -> state 1
State 0: XXX1 -> state 2
State 1: XXXX -> state 1
State 2: 0XXX -> state 1
State 2: 100X -> state 3
State 2: 101X -> state 1
State 2: 11XX -> state 1
State 3: 0XXX -> state 1
State 3: 100X -> state 4
State 3: 101X -> state 1
State 3: 11XX -> state 1
  ...
State 9: 0XXX -> state 9
State 9: 10XX -> state 1
State 9: 11XX -> state 9
```

The output is read as follows. First the free variables are printed in index-ordering (see 4.1). Free variables do not include variables that are declared but not used. Next, the initial state, the types of the various states, and the size of the automaton is shown. Finally, a list of transitions is printed. The list of transitions is specified using a bit-vector notation (this time horizontal). This notation is often exponential in the number of BDD nodes.

In Figure 3, the automaton is visualized as a drawing in the traditional DFA style. Note that from the initial state 0, the first three components of the letter do not matter; only the last component, corresponding to the `A`-track, has any importance. If `A` is false, the initial transition brings the automaton to the universally rejecting state 1. Otherwise, the automaton attempts to proceed from state 2 to 7, while taking checking that the set difference of `P` and `Q` is $\{0, 1, 2, 4\}$. Then, it counts modolo 2 in the cycle of length 2, while checking

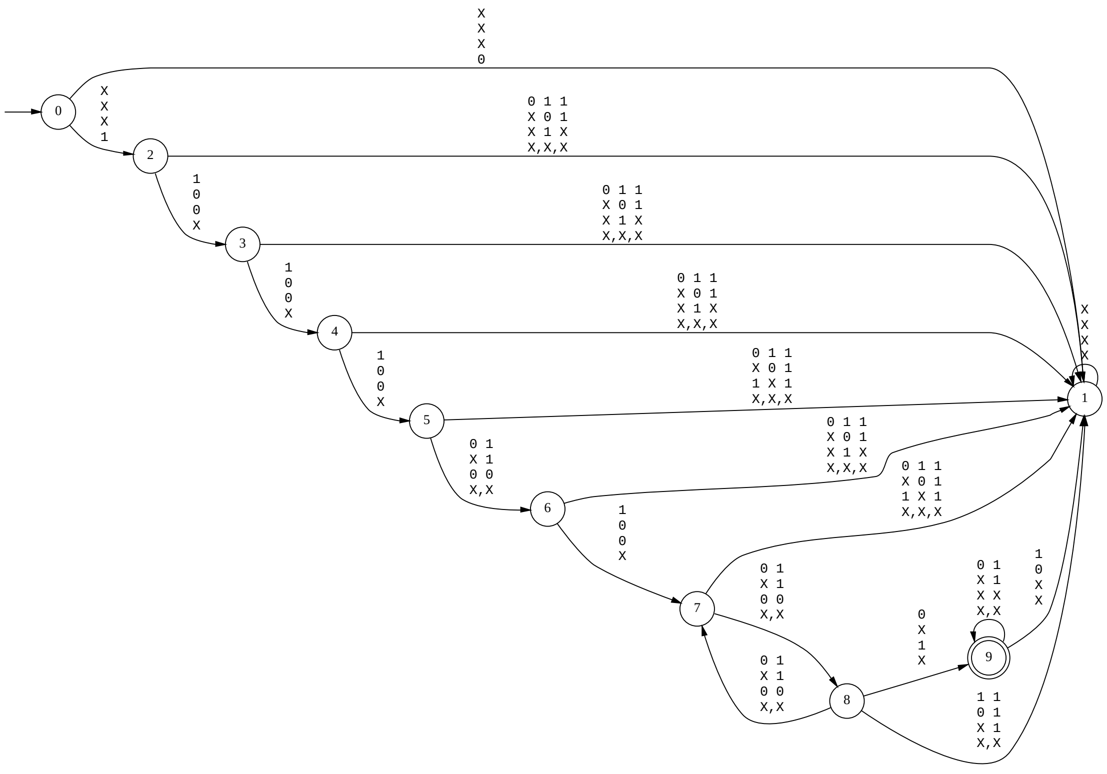


Figure 3: The `even.mona` automaton

the set difference, until `x` occurs. Depending on the parity of `x`, the latter event brings the automaton either into an accepting state or into the universally rejecting state.

Section 3.2 explains to how generate drawings of automata as graphs.

2.5 What are MONA automata really?

The automaton above has 16 (2^4) letters and 10 states. Therefore, its transition table has 160 entries. The MONA representation is a lot more concise than this number would indicate. In fact, the actual data structure for this automaton, depicted in Figure 4, is an acyclic, directed graph with only 35 nodes. The graph shown is essentially a *multi-terminal, shared BDD*, where the leaves are the boxes at the bottom. The internal BDD nodes are round and contain variable indices. Each node has a *low successor* denoted by a dashed arrow and a *high successor* denoted by a solid arrow. In the internal representation automata have three kinds of states: accepting (denoted with 1), rejecting (here denoted with -1), and don't-care (here denoted with 0), see Section 4.2. All states are described with their accept status in the array shown at the top. The only states that are not don't-care states are those that are reached by strings containing at least one occurrence of a "1" in the `x`-track. Variables are indexed from 0 (`P` in the first track) to 3 (`A` in the fourth track).

To see how the transition table is represented, consider state 2 and the letter

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

The next state is gotten as follows: follow the pointer out of the description of state 2 in the array to the BDD node, which has index 0 corresponding to variable `P`. The value of the `P` component is 1, so we go to the high successor, which in turn is marked 1, denoting the `Q`-component. Then, since this bit is 0, we take the low successor to the next node. That node corresponds to `x`, which is 0, so we end up at a leaf mark 4. That is the value of the next state.

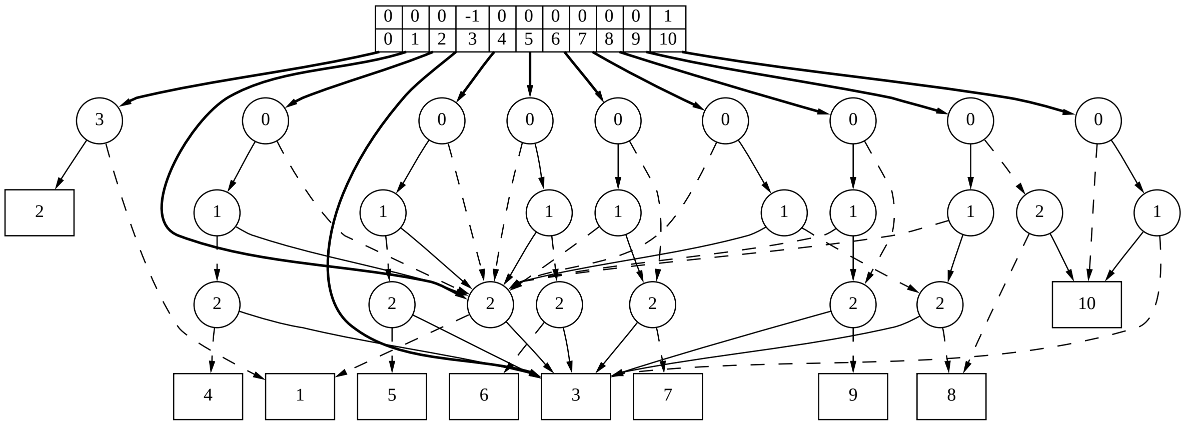


Figure 4: The BDD-based representation of `even.mona`

The graph shown in Figure 4 was generated using the features described in Section 3.7.

2.6 Predicates, macros, and variable indices

MONA provides two methods for parameterizing and reusing formulas. *Macros* are instantiated by syntactical expansion where actual parameters replace formal parameters. *Predicates* are compiled into automata that may be reused. Both macros and predicates may refer to global variables.

In Figure 5 below, `even.mona` has been rewritten such that the even property is expressed as a predicate. MONA is sometimes able to reuse the automaton for a predicate. During compilation MONA assigns unique *indices* to variables in ascending order as they are met. Also, MONA simplifies predicate invocations so that each actual parameter always is represented by a single variable. For example, `even(x+1)` is internally rewritten to `ex1 t: t = x+1 & even(t)`, where `t` is a fresh variable. If a predicate is used twice, where the actual parameters are ordered similarly, then MONA is able to reuse the automaton—similarity of the two parameter orderings means that the actual parameter variables are ordered relative to each other and to the global variables used in the body in the same way. In `even_with_pred.mona`, we could add more uses of the `even` predicate, while no recompilation of it would be necessary, since the predicate has only one free variable, so the variable ordering is trivially unchanged.

Semantically, there is no difference between macros and predicates, but the automata-theoretic calculations performed during their compilation may be very different. As a rule of thumb: use a macro if the formula to be parameterized is small, use a predicate otherwise.

Related issues: Section 3.9 describes the general technique for automatic reuse of intermediate results. Section 3.3 shows how predicates can be used for managing automaton libraries.

2.7 Assertions

It is often the case that the MONA programmer wants to analyze a formula under certain assumptions. For instance, if a program consists of two formulas, ϕ_1 and ϕ_2, one might want to *assert* that ϕ_1 holds, so that any counter-example that is printed satisfies ϕ_1 but not ϕ_2.

MONA provides a method for specifying such assertions. In general MONA forms a formula called ϕ_{main} that is the conjunction of all formulas in the file that are not preceded by the keyword `assert`. The latter kind of formulas are gathered in a conjunction called ϕ_{assert}.

A *counter-example* is an interpretation that does not satisfy the implication

$$\phi_{assert} \Rightarrow \phi_{main}$$

and a *satisfying example* is an interpretation that satisfies the conjunction

$$\phi_{assert} \wedge \phi_{main}$$

To find these interpretations, MONA must in fact calculate two program automata. The automaton corresponding to the implication is called the *implicative* automaton, the one corresponding to the conjunction is called the *conjunctive* automaton. The automaton being printed with the `-w` option is the conjunctive automaton, i.e. it accepts the set of strings corresponding to satisfying examples.

```
var2 P,Q;
P\Q = {0,4} union {1,2};

pred even(var1 p) =
  ex2 Q: p in Q
    & (all1 q:
        (0 < q & q <= p) =>
            (q in Q => q - 1 notin Q)
          & (q notin Q => q - 1 in Q))
    & 0 in Q;

var1 x;
var0 A;

A & even(x) & x notin P;
```

Figure 5: `even_with_pred.mona`

```
var2 P,Q;
assert P\Q = {0,4} union {1,2};

pred even(var1 p) =
  ex2 Q: p in Q
    & (all1 q:
        (0 < q & q <= p) =>
            (q in Q => q - 1 notin Q)
          & (q notin Q => q - 1 in Q))
    & 0 in Q;

var1 x;
var0 A;

assert A & x notin P;
even(x);
```

Figure 6: `even_with_assert.mona`

In Figure 6 below, the example from before has been rewritten using assertions, so that any satisfying example along with any counter-example that might be generated is guaranteed to satisfy both `P\Q = {0,4} union {1,2}` and `A & x notin P`. If one runs MONA on `even_with_assert.mona` as described in Section 2.3, the following satisfying example and counter-example are reported:

```
P = {0,1,2,4}
Q = {}
x = 6
A = true

P = {0,1,2,4}
Q = {}
x = 5
A = true
```

Both examples satisfy the asserted formulas, but their truth value differ on `even(x)`.

2.8 Example: reasoning about queues

In this example, we show how MONA can be used to carry out parameterized verification. The example is inspired by [14], where a data structure that is a hybrid between an automaton and a BDD was proposed. Essentially the same data structure arises in the MONA description we now formulate.

We are modeling queues of arbitrary length that contain elements in $\{0, 1, 2, 3\}$. To do so, we describe a queue `Q` with second-order variables `Qe`, and `Q1`, `Q2`:

- `Qe` denotes the used positions, so we assume it is an initial subset of the natural numbers.
- The four possible membership status combinations that a position `p` has relative to `Q1` and `Q2` encode the value stored at position `p`.

In Figure 7, we have described the encoding. The predicate `Queue` stipulates that the queue has length `l` and that it is ordered. Another predicate, `LooseOne`, expresses the removal of some element from a queue. The formulas of the program hold if there is a queue `Q` of length 4 and a queue `Q'` such that `Q'` contains the value 3 and is the same as `Q` with one element removed. We can indeed recognize such a situation from the MONA output below, where the bit-pattern notation comes in handy:

```
A satisfying example (for assertion & main) of least length (4) is:
Qe               X 1111
Q1               X 0011
Q2               X 0101
Qe'              X 1110
Q1'              X 001X
Q2'              X 011X

Qe = {0,1,2,3}
Q1 = {2,3}
Q2 = {1,3}
Qe' = {0,1,2}
Q1' = {2}
Q2' = {1,2}
```

For other practical examples, visit our Web site.

```
# Qe describes the valid indices of a queue
pred isWfQueue(var2 Qe) =
   all1 p: (p in Qe & p > 0  => p - 1 in Qe);

# isx holds if p contains an x
pred is0(var1 p, var2 Qe, Q1, Q2) = p in Qe & p notin Q1 & p notin Q2;
pred is1(var1 p, var2 Qe, Q1, Q2) = p in Qe & p notin Q1 & p in Q2;
pred is2(var1 p, var2 Qe, Q1, Q2) = p in Qe & p in Q1 & p notin Q2;
pred is3(var1 p, var2 Qe, Q1, Q2) = p in Qe & p in Q1 & p in Q2;

# lt compares the elements at positions p and q of a queue
pred lt(var1 p, q, var2 Qe, Q1, Q2) =
    (is0(p, Qe, Q1, Q2) & ~is0(q, Qe, Q1, Q2))
 |  (is1(p, Qe, Q1, Q2) & (is2(q, Qe, Q1, Q2) | is3(q, Qe, Q1, Q2)))
 |  (is2(p, Qe, Q1, Q2) & (is3(q, Qe, Q1, Q2)));

# isLast holds if p is the last element in the queue
pred isLast(var1 p, var2 Qe) =
  p in Qe & (all1 q': q' in Qe => q' <= p);

# an ordered queue of length l
pred Queue(var2 Qe, Q1, Q2, var1 l) =
   isLast(l - 1, Qe)
&  (all1 p, q: p < q & p in Qe & q in Qe => lt(p, q, Qe, Q1, Q2));

# eqQueue2 compares elements in two queues
pred eqQueue2(var1 p, q, var2 Q1, Q2, Q1', Q2') =
    (p in Q1 <=> q in Q1') & (p in Q2 <=> q in Q2');

# LooseOne holds about a queue Q and a queue Q' if
# queue Q' is the same as Q except that one
# element (denoted by p below) is removed
pred LooseOne(var2 Qe, Q1, Q2, Qe', Q1', Q2') =
  ex1 p: p in Qe
  &  (all1 q: (~isLast(q, Qe) => (q in Qe  <=> q in Qe'))
         & (isLast(q, Qe) => (q notin Qe')))
  &  (all1 q: q < p & q in Qe  => eqQueue2(q, q, Q1, Q2, Q1', Q2'))
  &  (all1 q: q > p & q in Qe  => eqQueue2(q, q - 1, Q1, Q2, Q1', Q2'));

var2 Qe, Q1, Q2;     # the queue Q
var2 Qe', Q1', Q2';  # the queue Q'

assert isWfQueue(Qe);

# the primed variables denote a queue of length 3 containing
# three of the elements 0, 1, 2, 3 in that order and the element 3
Queue(Qe, Q1, Q2, 4);   # the queue Q is a queue of length 3
                        # containing the elements 0, 1, 2, 3
LooseOne(Qe, Q1, Q2, Qe', Q1', Q2'); # Q' is Q except for one element
ex1 p: is3(p, Qe', Q1', Q2'); # Q' does contain the element 3
```

Figure 7: lossy_queue.mona

3 Advanced features

This section describes debugging features, import-export mechanisms, graphical output, and techniques for producing finite-state automata that are not tied to WS1S semantics.

3.1 Dumping translation information

MONA provides a number of facilities for monitoring the translation process. These options can be used for "debugging" MONA code in several ways. For instance, the `-p`, `-s` and `-t` options can reveal if state explosions occur—something that likely happens if complicated properties are expressed. The user can then attempt to rewrite that particular part of the source code and rerun MONA.

Progress information

The `-p` option makes MONA print progress information. It prints the name of the current translation phase, the time spent in each phase and while in the automaton-construction phase a completion percentage. Also, information about the DAG representing the code (3.9) is shown.

As an example,

```
mona -c -p lossy_queue.mona
```

yields the following output (where the counter-example and the satisfying example are omitted for clarity):

```
MONA v1.2 for WS1S/WS2S
Copyright (C) 1997-1998 BRICS

PARSING
Time: 00:00:00.020

TYPE CHECKING
Time: 00:00:00.000

CODE GENERATION
DAG hits: 303, misses: 137, nodes: 137
Time: 00:00:00.020

AUTOMATON CONSTRUCTION
100% completed
Time: 00:00:00.090

ANALYSIS
A counter-example (for assertion => main) of least length (4) is:
  ...

A satisfying example (for assertion & main) of least length (1) is:
  ...

Conjunctive automaton has 11 states, 43 BDD-nodes and 28 transitions
```

```
Total time: 00:00:00.130
```

The completion percentage shows how many of the automaton operations are completed.

Dumps

The `-d` option causes the following information to be printed:

- The main formula, assertions (Section 2.7), macros and predicates.
- The contents of the symbol table (mainly used for debugging purposes).
- The contents of the DAG (Section 3.9) shown as a tree.
- In tree mode (Section 5), the guide is also printed.

Statistics

The `-s` option makes MONA print some statistics about each automaton operation. It prints

- the type of the current operation,
- the sizes of the input and the resulting automata, and
- the location in the source code where the operation originates.

The output looks like:

```
    ...
Product & 'even.mona' line 11
    & 0 in Q;
    ^
  (7,14)x(4,4) -> (12,26)
  Minimizing (12,26) -> (6,11)
Right-quotient
Projecting offset 3 'even.mona' line 6
  ex2 Q: x in Q
  ^
  (6,11) -> (6,7)
  Minimizing (6,7) -> (5,6)
    ...
```

This can be read as follows: the conjunction in line 11 results in a product operation of two automata, one with 7 states and 14 BDD nodes, the other with 4 states and 4 BDD nodes. The product automaton with 12 states, 26 BDD nodes and its minimized version are also reported on, and so forth. The ^ character points to the column in the source line where the operation originates.

When the compilation is completed, a summary of the use of the various automata operations is printed. Also, the largest number of states and the largest number of BDD nodes of any intermediate, minimized automata is shown.

Timing

Using the option `-t` cause MONA to print a summary of the total time spent in the main automaton operations. If `-t` is used together with `-s` (statistics), the time spent in each operation is also printed.

Intermediate automata

The `-i` option is intended for use together with `-s` (see above). It extends the information being printed at each automaton operation with a verbose description of the resulting automaton.

The following example illustrates the output:

```
  ...
Resulting DFA:
Initial state: 0
Accepting states: 1
Rejecting states: 2
Don't-care states: 0
Transitions:
State 0:  -> state 1
State 1: #8=0, #10=0, #11=0 -> state 1
State 1: #8=0, #10=0, #11=1 -> state 2
State 1: #8=0, #10=1, #11=0 -> state 2
State 1: #8=0, #10=1, #11=1 -> state 1
State 1: #8=1, #11=0 -> state 2
State 1: #8=1, #11=1 -> state 1
State 2:  -> state 2
  ...
```

The format resembles the one from Section 2.4. The only differences are that "don't-care"-states can occur and that the transitions are written more explicitly. For example, `#8=1` means that the variable with index 8 has the value 1.

3.2 Visualization of automata

In WS1S mode, a graph representation of the conjunctive automaton can be generated with the `-gw` option. The format of the output is readable by the graphviz tool dot (property of AT&T). Example:

```
mona -gw even.mona > even.dot
dot -Tps even.dot -o even.ps
```

generates the graph shown on page 12 to the file `even.ps`. (The graphviz tools can be found at `http://www.research.att.com/sw/tools/graphviz/`.) Section 3.7 describes another kind of automaton visualization.

3.3 Separate compilation

MONA supports efficient use of libraries of predicate definitions through the *separate compilation* feature. The user can divide the input file into a number of smaller files and then use the `include`-construct to combine them. If MONA is executed with the option `-e`, all automata

```
# example predicate library

pred foo(...) = ...;

pred bar(...) = ...;
```

Figure 8: `library.mona`

```
# example application

include "library.mona";

...application...
```

Figure 9: `application.mona`

corresponding to predicate uses are stored in an automaton library, and automatically reused in subsequent executions of MONA. The user can decide where to store the automaton library by setting the environment variable `MONALIB`. When MONA finds out that a source file has changed (using the file time-stamp), the part of the automaton library containing automata for that specific source file is recomputed.

The automaton library is a directory containing a sub-directory for each source file. Each sub-directory contains a number of automaton files (in the format also used by the `import`/`export` mechanism, see 3.7) and a special contents-file.

In Figures 8 and 9, a template for separate compilation is shown.
Executing

```
mona -c -e application.mona
```

cause MONA to store the automata generated by applications of the predicates in the directory `$MONALIB/library/`. In subsequent compilations of the application file (`application.mona`), MONA will attempt to reuse the stored automata unless the library source file (`library.mona`) has been altered.

3.4 Restriction

MONA allows restrictions to be associated with the use of variables. At every kind of declaration of a first-order or second-order variable (e.g., `var2`, `all1`, etc.), it is possible to specify `where` ρ, which makes ρ the user-defined restriction of that particular variable. See 4.2 for a discussion of the semantics of restrictions. Note that restrictions *must* always be satisfiable for MONA to calculate correctly, see 4.2. MONA does *not* check this requirement.

It is possible to specify *default* user-definable restrictions using the constructs

`defaultwhere1(`p`) = `ϕ

and

```
defaultwhere2(P) = ϕ
```

For instance,

```
defaultwhere1(x) = x <= 7;
```

is equivalent to specifying `where` p `<= 7` for each first-order variable p that does not already have an explicit `where` restriction.

3.5 Emulating Monadic Second-order Logic on Strings

Any automaton produced by the MONA in its pure WS1S mode recognizes a language that is closed under certain string operations, see Section 4. For some applications, it is desirable to have the ability to generate any possible automaton over a bit-vector alphabet.

A slight variation on WS1S is called *Monadic Second-order Logic on Strings* (M2L-Str). In M2L-Str, all quantification is restricted to numbers less than the length of the string that the formula is interpreted over. This restriction is not a WS1S concept, because of the closure property just mentioned. We turn MONA into M2L-Str mode if we write

```
m2l-str;
```

in the top of a MONA program. This declaration is just an abbreviation for:

```
linear;
var1 $;
lastpos $;
defaultwhere1(p) = p <= $;
defaultwhere2(P) = P sub {0,...,$};
```

where the `$` variable denotes the last position in the string. The `defaultwhere`-declarations restrict all variables to $0, 1, \ldots,$ `$`. The non-WS1S concept of the last position in the string is enforced by the `lastpos` declaration. It works by conjoining a special basic automaton to the implicative and the conjunctive automata (see Section 3.4) that ensures that `$` is interpreted as the last position; it also causes the `$`-variable to be projected away. Therefore, it will not occur in the program automata or in the analysis.

3.6 Example: regular expressions over the ASCII alphabet

Regular expressions can be easily translated into MONA. Consider the problem of constructing an automaton over the ASCII alphabet that recognizes the language `a*(ab)*`. This can be done in a straightforward manner: we introduce a first-order variable `$` to denote the end of the string. We declare second-order variables `bit0`, ... , `bit7`—restricted to $\{0, \ldots,$ `$`$\}$—for the representation of a string of ASCII characters. We make the default restriction on first and second-order variables that they involve numbers less than or equal to `$` $+ 1$. Then, we recursively construct for each subexpression E of `a*(ab)*` a predicate `is_`E`(`p`, `q`)` such that `is_`E`(`p`, `q`)` holds if and only if the substring from position p to position $q - 1$ belongs to the language defined by E. Finally, we remember to declare `$` as a `lastpos` variable. The details can be found in Figure 10. It can been seen to be essential that restrictions are relative to

```
var1 $;
lastpos $;

defaultwhere1(p) = p <= $+1;
defaultwhere2(P) = P sub {0,...,$+1};

# we declare a string of 8-bit vectors
var2 bit0 where bit0 sub {0,...,$}; var2 bit1 where bit1 sub {0,...,$};
var2 bit2 where bit2 sub {0,...,$}; var2 bit3 where bit3 sub {0,...,$};
var2 bit4 where bit4 sub {0,...,$}; var2 bit5 where bit5 sub {0,...,$};
var2 bit6 where bit6 sub {0,...,$}; var2 bit7 where bit7 sub {0,...,$};

macro consecutive_in_set(var1 p, var1 q, var2 P) =
p < q & p in P & q in P & all1 r: p < r & r < q => r notin P;

# ASCII 'a' is 97, which is 1100001
macro is_a(var1 p, var1 q) =
q = p + 1 &
p in bit0 & p notin bit1 & p notin bit2 & p notin bit3 &
p notin bit4 & p in bit5 & p in bit6 & p notin bit7;

# ASCII 'b' is 98, which is 1100010
macro is_b(var1 p, var1 q) =
q = p + 1 &
p notin bit0 & p in bit1 & p notin bit2 & p notin bit3 &
p notin bit4 & p in bit5 & p in bit6 & p notin bit7;

# we concatenate by guessing the intermediate position where
# the string parsed according to the first regular expression
# (in this case "a") ends and the string parsed according to
# the second (in this case "b") starts

macro is_ab(var1 p, var1 q) =
ex1 r: is_a(p, r) & is_b(r, q);

# a star expression is handled by guessing the set of
# intermediate positions
macro is_ab_star(var1 p, var1 q) =
ex2 P: p in P & q in P &
       all1 r, r': consecutive_in_set(r, r', P) => is_ab(r, r');

macro is_a_star(var1 p, var1 q) =
ex2 P: p in P & q in P &
       all1 r, r': consecutive_in_set(r, r', P) => is_a(r, r');

macro is_a_star_ab_star(var1 p, var1 q) =
ex1 r: is_a_star(p, r) & is_ab_star(r, q);

is_a_star_ab_star(0, $+1);
```

Figure 10: Regular expression over ASCII alphabet

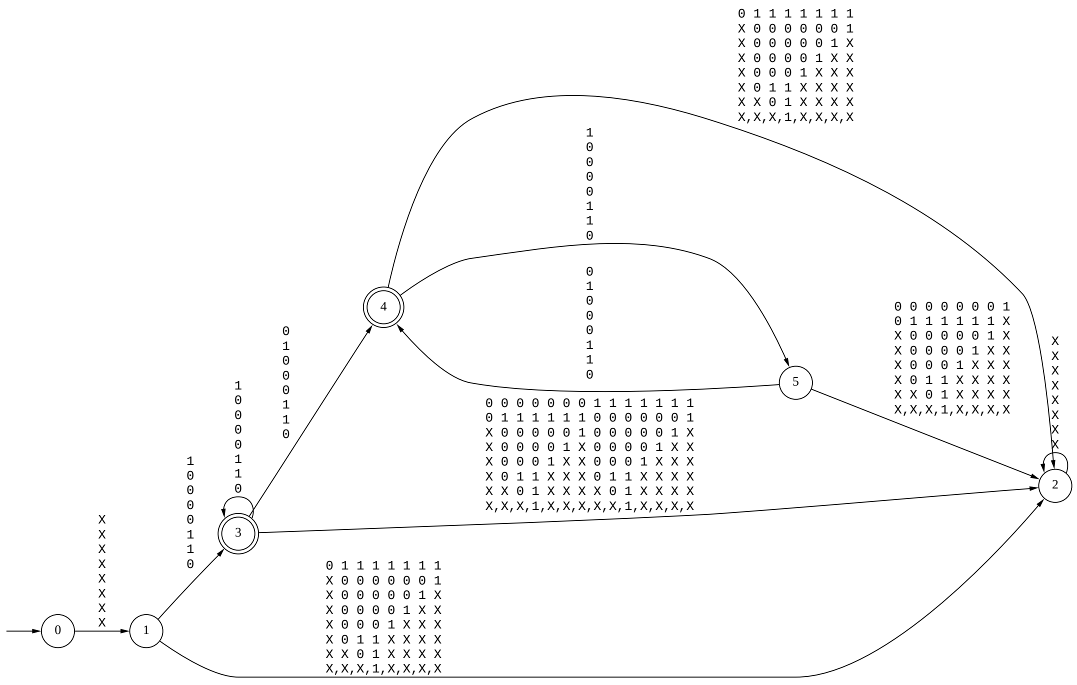


Figure 11: Regular expression over ASCII alphabet

$\$ + 1$; otherwise, the induction would be much more cumbersome to express. The resulting automaton is shown in Figure 11. It can be seen that the representation of intervals, like `a-z` would be very efficient, by the approach just sketched. Thanks to the BDD representation, any interval on which the transition relation remains constant for a given state introduces at most sixteen nodes, but on average only eight nodes. Also, the regular expressions can easily be extended with complementation and conjunction. Thus MONA should be a useful tool for the construction of automata for pattern matching via more concise formalisms than regular expressions.

3.7 Exporting and importing automata

If MONA is used as an automaton-generation tool, two features are available for *exporting* automata. The first is using the `-w` option as described in Section 2.4. The other is the formula-level construct

```
export("filename", ϕ)
```

which evaluates to the same (minimal) automaton as ϕ but has the side-effect of writing the automaton to the designated file. If `lastpos` p (see 3.5) has been specified, for instance by writing `m2l-str`, the variable p will be projected away from the automaton before the exporting takes place.

The automaton is stored in "`.dfa`"-format which is BDD-representation plus some auxiliary information written in ASCII. For instance,

```
var2 P,Q;
export("test.dfa", P sub Q);
```

yields the following contents of `test.dfa`:

```
MONA DFA
number of variables: 2
variables: P Q
orders: 2 2
states: 3
initial: 0
bdd nodes: 4
final: 0 1 -1
behaviour: 0 1 3
bdd:
-1 1 0
0 0 2
1 3 0
-1 2 0
end
```

Automata in "`.dfa`"-format can be imported using the construct

`import("`*filename*`",` n_1`->`n'_1`,` n_2`->`n'_2`, ... ,` n_k`->`n'_k`)`

which evaluates to the automaton stored in the designated file where the free variables $n_1, n_2, ..., n_k$ (from the `variables`-list in the `dfa`-file) have been substituted by $n'_1, n'_2, ..., n'_k$ respectivly.

It is checked that the type of each n_i is the same as that of n'_i. Also, the substitution must be order preserving in terms of variable indices.

Using DFAs in other applications

A simple C application programming interface for external DFAs is available in the MONA package (as `Lib/dfalib.[ch]`). It allows the user to load, manipulate, and store DFAs in the external DFA formal and to "run" a given string on the DFAs completely independently of MONA.

The interface file `dfalib.h` contains the following functions:

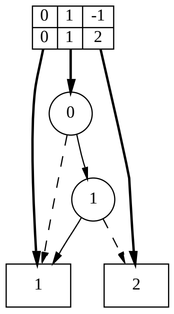
`mdDfa *mdLoad(char *filename)` - loads a DFA file into memory.

`int mdStore(mdDfa *dfa, char *filename)` - stores a DFA in a file. If the stored automaton is modified and used in MONA (using `import`), the user must ensure that the content is consistent and that the BDD is properly reduced and ordered.

`void mdFree(mdDfa *dfa)` - deallocates the memory used by the DFA.

`mdState mdDelta(mdDfa *dfa, mdState s, mA a)` - represents the transition function $\delta : Q \times \Sigma \rightarrow Q$ of the automaton. The parameter `a` refers to an array of 0/1 chars, which denotes an alphabet symbol. The array has one entry per variable.

`void mdDump(mdDfa *dfa, FILE *file)` - dumps the automaton to the designated file in `graphviz` format. The following graph is generated for the `P sub Q` example above:



The complete structure of the BDD representation is shown. A larger example is shown in Figure 4. See [21] for a description of the use of BDDs in MONA. The `graphviz` tool is described in Section 3.2.

An example application

A simple application of `dfalib` is located in the `Lib` directory as `dfa2dot.c`. It takes two command-line arguments: the name of a source `dfa`-file, and the name of a destination `dot`-file. It simply reads the `dfa`-file using `mdLoad` and dumps it using `mdDump` to the `dot`-file. This `dot`-file can then later be processed with the `dot` tool.

3.8 Prefix-closing

The predicate notation `prefix(`ϕ`)` stands for an operation that calculates the prefix-closure of the automaton associated to ϕ. In other words, a string w satisfies a formula `prefix(`ϕ`)` if and only if there is some string v so that $w{\cdot}v$ satisfies ϕ.

The `prefix` operation has no logical meaning, but it may be useful in the synthesis of controllers for reactive systems.

3.9 Controlling reuse of intermediate results

Internally MONA is divided into a front-end and a back-end. The front-end parses the input and reduces it to an `internal code` describing the automata-theoretic operations that will calculate the resulting program automata. The back-end then carries out the automata operations in a way similar to the simplified decision procedure explained in Section 4.

For efficiency reasons, the internal code is stored in a DAG (Directed Acyclic Graph), not a tree. The atomic formulas are located in the leaves and the composite constructs are in the internal nodes.

The DAG can be thought of as being constructed from the tree using a bottom-up collapsing process. This process is based on two kinds of formula equivalence relations:

syntax equivalence Two formulas are *syntax-equivalent* if their code trees are identical. Obviously, this notion of equivalence implies identical corresponding automata and hence suggests a very simple kind of reuse of intermediate results.

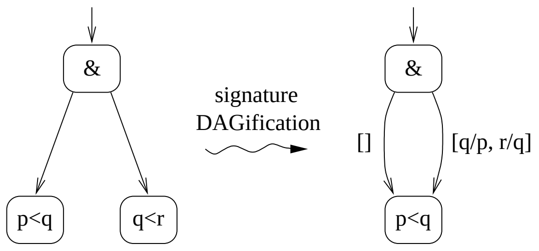
signature equivalence Two formulas ϕ and ϕ' are *signature-equivalent* if there is an order-preserving renaming of the variables in ϕ (i.e., increasing w.r.t. the indices of the variables) such that ϕ and ϕ' become syntax-equivalent. Signature equivalence is implied by syntactic equivalence.

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 property is used by MONA for automatic reuse of intermediate results.

For example, consider the formula

```
ex1 q: p<q & q<r
```

where the variables `p`, `q`, `r` have the indices $1, 2, 3$ respectively. The automata for the subformulas `p<q` and `q<r` are isomorphic, so their tree nodes are collapsed. The edges of the resulting DAG are labelled with the renaming information.



Unfortunately, it takes linear time to calculate the signature-equivalence class of a single subtree, so the total expense of the "DAGification"-process becomes quadratic. Therefore the signature-equivalence collapsing is limited to subtrees for which the number of variable occurences are less than the "DAGification degree", which is set by the `-r`N option. The rest of the subtrees are collapsed using syntax equivalence.

Experiments show that significant time improvements (at the expense of larger space usage) using "DAGification" (`-r100`) are possible, up to a factor 10.

The contents of the DAG can be shown as a graph using the `-gd` option and the graphviz dot tool. See 3.2 for more information about visualization using graphviz.

4 The automaton–logic connection

Since the MONA notation is a logic interpreted over natural numbers, it is not difficult to define its formal semantics. But a programming notation also has a *compilation semantics*, which details how the source code is transformed. Usually, an exact description of the compilation, such as the source code for a compiler, is neither available nor useful to a programmer. With MONA, however, we need to carefully explain the compilation, since the process may involve heavy or even infeasible symbolic calculations. Our explanation is in two parts: first, we explain the classical WS1S approach (Büchi [9] and Elgot [12]); next, we introduce a version of WS1S where restrictions are syntactically manifest, and we explain the semantic and automata-theoretic consequences.

4.1 The classical approach

The central idea in the classical approach is to recursively translate each subformula of a main formula ϕ_0 into a deterministic, finite-state automaton that represents the set of satisfying interpretations. The approach can be presented as follows.

Simplifying the language

Before the actual translation takes place, some syntactic transformations of ϕ_0 are performed:

- First-order terms are encoded as second-order terms, since a first-order value can be seen as a singleton second-order value. Also, we may encode Booleans in a variety of ways, but for simplicity, we assume here that strings start with position 0 (as opposed to the actual encoding explained previously).
- All second-order terms are "flattened" by introducing variables that contain the values of all subterms. Example: `A = (B union C) inter D` is reduced to `ex2 V: A = V inter D & V = B union C`, where `V` is a fresh variable.
- The subformulas are rewritten to involve fewer kinds of operations. Example: `all2 A:` ϕ is reduced to `~(ex2 A: ~`ϕ`)`.

Consequently, it can be shown that ϕ_0 can be massaged into a form where atomic subformulas (those without Boolean connectives or quantifiers) express either a subset, a set difference, or a successor relation, and where each logical operator expresses either a negation, a conjunction, or an existential quantification. The abstract syntax for simplified WS1S formulas can be defined by the following grammar (where P_i ranges over a set of variables):

$$\begin{array}{lllllll} \phi & ::= & \texttt{\~{}}\phi' & | & \phi' \mathbin{\texttt{\&}} \phi'' & | & \texttt{ex2 } P_i : \phi' \\ & | & P_i \texttt{ sub } P_j & | & P_i \texttt{ =} P_j \setminus P_k & | & P_i \texttt{ =} P_j \texttt{ +1} \end{array}$$

Semantics of the simplified language

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. Assume every variable of ϕ_0 is assigned a unique index in the range $1, 2, .., k$. Let P_i denote the variable with index i. As indicated in Section 1.1, the string w now determines an interpretation $w(P_i)$ of P_i defined as the finite set $\{j \mid \text{the } j\text{th bit in the } P_i\text{-track is } 1\}$. For example, the formula $\phi_0 \equiv \exists\mathtt{C} : \mathtt{A} = \mathtt{B}\backslash\mathtt{C}$ has variables `A`, `B`, and `C`, which we can assign indices 1, 2, and 3, respectively. A typical string w over $\mathbb{B}^3$ looks like:

$$\begin{array}{cc}
\begin{array}{c}\mathtt{A}\\\mathtt{B}\\\mathtt{C}\end{array} & \begin{pmatrix}1\\0\\1\end{pmatrix}\begin{pmatrix}0\\0\\1\end{pmatrix}\begin{pmatrix}1\\1\\0\end{pmatrix}\begin{pmatrix}0\\0\\0\end{pmatrix} \\
 & \;\;0\qquad\;\;1\qquad\;\;2\qquad\;\;3
\end{array}$$

This string interprets all three variables although `C` is not free in ϕ_0. Thus, the language associated with ϕ_0 is independent of the `C`-track in the sense that changing the bits on the track for a string w does not affect the membership status of w. Also note that suffixing w with any string of the form

$$\begin{pmatrix}0\\\vdots\\0\end{pmatrix}^*$$

defines the same interpretation as w. Therefore, we will say that w is *minimum* if there is no such non-empty suffix.

The semantics of a formula ϕ 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:

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

where we use the notation $w[P_i \mapsto M]$ for the shortest string w' that interprets all variables P_j, $j \neq i$, as w does, but interprets P_i as M. Note that if we assume that w is minimum, then w' decomposes into $w' = w \cdot w''$, where w'' is a string of letters of the form

$$\begin{pmatrix}0\\\vdots\\0\\X\\0\\\vdots\\0\end{pmatrix} \tag{1}$$

where the ith component is the only one that may be different from 0.

The semantics allows quantification over only *finite* sets of naturals. Without this restriction, the logic is simply S1S (Second-order logic of One Successor), which is also non-elementary decidable [10], although no efficient decision procedure is known.

Automaton construction

For a formula ϕ, define its *language* $\mathcal{L}(\phi)$ as the set of satisfying strings:

$$\mathcal{L}(\phi) = \{w \mid w \vDash \phi\}$$

We now formulate the automata-theoretic calculations that allow us to conclude that any $\mathcal{L}(\phi)$ is a regular language. Thus, we will show by induction on the formula how to construct deterministic automata A such that $\mathcal{L}(A) = \mathcal{L}(\phi)$, where $\mathcal{L}(A)$ is the language recognized by A. The atomic formulas are hand-translated into what we call *basic* automata. Composite formulas correspond to well-known automaton operations.

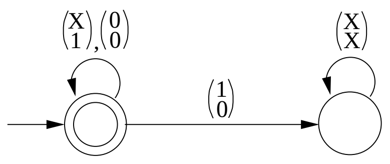
Translating atomic formulas

For simplicity, we only show basic automata for the case where $i = 1, j = 2, k = 3$. We also only show the first two or three components of a letter.

$\phi = P_1$ `sub` P_2**:** The automaton must recognize the language

$$\mathcal{L}(P_1 \texttt{ sub } P_2) = \{w \in (\mathbb{B}^k)^* \mid \text{for all letters in } w\text{: if the first component is 1, then so is the second}\}$$

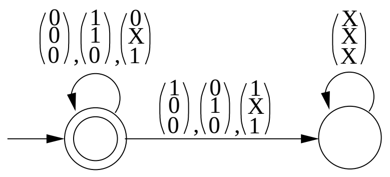
Thus, the automaton is:



$\begin{pmatrix}X\\1\end{pmatrix},\begin{pmatrix}0\\0\end{pmatrix}$
$\begin{pmatrix}1\\0\end{pmatrix}$
$\begin{pmatrix}X\\X\end{pmatrix}$

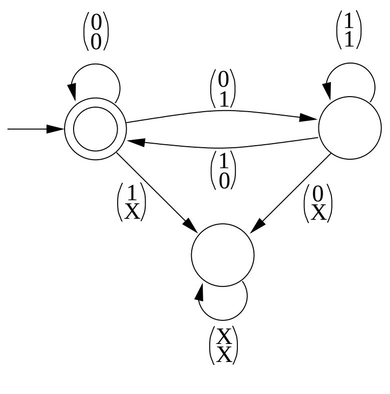
The other atomic formulas are treated similarly.

$\phi = P_1{=}P_2\backslash P_3$**:**



$\begin{pmatrix}0\\0\\0\end{pmatrix},\begin{pmatrix}1\\1\\0\end{pmatrix},\begin{pmatrix}0\\X\\1\end{pmatrix}$
$\begin{pmatrix}1\\0\\0\end{pmatrix},\begin{pmatrix}0\\1\\0\end{pmatrix},\begin{pmatrix}1\\X\\1\end{pmatrix}$
$\begin{pmatrix}X\\X\\X\end{pmatrix}$

$\phi = P_1{=}P_2{+}1$**:**



$\begin{pmatrix}0\\0\end{pmatrix}$
$\begin{pmatrix}1\\1\end{pmatrix}$
$\begin{pmatrix}0\\1\end{pmatrix}$
$\begin{pmatrix}1\\0\end{pmatrix}$
$\begin{pmatrix}1\\X\end{pmatrix}$
$\begin{pmatrix}0\\X\end{pmatrix}$
$\begin{pmatrix}X\\X\end{pmatrix}$

Translating composite formulas

ϕ = `~`ϕ': Negation of a formula corresponds to automaton complementation, so if we have already calculated A' such that $L(\phi') = \mathcal{L}(A')$, then

$$\mathcal{L}(\texttt{\~{}}\phi') = \complement\mathcal{L}(\phi') = \complement\mathcal{L}(A') = \mathcal{L}(\complement A'),$$

where $\complement$ denotes both the language-theoretic operation of complementation and automata-theoretic operation of complementation. Thus, $A = \complement A'$ is the desired automaton for ϕ. Naturally, the automaton operation $\complement$ in MONA is implemented as the linear-time complementation algorithm that simply consists of flipping accepting and rejecting states.

ϕ = ϕ' `&` ϕ'': Conjunction corresponds to language intersection:

$$\mathcal{L}(\phi' \,\&\, \phi'') = \mathcal{L}(\phi') \cap \mathcal{L}(\phi'').$$

So the desired automaton A is the automaton product $A' \times A''$ of the automata for the subformulas. In MONA, only the reachable product states—pairs of the form (s', s''), where s' and s'' are states of A' and A''—are calculated. Usually, this set is much smaller than the product state space.

ϕ = `ex2` P_i : ϕ': Intuitively, the desired automaton A acts as the automaton A' for ϕ' except that it is allowed to guess the bits on the P_i-track. Let A'' be the result of such an automaton-theoretic *projection operation* on A', that is, A'' is just a non-deterministic version of A', where there are up to two transitions possible out of each state on each letter. In order to acquire a deterministic automaton A, we must further subject A'' to a subset construction to get A. Unfortunately, already the A'' automaton does not quite do the job. The problem is that the witness $w[P_i \mapsto M]$ may be longer than w. However, if this is the case, then it can be seen that there is a state s in A'' reachable on w from the initial state such that a final state can be reached from s on a string consisting of letters of the form (1). Thus, it suffices to characterize such states s as accepting before the projection and subset construction are carried out. This step can be carried out in linear time by a breadth-first, backwards exploration of the automaton from final states. The subset construction is carried out so that only reachable subset states are calculated. In practice, this construction is often linear, not exponential as it may be feared [3].

In language-theoretic terms, the operations above can be characterized as follows. Let the *right-quotient* of a language L with a language L' be defined as

$$L/L' = \{w \mid \exists u \in L' : \; w{\cdot}u \in L\}.$$

Also, let the *projection operation* E^i be defined by

$$E^i(L) = \{w \mid \exists w' : \; w \text{ is identical to } w' \text{ except for the } P_i\text{-track}\}.$$

Choose the language L^i to be

$$L^i = \{w \in (\mathbb{B}^k)^* \mid \text{the } P_j\text{-track } w \text{ is of the form } 0^* \text{ for } j \neq i\}$$

It can then be proven that

$$\mathcal{L}(\texttt{ex2}\ P_i : \phi') = E^i(\mathcal{L}(\phi')/L^i),$$

which is a language-theoretic explanation of the automata calculations outlined above.

Issues in the classical approach

The elimination of first-order variables poses conceptual and practical problems. The conceptual problem is that viewing a first-order term t as a second-order term T relies on restricting Ts values to be singleton sets where the sole element denotes the value of t; therefore, the semantics is not closed under complementation. For example, the formula $\phi =$ `p=0`, where `p` is first-order is handled as $\phi' =$ `P={0}`, where `P` is second-order. But the complement of ϕ' is `~(P = {0})`, something that is different from the representation of `~(p = 0)`, namely `~(P = {0}) & singleton(P)`, where `singleton(P)` is a predicate denoting that `P` is a singleton set. So the problem boils down to a simple fact: regarding formulas under conjunctive restrictions is not robust under negation.

The practical problem is that if we try to keep automata in a normal form, where the singleton restriction for all first-order variables is obeyed, then additional product and minimization calculations would be necessary: for each automaton A representing a subformula ϕ and each free variable P_i, the automaton representing the singleton property for P_i must be conjoined to A.

The singleton restriction is not the only one we need. For example, to emulate the semantics of the Monadic Second-order Semantics on Strings in WS1S (see Section 3.5), we must restrict all first and second-order terms to numbers less than or equal to a value `$` denoting the end of the string. A naive approach, where each occurrence of a first-order variable `p` in an atomic formula is accompanied by conjoining the restriction `p <= $` to the atomic formula, may easily lead to doubly-exponential blow-ups [28]. Instead, we would like a general semantic mechanism, and a simple syntactic means, of safely compiling formulas under such constraints.

4.2 The MONA approach

MONA uses automata that extend the classical translation in three ways:

- To address the issues in the classical approach, MONA uses the ternary partitioning of strings suggested in [28]. Thus, the states of MONA automata are partitioned into three kinds: *accepting*, *rejecting*, and *don't-care*.

- To handle Boolean variables efficiently, MONA automata read an initial letter that encodes only Boolean variables before reading the letters that define first and second-order variables. This was explained in Section 2.

- A larger number of basic automata and Boolean connectives are used in order to reduce the overhead of simplifying the formulas.

WS1S with restrictions

We introduce WS1S-R, a version of the simplified WS1S above where restrictions are now made explicit by the syntax. The basic notion is that a variable P can be associated with a *restriction* ρ, which is a formula restraining the values of P. For example,

$$\phi = \texttt{ex2 X where } \underbrace{\texttt{X sub \{0,...,\$\}}}_{\rho} : \phi$$

restricts `X` so that the formula ϕ is compiled under the restriction $\rho = \texttt{X} \subseteq \{0,\ldots,\$\}$. Of course, the automaton for ϕ is equivalent to the one calculated as

```
ex2 X: X sub {0,...,$} & ϕ'
```

However, our intention with a restriction is that it should be implicitly conjoined to any subformula mentioning P.

To do this, we assume again that all formulas are subformulas of a main formula ϕ_0. We focus on the simplified syntax, but we change the rule for existential quantification to:

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

Let $\boldsymbol{\rho}(P_i) = \rho$ be the restriction of variable P_i. We assume that P_i is free in $\boldsymbol{\rho}(P_i)$. Also, we assume that each P_i is restricted, possibly to the formula P_i=P_i, i.e., `true`.

The ternary semantics

Under the binary semantics, a formula ϕ is either *true* (1) or *false* (0) for a given interpretation. MONA semantics defines a third possibility: that the formula ϕ is *don't-care* ($\perp$) if the restriction of some free variable in ϕ is not fulfilled.

As an example, consider the following MONA program:

```
var1 $ where $ <= 5;
var1 p where p <= $;
p > 5;
```

The status of $\phi_0 = \texttt{p} > \texttt{5}$ under the interpretation $[\texttt{p} \mapsto i]$ is $\perp$ if $i > 5$ and 0 for $i \leq 5$. In particular, `p > 5` is not satisfiable under the conjunction of the restrictions for `p` and `$`.

It is shown in [28] how this notion yields a robust semantics: variables can be constrained where they occur, and the meaning of constraints is preserved under the connectives and quantifiers. Here, we briefly explain the semantics along with its automata-theoretic implications.

Let X be an expression, that is X is either a formula or a variable. We define $\boldsymbol{\rho}^*(X)$ to be the formula that is the conjunction of all $\boldsymbol{\rho}(P_i)$, where P_i is free in X, or free in $\rho(P_j)$ for some P_j free in X, etc.; that is,

$$\boldsymbol{\rho}^*(X) = \mathop{\&}_{P_i \in \mathcal{P}} \quad \boldsymbol{\rho}(P_i),$$

where $\mathcal{P}$ is the least set such that $FV(X) \subseteq \mathcal{P}$ and $\cup_{P \in \mathcal{P}} FV(\boldsymbol{\rho}(P)) \subseteq \mathcal{P}$. In particular, $\boldsymbol{\rho}(P_i)$ is the natural closure of $\rho(P_i)$: $\boldsymbol{\rho}(P_i)$ implies the direct restriction $\rho(P_i)$ and all indirect restrictions $\rho(P_j)$, where P_j appears in the transitive closure of free variables in restrictions. (We assume that this closure is finite.)

The semantics of the Boolean connectives in the three-valued interpretation is the usual one augmented with the rule that any Boolean formula is $\perp$ if and only if a subformula is $\perp$. In particular, we have the following truth tables:

~	
$\perp$	$\perp$
0	1
1	0

&	$\perp$	0	1
$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	0	0
1	$\perp$	0	1

Now the three-valued semantics is defined as:

$$\begin{array}{rcl}
[\![\texttt{\~{}}\phi']\!]w & = & \texttt{\~{}}[\![\phi']\!]w \\
[\![\phi' \ \texttt{\&}\ \phi'']\!]w & = & [\![\phi']\!]w \ \texttt{\&}\ [\![\phi'']\!]w \\
[\![\texttt{ex2}\ P_i\ \texttt{where}\ \rho\colon \phi']\!]w & = & \begin{cases} 1 & \text{if } \exists M : [\![\phi']\!]w[P_i \mapsto M] = 1 \\ 0 & \text{if } \forall M : [\![\phi']\!]w[P_i \mapsto M] \neq 1 \text{ and } \exists M : [\![\phi']\!]w[P_i \mapsto M] = 0 \\ \perp & \text{if } \forall M : [\![\phi']\!]w[P_i \mapsto M] = \perp \end{cases} \\
[\![P_i\ \texttt{sub}\ P_j]\!]w & = & \begin{cases} 1 & \text{if } w \vDash P_i\ \texttt{sub}\ P_j \text{ and } [\![\boldsymbol{\rho}^*(P_i)\ \texttt{\&}\ \boldsymbol{\rho}^*(P_j)]\!]w = 1 \\ 0 & \text{if } w \nvDash P_i\ \texttt{sub}\ P_j \text{ and } [\![\boldsymbol{\rho}^*(P_i)\ \texttt{\&}\ \boldsymbol{\rho}^*(P_j)]\!]w = 1 \\ \perp & \text{if } [\![\boldsymbol{\rho}^*(P_i)\ \texttt{\&}\ \boldsymbol{\rho}^*(P_j)]\!]w \neq 1 \end{cases}
\end{array}$$

where we have only shown one kind of atomic formula, since the other ones are treated similarly: the meaning of an atomic formula is $\perp$ if the meaning of the conjunction of all restrictions of variables in the formula is not 1; otherwise the semantics is the standard one.

The automata-theoretic approach

As in the case of WS1S, we can show that the semantics can be represented by finite-state automata. So, we need to demonstrate automata A_ϕ that calculate $[\![\phi]\!]w$. Naturally, the states of these automata are characterized according to the three-valued domain, that is, as accepting (1), rejecting (0), or don't-care ($\perp$). We write $A(w)$ to denote the value calculated by A on word w; this value is the acceptance status of the state that A reaches when reading w. A basic observation is that any automaton A, by a simple relabeling of acceptance status, can be transformed into an automaton $\mathcal{R}A$ such that

$$\mathcal{R}A(w) = \begin{cases} 1 & \text{if } A(w) = 1 \\ \perp & \text{if } A(w) \neq 1 \end{cases}$$

Then, it is not hard to see that an atomic formula like $\phi = P_i$ `sub` P_j can be represented by the automaton

$$\mathcal{R}A_{\boldsymbol{\rho}^*(P_i)} \ \times\ \mathcal{R}A_{\boldsymbol{\rho}^*(P_j)} \ \times A,$$

where A is the classical automaton for P_i `sub` P_j as already presented and $\times$ is the automata product with product states characterized according to the three-value truth table for `&`. Note the use of the recursion in arguments $\boldsymbol{\rho}^*(P_i)$ and $\boldsymbol{\rho}^*(P_j)$.

It can be shown that the automata-theoretic calculations of the three inductive cases in the classical approach can be generalized to the three-valued semantics. In fact, only the case of existential quantification requires careful consideration.

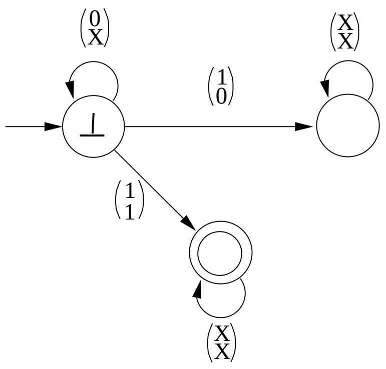
The automata being printed using the `-w` option (see 2.4) do not contain any "don't-care" states. At this final stage of the translation process, it is not relevant whether a state is categorized as "rejecting" or "don't-care". Before the automaton is printed, all don't-care states are hence converted to reject states, and the resulting automaton is minimized so that an absolute minimal-size automaton accepting the desired language is shown.

An example

Consider the atomic formula `p in P` in the full MONA syntax. If there are no restrictions introduced by `where` or `defaultwhere` clauses, then the automaton for `p in P` is formed as the product of

- an automaton that calculates a value that is either 1 or $\perp$ according to whether the singleton property holds for the `p`-track, and
- the basic automaton for `p sub P`, where `p` is regarded as a second-order variable.

This automaton looks like:



where we assume that the index of `p` is 1 and that of `P` is 2.

Requirements on restrictions

The ternary semantics will not provide meaningful results unless the following requirement holds for all formulas ϕ of the form `ex2` P_i `where` ρ : ϕ':

$$\models \texttt{ex2}\ P_i : \boldsymbol{\rho}^*(P_i)$$

That is: for any interpretation that defines the values of all free variables, except for P_i, in the closure $\boldsymbol{\rho}^*(P_i)$ of P_i, there is an interpretation of P_i that makes the closure hold. For example, $\phi =$ `ex2 P where ~P=P: P=P` does not satisfy this requirement, since the restriction is always false. Indeed, there is a problem here: the formula evaluates to $\perp$ under the semantics just given, not to 0 as one would expect of a false formula that contains no restricted free variables.

Equivalence of the binary and ternary semantics

It can be proven [28], that the classical semantics is equivalent to the ternary semantics for WS1S-R in the following sense

$$\begin{aligned} w \not\models \boldsymbol{\rho}^*(\phi) \quad &\Leftrightarrow \quad [\![\phi]\!]w = \perp \\ w \models \phi \ \texttt{\&}\ \boldsymbol{\rho}^*(\phi) \quad &\Leftrightarrow \quad [\![\phi]\!]w = 1 \\ w \models \texttt{\~{}}\phi \ \texttt{\&}\ \boldsymbol{\rho}^*(\phi) \quad &\Leftrightarrow \quad [\![\phi]\!]w = 0 \end{aligned}$$

5 WS2S and Guided Tree Automata

WS2S is the logic obtained by generalizing WS1S to be interpreted over the domain of elements generated from two successors instead of one. So, in WS1S a *string* defines an interpretation, whereas in WS2S a *binary tree* defines an interpretation. In WS1S, each first-order variable is interpreted as a natural number, but in WS2S each first-order variable is interpreted as a position in the infinite binary tree. WS2S is also decidable using automata.

MONA runs in either *linear mode* or *tree mode* corresponding to WS1S or WS2S respectively. The mode is chosen with the keywords `linear` or `tree` in the header. Linear mode is the default.

In WS2S, the `+1` successor notation is replaced with `.0` and `.1` representing the *left* and *right* successor, respectively. A string of `0`s and `1`s can be used as abbreviation for multiple applications of the successor operators, e.g. `p.011` means `((p.0).1).1`. The predecessor operator `-1` in WS1S has an WS2S counterpart named `^` ("up").

5.1 Guided Tree Automata

This section describes the automata used in the decision procedure for WS2S.

A *tree* t over the alphabet Σ, written $t \in T_\Sigma$, can be defined by the grammar

$$t \quad ::= \quad \epsilon \quad | \quad \alpha\texttt{<}t_1, t_2\texttt{>}$$

where ϵ denotes the empty tree, $\alpha \in \Sigma$ and t_1 and t_2 are the left and right subtrees of t respectively.

A *guide* $G = (D, \mu, d_0)$ is a top-down, deterministic tree automaton that does not look at the labeling. Its states will be used to designate state space names of bottom-up automata. More formally, G consists of

– D, a finite set of state space names,
– $\mu : D \to D \times D$, the guide function, and
– $d_0 \in D$, the initial state space name.

A *guided tree automaton* (GTA) M_G with guide G is now a set of bottom-up tree automata: $M_G = (\{Q_d\}_{d\in D}, \Sigma, \{\delta_d\}_{d\in D}, \{\overline{q}_d\}_{d\in D}, F)$ where

– $\{Q_d\}_{d\in D}$ is a family of disjoint finite sets, one set for each state space name,
– Σ is the alphabet,
– $\{\delta_d\}_{d\in D}$ is a family of transition functions, one for each state space name, such that if $\mu(d) = (d_1, d_2)$, then δ_d is of the form $\delta_d : (Q_{d_1} \times Q_{d_2}) \to (\Sigma \to Q_d)$,
– $\{\overline{q}_d\}_{d\in D}$ is the family of initial states, one for each state space name, such that $\overline{q}_d \in Q_d$ for each d, and
– $F \subseteq Q_{d_0}$ is the set of final states.

Given a tree t and a GTA M_G, we define whether M_G accepts t by a two-step process:

1. First, a state space is assigned to each node in t. Let $T_{(\Sigma,D)}$ denote the set of trees defined by the grammar

   $$\tilde{t} \quad ::= \quad d \quad | \quad (\alpha, d)\texttt{<}\tilde{t}_1, \tilde{t}_2\texttt{>}$$

   where $\alpha \in \Sigma$ and $d \in D$. The bottom-up tree automaton distinguishes between different empty subtrees—thus the need for the leaf of the form d above. The tree t can now be labeled with state spaces in a top-down style by applying the function $\hat{\mu} : T_\Sigma \times D \rightarrow T_{(\Sigma,D)}$ to (t, d_0), where $\hat{\mu}$ is defined by:

$\hat{\mu}(\epsilon, d) = d$
$\hat{\mu}(\alpha{<}t_1, t_2{>}, d) = (\alpha, d) <\hat{\mu}(t_1, d_1), \hat{\mu}(t_2, d_2)>$, where $\mu(d) = (d_1, d_2)$

Notice that the state spaces are assigned independently of the labels in t.

2. Next, each subtree of the resulting tree $\hat{\mu}(t, d_0)$ is assigned a state in a bottom-up style by the function $\hat{\delta} : T_{(\Sigma,D)} \rightarrow \bigcup_{d \in D} Q_d$ defined by:

   $\hat{\delta}(d) = \overline{q}_d$
   $\hat{\delta}((\alpha, d){<}\tilde{t}_1, \tilde{t}_2{>}) = \delta_d(\hat{\delta}(\tilde{t}_1), \hat{\delta}(\tilde{t}_2))(\alpha)$

The *language* recognized by M_G is the set of trees $t \in T_\Sigma$ such that $\hat{\delta} \circ \hat{\mu}(t, d_0) \in F$.
A GTA can be seen as an ordinary tree automaton, where the state space has been factorized according to the guide. A GTA with only one state space is thus just an ordinary tree automaton.

The reason for performing the factorization is to provide a means for avoiding state explosions. If one has certain knowledge or heuristics about locations of independent information in the infinite binary tree, a guide can be constructed exploiting these features by providing the positional knowledge. An example of this is shown in the following Section.

5.2 Specifying guides and universes

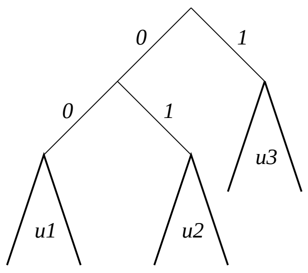
In MONA the guide is defined in the header with the `guide` construct. As an example

```
guide a->(b,c), b->(b,b), c->(c,c);
```

defines a guide with state-space names `a`, `b`, `c`, the guide function which maps `a` to (`b`, `c`) etc. and with `a` (the first name in the list) as the initial state space name.

A *universe* u is a subtree of the infinite binary tree satisfying the property that if the position p is in u then both the left and the right successors of p are also in u.
A universe can thus be identified by a string of 0s and 1s representing the sequence of left and right successors respectively which constitute the path from the root of the infinite binary tree to the root of the universe. A universe u has associated to it the set of state spaces D_u reachable from the root of u: a state space d is in D_u if and only if there is some tree containing a node in u that is assigned state space d.

We assume that the infinite binary tree is covered with disjoint universes in the following sense: along every infinite path starting from the root, exactly one universe is eventually reached. This scenario can be depicted by the following example illustrating a division of the infinite binary tree into three universes, `u1`, `u2` and `u3` (identified by the strings 00, 01 and 1 respectively) and some nodes in the top that are not part of any universe.



MONA allows the user to define the universes above by the declaration:

```
universe u1:00, u2:01, u3:1;
```

Boolean variables are assigned a special state space always located in the root of the infinite binary tree. This state space may not belong to any universe.

The user can declare variables to range only over values in certain universes. For instance,

```
var1 [u1] x;
var2 [u2,u3] Y,Z;
```

declares that `x` is interpreted as a node in `u1` and that `Y` and `Z` are interpreted as subsets of the union of universes `u2` and `u3`.

Guides and universes can be specified by one of three methods:

1. Neither a guide nor any universes are specified. MONA then generates two universes (of which one is a "dummy") and a trivial guide. (Two universes are needed because the encoding of Boolean variables needs a special state space located in the root of the infinite binary tree). The automata being constructed are essentially traditional tree automata.
2. A number of universes without positions are specified but no guide. MONA then generates a guide as a balanced tree and places the universes at the leaves of this tree.
3. A guide and a number of universes with positions are specified completely by the user as described above.

Other tree-mode specific constructs

The root of a universe u can be expressed using the first-order construct

`root(`u`)`

If the guide and universes are specified according to method 1, then the expression `root` can be used to denote the root of the implicitly constructed (non-dummy) universe.

The `root(`u`)` term can be used to express first-order constants (i.e. position constants), e.g. `root(u2).01101`.

Another way of accessing the roots of universes is using the formula

`root`$(t, [u_1, u_2, \ldots, u_n])$

```
tree;

var2 A,B;

ex1 p1,p2,p3,p4,p5:
  p1<p2 & p2<p3 & p3<p4 & p4<p5 &
  A = {p1,p2,p3,p4,p5};

ex1 p1,p2,p3,p4,p5,p6,p7:
  p1<p2 & p2<p3 & p3<p4 & p4<p5 & p5<p6 & p6<p7 &
  B = {p1,p2,p3,p4,p5,p6,p7};
```

Figure 12: `ab1.mona`

where t is a first-order term and each u_i is a universe name. It is true if and only if t is the root of one of the universes $u_1, u_2, \ldots, u_n$.

Where a universe name is required, a variable name can be used instead. The variable name then denotes the set of universes over which the variable ranges.

Example

Assume we want a GTA to accept the set of trees which have a branch with exactly 5 `a`-labels and a branch with exactly 7 `b`-labels. With ordinary tree automata (e.g. using method 1 for describing universes), the automaton could be encoded as shown in Figure 12. Since the resulting automaton has to "remember" both how many `a`s (0 to 5) and how many `b`s (0 to 7) it encounters, it has approximately $6 \cdot 8$ states.

Now assume that we happen to know that `a`s occur only in the left part of the infinite binary tree, that is, any node labeled `a` is below `root.0`. Similarly, any node labeled `b` is below `root.1`. Utilizing this location independence, we can construct a guide (with three state spaces) and two universes and restrict the various variables to the appropriate universes as shown in Figure 13.

In effect, we now get one automaton counting `a`s and one counting `b`'s, so the total number of states is now around $6 + 8$. In general, exploiting positional knowledge using the guide/universe technique can give an exponential decrease in state-space size. Experience shows that positional knowledge often arises naturally [26].

5.3 Tree mode output format

In the following, the format of output of automata, satisfying examples and counter-examples, etc. for WS2S mode is described.

WS2S formula analysis

Executing

```
mona -c ab2.mona
```

```
tree;

guide d0->(a,b), a->(a,a), b->(b,b);
universe ua:0, ub:1;

var2 [ua] A;
var2 [ub] B;

ex1 [ua] p1,p2,p3,p4,p5:
  p1<p2 & p2<p3 & p3<p4 & p4<p5 &
  A = {p1,p2,p3,p4,p5};

ex1 [ub] p1,p2,p3,p4,p5,p6,p7:
  p1<p2 & p2<p3 & p3<p4 & p4<p5 & p5<p6 & p6<p7 &
  B = {p1,p2,p3,p4,p5,p6,p7};
```

Figure 13: `ab2.mona`

generates (amongst other output) the following satisfying example, which illustrates the WS2S output format:

```
A satisfying example (for assertion & main) is:
Booleans:
XX
Universe ua:
(1X,(1X,(1X,(1X,(),(1X,(),())),()),()),())
Universe ub:
(X1,(X1,(X1,(X1,(X1,(X1,(X1,(),()),()),()),()),()),()),())
```

First, the values from the Boolean state space is written in a string. None of the free variables in this example are Boolean, so the string just contains `X`s. Next, for each universe, a tree is printed. An empty subtree is written as `()`, and a node of the form α<t_1, t_2> (see p. 36) is written as `(`α, t_1, t_2`)`. As one can see, `A` is assigned a set with 5 elements and `B` is assigned a set with 7 elements as requested by the MONA program.
It also prints the following summary about the conjunctive automaton (unless `-w` is also specified):

```
Conjunctive automaton:
State space 0: 2 states, 2 BDD nodes
State space 1: 7 states, 18 BDD nodes
State space 2: 9 states, 22 BDD nodes
Total: 18 states, 42 BDD nodes
```

The satisfying examples and counter-examples can be shown graphically using the graphviz tool dot (see 3.2). For instance, executing

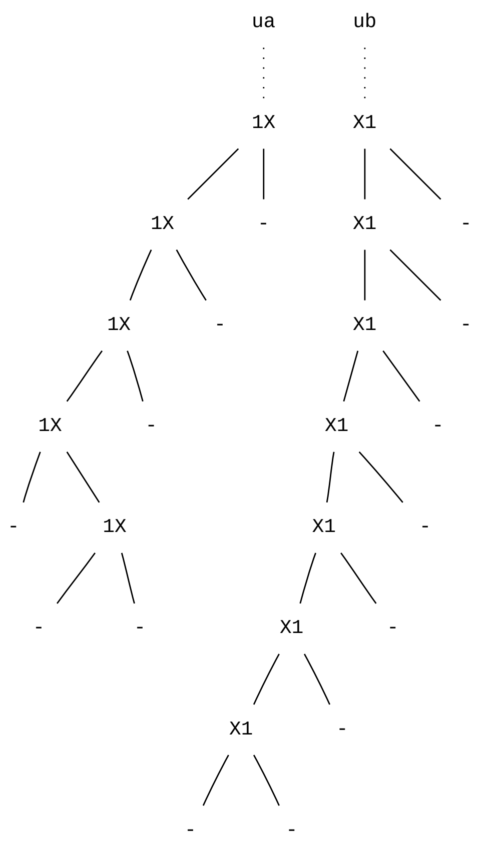
```
mona -gs ab2.mona > ab2-sat_ex.dot
dot -Tps ab2-sat_ex.dot -o ab2-sat_ex.ps
```

generates the following output in the file `ab2-sat_ex.ps`, which shows the satisfying example from before:

```
SATISFYING EXAMPLE

Free variables are: A, B

Booleans: XX
```



Outputting minimal GTAs

Executing

```
mona -w ab2.mona
```

first prints

```
Guide:
0 -> (1,2)
1 -> (1,1)
2 -> (2,2)
```

which shows how numbers are assigned to the state spaces. The number 0 is always the number of the Boolean state space, i.e. the state space in the root of the infinite binary tree. It then prints:

```
GTA for formula with free variables: A B
Accepting states: 1
Rejecting states: 0

State space 0 (size 2):
Initial state: 0
Transitions:
(0,0,XX) -> 0
(0,1,XX) -> 0
(0,2,XX) -> 0
  ...
(6,7,XX) -> 0
(6,8,XX) -> 0

State space 1 (size 7):
Initial state: 6
Transitions:
(0,0,XX) -> 2
  ...
(6,6,0X) -> 6
(6,6,1X) -> 5

State space 2 (size 9):
Initial state: 0
Transitions:
(0,0,X0) -> 0
  ...
(8,7,XX) -> 4
(8,8,XX) -> 4

Number of transitions: 18
```

The format resembles the one described in Section 2.4. As an example: the very last transition shown here defines that $\delta_2(8,8)(\mathtt{XX}) = 4$ in the notation used on p. 36.

Translation statistics

The format of statistics output (see 3.1) in tree mode is illustrated with the following piece of output:

```
    ...
Product & 'ab2.mona' line 16
    p1<p2 & p2<p3 & p3<p4 & p4<p5 & p5<p6 & p6<p7;
          ^
  (3,3; 1,1; 225,5425)x(3,3; 1,1; 6,16) -> (7,7; 1,1; 294,3702)
  Minimizing (7,7; 1,1; 294,3702) -> (3,3; 1,1; 193,4202)
    ...
```

It shows a product operation which results in a GTA with 3 states and 3 BDD nodes in state space 0, 1 state and 1 BDD node in state space 1, and 193 states and 4202 BDD nodes in state space 2.

5.4 Emulating Monadic Second-order Logic on Trees

Just as WS2S is a generalization of WS1S, *Monadic Second-order Logic on Trees* (M2L-Tree) is a generalisation of M2L-Str, which was briefly described in Section 3.5. M2L-Tree can be emulated by writing

```
m2l-tree;
```

which is the same as writing

```
tree;
var2 $ where all1 p: (p in $) => ((p^ in $) | (p^=p));
lastpos $;
defaultwhere1(p) = p in $;
defaultwhere2(P) = P sub $;
```

The idea is the same as for M2L-Str: A special variable `$` is used as a “skeleton” to which all other variables are restricted.

5.5 External GTAs

Guided Tree Automata can be exported and imported just as ordinary DFAs can (see 3.7).

Format of external GTA files

Executing MONA on the following program

```
tree;
var2 P,Q;
export("test.gta", P sub Q);
```

generates the file `test.gta` which illustrates the format of a “`.gta`”-file:

```
MONA GTA
number of variables: 2
variables: P Q
orders: 2 2
state spaces: 3
state space sizes: 2 2 1
final: 1 -1
guide:
1 2
1 1
2 2

state space 0:
initial state: 0
bdd nodes: 2
behaviour:
0
1
bdd:
-1 0 0
-1 1 0

...
end
```

Using GTAs in other applications

Similar to the DFA library described in Section 3.7, a small GTA library is available in the MONA package (as `Lib/gtalib.[ch]`) so that the GTAs generated by MONA can be used in other applications. The following functions are in the library:

`mgGta *mgLoad(char *filename)` - loads a GTA from a file.

`int mgStore(mgGta *gta, char *filename)` - stores a GTA in a file.

`void mgFree(mgGta *gta)` - deallocates the memory used by a GTA.

`mgState mgDelta(mgGta *gta, mgId ss, mgState left, mgState right, mA *a)` - represents the transition function of a GTA (see 5.1). The arguments are a GTA, a state space identifier, a state from the left state space, a state from the right state space, and an alphabet symbol.

`void mgAssign(mgGta *gta, mgTreeNode *t, mgId id)` - assigns state space identifiers and states to the nodes of a labeled binary tree.

`int mgCheck(mgGta *gta, mgTreeNode *t)` - takes a GTA and a labeled binary tree as arguments and checks whether the tree is accepted by the GTA or not.

See `gtalib.h` for further descriptions.

6 Plans for future versions

This section describes the major ideas and plans for future work on the MONA project. Users of the MONA tool are encouraged to send other ideas to us.

6.1 Specifying BDD variable ordering

The indices assigned to the variables in a formula are used in the internal BDD representation to define the ordering of the BDD nodes. It is a well-known fact (see [8]) that this ordering has a strong influence on the number of BDD nodes required. When the number of nodes increases, the time spent on automaton operations increases correspondingly.

As previously mentioned, in the current version of MONA, indices are assigned to variables in order of occurrence. Consider the following MONA program:

```
var2 P1,Q1,P2,Q2,P3,Q3,P4,Q4,P5,Q5;
P1=Q1 & P2=Q2 & P3=Q3 & P4=Q4 & P5=Q5;
```

The corresponding automaton generated by MONA has 3 states and 17 BDD nodes are used to represent the transition function. If we simply change the order of the declarations of the variables as shown below, 95 BDD nodes are required to represent essentially the same automaton.

```
var2 P1,P2,P3,P4,P5,Q1,Q2,Q3,Q4,Q5;
P1=Q1 & P2=Q2 & P3=Q3 & P4=Q4 & P5=Q5;
```

By adding more Pi-Qi pairs, the number of BDD nodes grows linearly in the first program, and exponentially in the second. This is thus an example of an exponential difference caused by the ordering of the indices.

Another well-known fact about BDDs is that deciding the optimal ordering is NP-complete, so searching for optimal orderings is not feasible. Instead we want the MONA programmer to have a more succinct and pragmatic method for choosing the orderings manually based on heuristics about the problem being represented by the MONA program.

Preliminary experiments have shown that useful heuristics are of the form "P should have a higher index than Q" or "the indices of P and Q should be close to each other". Our plan is to provide a syntactical means (called "pragmas") for expressing such rules concisely and make MONA generate an ordering satisfying as many of the rules as possible.

For the curious reader we have included the preliminary syntax for specifying "pragmas" in the MONA syntax in appendix A.

6.2 Encoding of Boolean variables in tree mode

The Boolean variable encoding in trees often leads to exponential explosions, since the tree automata can work under no assumptions about what the Boolean values are in the top. This is in contrast to the situation for strings, where the first thing that the automaton encounters is the assignment of values to the Booleans. In a future version of MONA, tree mode will use the same idea.

6.3 Symbolic reductions

Before the actual translation of formulas into automata takes place, experiments have shown that it is useful to perform symbolic reductions on the formulas. The reductions on a formula are based on an analysis, which attempts to deduce validity, unsatisfiability, implications and equivalences of the subformulas. The formula is subsequently simplified using some rewriting rules based on the information obtained by the analysis.

Since the logic is decidable it is possible to obtain precise answers from the analysis, but as the decision procedure has a non-elementary lower bound, precise answers are not what the analysis should aim for. Instead it should be an approximating, conservative analysis with a low complexity based on heuristics.

As it is the case for optimizers in traditional compilers, the largest need for optimizations is on machine-generated code. The FIDO tool (see [26]) is an example of a tool which automatically generates MONA code. Due to the compositional design of FIDO, the processing by MONA of the code generated by FIDO would benefit greatly by symbolic reductions.

Especially one kind of reductions (which resembles *let*-reduction) is useful: Assume a subformula has the form $\exists P : \phi$ and the formula analysis has established that $\phi \Rightarrow P = T$ where T is some term satisfying $FV(T) \subseteq FV(\phi)$. Then it is sound to replace $\exists P : \phi$ by $\phi[T/P]$ (which means: ϕ where each occurrence of P is substituted by T). The resulting formula has one less quantifier, and since quantifiers are the main source of state-space explosions (due to the automaton determinization, see 4.1), reductions like this can be very useful.

Preliminary experiments show drastic improvements in time and space consumption on machine-generated MONA code (generated by the FIDO tool). A high-priority plan of ours is to make and implement a system of analysis and rewriting rules.

A Syntax

The grammar below describes the full MONA syntax in a BNF-like notation with the following meta-syntax:

$X \mid Y$	either X or Y,
$[\ X\ ]^?$	an optional X,
$[\ X\ ]^*$	zero or more occurrences of X,
$[\ X\ ]^\odot$	zero or more occurrences of X separated by commas,
$[\ X\ ]^+$	one or more occurrences of X, and
$[\ X\ ]^\oplus$	one or more occurrences of X separated by commas.

A.1 MONA grammar

MONA program

$program$ ::= $[\ header$ `;` $]^*$ $[\ declaration$ `;` $]^+$

Declarations

$header$	::=	`linear`
	|	`tree`
	|	`m2l-str`
	|	`m2l-tree`
	|	`guide` $[\ guidearg\ ]^\oplus$
	|	`universe` $[\ univarg\ ]^\oplus$
	|	`include "`$filename$`"`

$declaration$	::=	ϕ
	|	`assert` ϕ
	|	`const` c `=` I
	|	`defaultwhere1 (` p `) =` ϕ
	|	`defaultwhere2 (` P `) =` ϕ
	|	`var0` $[\ \alpha\ ]^\oplus$ $[\ pragmas\ ]^?$
	|	`var1` $[\ univs\ ]^?$ $[\ varwhere1\ ]^\oplus$ $[\ pragmas\ ]^?$
	|	`var2` $[\ univs\ ]^?$ $[\ varwhere2\ ]^\oplus$ $[\ pragmas\ ]^?$
	|	`macro` $name$ $[\ params\ ]^?$ `=` ϕ
	|	`pred` $name$ $[\ params\ ]^?$ `=` ϕ

Formulas

$$\begin{array}{rclcl}
\phi & ::= & \texttt{true} & | & \texttt{false} \\
 & | & \alpha & | & \mathit{name}\ \texttt{(}\ [\ \mathit{exp}\ ]^{\odot}\ \texttt{)} \\
 & | & \mathit{name} & | & t_1\ \texttt{=}\ t_2 \\
 & | & t_1\ \texttt{\textasciitilde=}\ t_2 & | & t_1\ \texttt{<}\ t_2 \\
 & | & t_1\ \texttt{>}\ t_2 & | & t_1\ \texttt{<=}\ t_2 \\
 & | & t_1\ \texttt{>=}\ t_2 & | & T_1\ \texttt{=}\ T_2 \\
 & | & T_1\ \texttt{\textasciitilde=}\ T_2 & | & T_1\ \texttt{sub}\ T_2 \\
 & | & t\ \texttt{in}\ T & | & t\ \texttt{notin}\ T \\
 & | & \texttt{empty (}T\texttt{)} & | & \texttt{\textasciitilde}\ \phi \\
 & | & \phi_1\ \texttt{\&}\ \phi_2 & | & \phi_1\ \texttt{|}\ \phi_2 \\
 & | & \phi_1\ \texttt{=>}\ \phi_2 & | & \phi_1\ \texttt{<=>}\ \phi_2 \\
 & | & \texttt{(}\ \phi\ \texttt{)} & | & \texttt{export ("}\mathit{filename}\texttt{"}\ \phi\ \texttt{)} \\
 & | & \multicolumn{3}{l}{\texttt{import ("}\mathit{filename}\texttt{"}\ [\ \texttt{,}\ v\ \texttt{->}\ \mathit{var}\ ]^{*}\ \texttt{)}} \\
 & | & \multicolumn{3}{l}{\texttt{ex0}\ [\ \alpha\ ]^{\oplus}\ [\ \mathit{pragmas}\ ]^{?}\ \texttt{:}\ \phi} \\
 & | & \multicolumn{3}{l}{\texttt{all0}\ [\ \alpha\ ]^{\oplus}\ [\ \mathit{pragmas}\ ]^{?}\ \texttt{:}\ \phi} \\
 & | & \multicolumn{3}{l}{\texttt{ex1}\ [\ \mathit{univs}\ ]^{?}\ [\ \mathit{varwhere1}\ ]^{\oplus}\ [\ \mathit{pragmas}\ ]^{?}\ [\ \texttt{restr}\ T\ ]^{?}\ \texttt{:}\ \phi} \\
 & | & \multicolumn{3}{l}{\texttt{all1}\ [\ \mathit{univs}\ ]^{?}\ [\ \mathit{varwhere1}\ ]^{\oplus}\ [\ \mathit{pragmas}\ ]^{?}\ [\ \texttt{restr}\ T\ ]^{?}\ \texttt{:}\ \phi} \\
 & | & \multicolumn{3}{l}{\texttt{ex2}\ [\ \mathit{univs}\ ]^{?}\ [\ \mathit{varwhere2}\ ]^{\oplus}\ [\ \mathit{pragmas}\ ]^{?}\ [\ \texttt{restr}\ T\ ]^{?}\ \texttt{:}\ \phi} \\
 & | & \multicolumn{3}{l}{\texttt{all2}\ [\ \mathit{univs}\ ]^{?}\ [\ \mathit{varwhere2}\ ]^{\oplus}\ [\ \mathit{pragmas}\ ]^{?}\ [\ \texttt{restr}\ T\ ]^{?}\ \texttt{:}\ \phi} \\
 & | & \multicolumn{3}{l}{\texttt{let0}\ [\ \alpha = \phi\ ]^{\oplus}\ [\ \mathit{pragmas}\ ]^{?}\ \texttt{in}\ \phi} \\
 & | & \multicolumn{3}{l}{\texttt{let1}\ [\ p = t\ ]^{\oplus}\ [\ \mathit{pragmas}\ ]^{?}\ \texttt{in}\ \phi} \\
 & | & \multicolumn{3}{l}{\texttt{let2}\ [\ P = T\ ]^{\oplus}\ [\ \mathit{pragmas}\ ]^{?}\ \texttt{in}\ \phi}
\end{array}$$

The following formula is allowed in linear mode:

$$\phi \quad ::= \quad \texttt{prefix (}\phi\texttt{)}$$

The following formula is allowed in tree mode:

$$\phi \quad ::= \quad \texttt{root (}t\ \texttt{,}\ \mathit{univs}\texttt{)}$$

First-order terms in linear mode

$$\begin{array}{rclclclclclclclc}
t & ::= & p & | & \texttt{(}\ t\ \texttt{)} & | & I & | & t\ \texttt{+}\ I & | & t\ \texttt{-}\ I & | & t_1\ \texttt{+}\ I\ \texttt{\%}\ t_2 & | & t_1\ \texttt{-}\ I\ \texttt{\%}\ t_2 \\
 & | & \texttt{max}\ T & | & \texttt{min}\ T
\end{array}$$

Second-order terms in linear mode

$$\begin{array}{rclclclclcl}
T & ::= & \{\ [\ \mathit{elems}\ ]^{\odot}\ \} & | & \texttt{(}\ T\ \texttt{)} & | & P & | & \texttt{empty} \\
 & | & T_1\ \texttt{union}\ T_2 & | & T_1\ \texttt{inter}\ T_2 & | & T_1 \setminus T_2 & | & T\ \texttt{+}\ I & | & T\ \texttt{-}\ I
\end{array}$$

First-order terms in tree mode

$$t \quad ::= \quad p \quad | \quad \texttt{(}\ t\ \texttt{)} \quad | \quad t\texttt{.}\mathit{succ} \quad | \quad t\texttt{\textasciicircum} \quad | \quad \texttt{root}\ [\ \texttt{(}\ u\ \texttt{)}\ ]^{?}$$

Second-order terms in tree mode

$$\begin{array}{rclclclclcl}
T & ::= & \{\ [\ \mathit{elems}\ ]^{\odot}\ \} & | & \texttt{(}\ T\ \texttt{)} & | & P \\
 & | & T_1\ \texttt{union}\ T_2 & | & T_1\ \texttt{inter}\ T_2 & | & T_1 \setminus T_2 & | & T.\mathit{succ} & | & T\texttt{\textasciicircum}
\end{array}$$

Other

$elems$ ::= t | t_1 `,` ... `,` t_2

$univs$ ::= `[` [u]$^{\oplus}$ `]`

$succ$ ::= [0 | 1]$^{+}$

exp ::= t | T | ϕ

$params$ ::= `(` [par]$^{\oplus}$ `)`
par ::= `var0` [α]$^{\oplus}$
| `var1` [$varwhere1$]$^{\oplus}$
| `var2` [$varwhere2$]$^{\oplus}$
| `universe` u

var ::= α | p | P

$pragmas$ ::= { [$pragma$]$^{\oplus}$ }
$pragma$ ::= [i `:`]$^{?}$ var wop var
wop ::= op | op i
op ::= $<$ | $>$ | $=$

$guidearg$ ::= σ_1 `->` `(` σ_2 `,` σ_3 `)`

$univarg$ ::= u [`:` $succ$]$^{?}$

$varwhere1$ ::= p [`where` ϕ]$^{?}$
$varwhere2$ ::= P [`where` ϕ]$^{?}$

Restrictions and comments

- α is a name of a Boolean variable, which we also regard as a zeroth-order variable; p, P are names of first and second-order variables, respectively. A name can be any string of letters, digits, underscores, dollar symbols, and single quotes.

  I is an constant integer expression, e.g. $(2+4*k/(7-c))$, where k and c are constants declared by the `const` declaration.

  i denotes an integer.

  c, $name$, σ and u denote the name of a constant, a predicate or macro, a state space, or a name of universe, respectively.

  v denotes the name of a variable in an external file.

- Anything related to guides and universes only makes sense in tree mode. There can be at most one guide header and one universe header.

- In general, all names must be defined before they are used, i.e., "forward references" are not allowed.

- If `defaultwhere` declarations are used, they must be placed *before* all predicate and macro declarations.

It is possible to insert comments in the formulas. This is done by putting a `#` in a line whereby MONA treats the rest of the line as a comment.

A.2 Precedence and associativity

The table below shows the precedence and associativity of the MONA operators. If for instance the operator op_1 has higher precedence (lower precedence number) than the operator op_2, then the expression $E_1\ op_1\ E_2\ op_2\ E_3$ is interpreted as $(E_1\ op_1\ E_2)\ op_2\ E_3$. If the precedences are equal, then the interpretation is decided by the associativity, e.g. if *op* is right-associative then $E_1\ op\ E_2\ op\ E_3$ is interpreted as $E_1\ op\ (E_2\ op\ E_3)$. The default rules can be overridden with parentheses.

Precedence	Operator	Associativity
1	`.` `^`	non-associative
2	`*` `/` `%`	left-associative
3	`+` `-`	left-associative
4	`\`	left-associative
5	`inter`	left-associative
6	`union`	left-associative
7	`max` `min`	non-associative
8	`=` `~=` `>` `>=` `<` `<=`	non-associative
9	`in` `notin` `sub`	non-associative
10	`~`	non-associative
11	`&`	left-associative
12	`|`	left-associative
13	`=>`	right-associative
14	`<=>`	right-associative
15	`:`	non-associative

B Usage

The usage of the MONA-tool is:

```
mona [options] <filename>
```

The options are:

`-c` Analyzes resulting automaton, prints "`valid`", "`unsatisfiable`" or a satisfying example and a counter-example. *See Section 2.3*.

`-w` Outputs a description of the resulting automaton. *See Section 2.4*.

`-t` Prints the time spent computing. *See Section 3.1*.

`-s` Prints statistics for each automaton operation. *See Section 3.1*.

`-p` Prints progress information. *See Section 3.1*.

`-i` After each automaton operation, the resulting automaton is printed. *See Section 3.1*.

`-d` Dumps the abstract-syntax tree, symbol-table contents, the guide (in tree mode), and the code DAG. *See Section 3.1*.

`-e` Enables separate compilation. The directory designated by the environment variable `MONALIB` is used as base for the automaton library. If `MONALIB` is not set, the current working directory will be used instead. *See Section 3.3*.

`-o`N Enables optimizations of the formula. (Not implemented yet.) *See Section 6.3*.

`-r`N Sets the automaton reuse ("DAGification") degree to N. Default value is 100. *See Section 3.9*.

`-f` Forces tree-mode output style of satisfying examples and counter-examples.

`-gw` Outputs resulting automaton in graphviz format if in linear mode. *See Section 3.2*.

`-gs` Outputs satisfying example tree in graphviz format if in tree mode. *See Section 5.3*.

`-gc` Outputs counter-example tree in graphviz format if in tree mode. *See Section 5.3*.

`-gd` Outputs DAG in graphviz format. *See Section 3.9*.

The default options are: `-c -p`.

C The MONA BDD package

The MONA BDD package has been written for maximum speed of the BDD operations needed for the BDD-represented automata in the MONA tool. The package achieves a factor 6 speed-up over David Long's BDD package for the specialized task of MONA calculations, but at the expense of a storage model that may make it hard to use correctly.

In the MONA tool, an automaton with n states is represented by a shared BDD with n roots and n leaves. It is assumed that when an automaton is calculated, there is not a large proportion of it that can be retrieved as part of an already calculated automaton. This is in contrast to usual BDD packages, which use facts such as $1 \wedge \phi = \phi$ to calculate certain, specialized products in unit time by pointing to an already calculated subresult. This strategy hinges on using a global, hashed node space. In the world of automata, it is computationally expensive to identify isomorphic subgraphs, in contrast to the case of BDDs, where a node can be hashed relative to its successors in a well-founded manner.

Thus in the MONA BDD package, the shared BDD of an automaton is represented in a node space unique to the automaton. Such a node space is administered through a BDD manager of type `bdd_manager`.

The `bdd_manager`

A BDD manager keeps the BDD nodes in an array `node_table`. Thus nodes are not individually allocated, in contrast to conventional BDD packages. A `bdd_ptr` is an offset into this table. The first used offset is `BDD_NUMBER_OF_BINS` (2). The table consists of a hashed part, whose size `table_size` is $2^{\texttt{table\_log\_size}}$, where `table_log_size` is a natural number. In addition, there is an overflow area following the hashed part. The overflow area is enlarged in increments of `table_overflow_increment`. The field `table_total_size` is the number of BDD nodes in the allocated table. Initially, this number is `BDD_NUMBER_OF_BINS` + `table_size` + `table_overflow_increment`. The maximum allowed value of `table_total_size` is 2^{24}, roughly 16 million.

The BDD manager also keeps a result cache, which is a table of previously computed results used for binary apply and project operations. It is pointed to by `cache`. This pointer is `null` if the cache has not been allocated. The cache is hashed with an overflow area incremented in steps of `cache_overflow_increment`.

Various statistics about the number of lookups, insertions, and collisions in the node table and the result cache are maintained by the BDD manager.

BDD nodes

A BDD node is described in a structure data type name `bdd_record`. The left and right successors are each described as a `bdd_ptr` packed into three bytes. The node index (name of variable) is a two byte unsigned integer. Thus the maximum allowed index is 254, since the value 255 encodes a leaf. The successors and the index are packed into two words (each word is four bytes) named `lri[2]`. The `bdd_record` also contains a `next` field, which holds a `bdd_ptr` to an overflow list, and a `mark` field used by the unary apply routine and by other routines. A BDD node occupies four words or 16 bytes. This small size should enable two consecutive nodes to be loaded into a CPU cache line at a time. Consequently, the hashing scheme uses two bins per bucket, that is, `BDD_NUMBER_OF_BINS` = 2. Note that a BDD table consists of at most 2^{28} bytes or 256Mb.

Since BDD nodes sit in the hashed node table, BDD pointers are volatile data: when the node table is doubled, all BDD pointers in existence become invalid.

For this reason, a BDD manager provides an alternative way of describing the results of BDD operations. The manager maintains a dynamic array of BDD pointers `bdd_roots`, which are automatically updated when the node table is doubled. An offset into this array is called a `bdd_handle`. The BDD package guarantees that the pointer described by a handle always denotes the same BDD node. The apply operations augment automatically the `bdd_roots` array to contain the result of the operation. Some of the basic operations do not use the `bdd_roots` list, but can be given a user defined list whose pointers are updated in the case that the table is doubled. Such lists are declared and manipulated using the `SEQUENTIAL_LIST` macros, which enable lists with pop and push operations to be efficiently implemented as dynamically allocated arrays.

Basic operations that allow updating of lists also can be given a pointer of type `void (*update_fn) (unsigned (*new_place) (unsigned node))`. The denoted function takes as argument a function `new_place`. The basic operation calls `update_fn` when the table is doubled and it supplies the function `new_place`, which specifies the new pointer value of a BDD node given the old value.

Manipulation of `bdd_roots`

The macro `BDD_ADD_ROOT(`*`bddm`*`, `*`p`*`)` adds a BDD pointer *p* to the `bdd_roots` list of the BDD manager *bddm*. The macro `BDD_LAST_HANDLE(`*`bddm`*`)` can be used after a `BDD_ADD_ROOT(`*`bddm`*`, `*`p`*`)` application to get the handle of the `bdd_roots` list where the BDD pointer to the node currently designated as *p* is stored. For convenience, `BDD_ADD_ROOT_SET_HANDLE(`*`bddm`*`, `*`p`*`, `*`h`*`)` combines the two preceding operations and assigns the variable *h* of type `bdd_handle` the value of the last handle. The macro `BDD_ROOT(`*`bddm`*`, `*`handle`*`)` looks at the BDD pointer corresponding to *handle*.

Sequential mode

It is possible to use the BDD manager in a *sequential* mode, where nodes are not hashed. This is useful when it is known that any node inserted is a new one. Such a situation may occur when a BDD is read from a file or when the apply (product) of two BDDs is performed with a leaf function that form pairs. In these cases, BDD nodes can be inserted sequentially. When table is doubled, nodes do not change position; thus for sequential insertions, `bdd_ptr` is not volatile.

Sequential and hashed modes cannot be combined.

Basic operations

The *find_node* operation locates a BDD node if it already is known to exist (hashed mode only) or creates a new node. In hashed mode, there are two variations of the *find_node* operation: both add the current value of the BDD pointer to the `bdd_roots` list, and they differ only in whether this pointer is returned as a result or the handle is returned. There is also a more primitive version that as parameters take a list of `bdd_ptr`s and an `update_fn` as discussed above.

The apply operations

The unary apply operation has the following declaration:

```
bdd_ptr bdd_apply1(bdd_manager *bddm, bdd_ptr p,
                   bdd_manager *bbdm_r,
                   unsigned (*apply1_leaf_function)(unsigned value));
```

Here, **bddm* is the manager that holds the BDD on which the apply operation is carried out. We call this manager the *source manager*. The operation is initiated in the node *p* of the source table. The result is written into the shared BDD administered by **bbdm_r*, the *result manager*, and a BDD pointer, denoting the resulting BDD, in **bbdm_r* is returned; as a side-effect, this node is added to `bdd_roots` of the result manager. Thus a handle to the result can be obtained by using `BDD_LAST_HANDLE(bddm_r)` right after the apply operation.

A unary apply operation does not use a result cache. Instead, the value of the apply operation on a node is stored in the node itself. The `mark` field holds this information. Therefore all such fields must be initialized before the apply operation can be performed. This initialization is carried out by a call of

```
void bdd_prepare_apply1(bdd_manager *bddm);
```

In a sequence of unary apply operations with the same leaf function, it is not necessary to reinitialize the manager between operations.

The case that *bddm==bbdm_r* is allowed. If, on the other hand, the managers are different, then a doubling of the result table does not invalidate BDD pointers in the source table.

The binary apply operation, `apply2`, and the project operation, `bdd_project`, are similar to `apply1` with one important difference: a result cache is used to store the results of earlier apply (or project) operations, and this cache is administered by the result manager. Therefore, before any such operation is initiated, a cache must have been allocated for the result manager. Also, it is important to kill and recreate the cache whenever a new apply or project operation is carried out. Only if the new operation is identical (same leaf function or same index that is projected on) may the cache be reused. When the node table is doubled, the result cache is also doubled. The BDD package allows two cache doubling policies: either the new cache can be erased, or the BDD pointers in the cache may be rehashed. Experiments indicate that the former policy is the faster one.

A complicated example

In Figures 14 and 15, a simple application of the BDD package is shown, including the gathering of very detailed statistics.

The example in Figure 16 shows how to use the hashed version of find-node in a setting where the program stores BDD pointers in global and local variables.

```
/* Small dummy example of the use of the bdd-package */
#include "bdd.h"
#include <stdio.h>
#include <assert.h>

unsigned and(unsigned a, unsigned b) {
  if (a && b)
    return 1;
  else
    return 0;
};

unsigned not(unsigned a) {
  if (a)
    return 0;
  else
    return 1;
};

void print_bdd(bdd_manager *bddm, bdd_ptr b) {
  unsigned index;

  if (bdd_is_leaf(bddm, b)) {
    printf("(leafvalue: %d)", bdd_leaf_value(bddm, b));
  }
  else {
    index=bdd_ifindex(bddm,b);
    printf("(var %d = 1: ", index);
    print_bdd(bddm, bdd_then(bddm,b));
    printf(")");
    printf("(var %d = 0: ", index);
    print_bdd(bddm, bdd_else(bddm,b));
    printf(")");
  };
}

void main() {
  bdd_manager *bddm, *bddm1;
  bdd_ptr zero, one, and_2_7, nand_2_7;
  bdd_handle handle, var2, var7;

  bdd_init(); /* needed since we're using statistics */

  bddm = bdd_new_manager(100,50);
  /* get a BDD pointer to a node that is the leaf with value 0 */
  zero = bdd_find_leaf_hashed_add_root(bddm, 0);
  /* and a leaf with value 1 */
  one =  bdd_find_leaf_hashed_add_root(bddm, 1);
  /* note already at this point "zero" could have been invalidated if
     the table doubled, but we know that there is room for a 100
     nodes---anyway, this is really very bad style, so we go on in
     a more appropriate manner */
```

Figure 14: Simple use of package

```
 /* "then" is "l" ("left"), "else" is "r" ("right") */
 var2 = bdd_handle_find_node_hashed_add_root(bddm, one, zero, 2);
 var7 = bdBDDd_handle_find_node_hashed_add_root(bddm, one, zero, 7);

 /* check node pointers and handles */
 assert(zero == BDD_ROOT(bddm, 0)); /* since table was not doubled */
 assert(one  == BDD_ROOT(bddm, 1)); assert(var2 == 2); assert(var7 == 3);

 bddm1 = bdd_new_manager(100,50); /* make room for at least 100 nodes,
                                     overflow increment is 50 */

 bdd_make_cache(bddm1, 100, 50); /* apply2 needs a result cache, here the size
                                    is a hundred with increment 50 */

 /* apply operation on var2 and var7 in bddm; the result is
    a completely fresh bdd in bddm1 and a BDD pointer, named "and_2_7" */
 and_2_7 = bdd_apply2_hashed(bddm, BDD_ROOT(bddm, var2), /* BDD #1 */
                             bddm, BDD_ROOT(bddm, var7), /* BDD #2 */
                             bddm1, /* result BDD */
                             &and); /* leaf operation */

 bdd_update_statistics(bddm, 0); /* update statics group "0" with data from bddm
                                    before killing the manager */

 printf("Size of bddm: %d\n\n", bdd_size(bddm)); /* let's see the number of nodes created */

 bdd_kill_manager(bddm);

 printf("Size of bddm1: %d\n\n", bdd_size(bddm1));

 handle = BDD_LAST_HANDLE(bddm1);

 assert(handle == 0);
 assert(BDD_ROOT(bddm1, handle) == and_2_7);

 /* reset all mark fields in bddm1 before an apply1 operation */
 bdd_prepare_apply1(bddm1);

 /* a new bdd (which as an unlabeled graph is isomorphic to old one)
    in bddm1 is the result of the following apply operation */

 /* it's safe here to use and_2_7 since no operations were performed
    after it was calculated that could have entailed doubling of table */
 nand_2_7 = bdd_apply1(bddm1, and_2_7, bddm1, &not);

 bdd_update_statistics(bddm1, 1);
 printf("Size of bddm1: %d\n\n", bdd_size(bddm1));
 bdd_kill_manager(bddm);
 print_bdd(bddm1, and_2_7);  printf("\n\n");
 print_bdd(bddm1, nand_2_7);  printf("\n\n");
 bdd_print_statistics(0, "bddm"); /* print group 0 statistics with heading "bddm" */
 bdd_print_statistics(1, "bddm1"); /* print group 1 statistics with heading "bddm1" */
};
```

Figure 15: Simple use of package (continued)

```
bdd_ptr bddpaths[10];

/* function that updates BDD pointers floating around in user's code;
the remaining ones are kept in the sub_results list below */

void update_bddpaths(unsigned (*new_place) (unsigned node)) {
  int j;

  /* update the pointers in bddpaths */

  for (j = 0; j < exp_count; j++)
    bddpaths[j] = new_place(bddpaths[j]);
}

/* we can only update pointers that are not stored as local
   variables, so values of local variables are thrown onto a stack */

DECLARE_SEQUENTIAL_LIST(sub_results, unsigned);

bdd_ptr makepath(bdd_manager *bddm, ...,
                  void (*update_bddpaths)
                     (unsigned (*new_place) (unsigned node))) {
  ...
  bdd_ptr res, sub_res, default_state_ptr;

  sub_res = makepath(bddm, n+1, leaf_value, update_bddpaths);

  /* push BDD pointer sub_res on list sub_results */

  PUSH_SEQUENTIAL_LIST(sub_results, unsigned, sub_res);

  /* insert a new hashed node; thus potentially changing the pointer
  to the node known as sub_res above, and also potentially changing
  the pointers in bddpaths[10].  But bdd_find_leaf_hashed automatically
  updates all pointers in the sub_results list, since it was provided
  as an argument.  Also, the function update_bddpaths is called when a
  doubling of the table takes place and an appropriate renaming function
  new_place is supplied by bdd_find_leaf_hashed */

  default_state_ptr =
    bdd_find_leaf_hashed(bddm,
                         default_state,
                         SEQUENTIAL_LIST(sub_results),
                         update_bddpaths);

  /* restore the value of sub_res */

  POP_SEQUENTIAL_LIST(sub_results, unsigned, sub_res);
  ...
}
```

Figure 16: Dealing with volatility of BDD pointers

References

[1] Abdelwaheb Ayari, David Basin, and Andreas Podelski. Lisa: A specification language based on ws2s. In *CSL '97 Proceedings*, LNCS, 1998. to appear.

[2] D. Basin and N. Klarlund. Hardware verification using monadic second-order logic. In *Computer aided verification : 7th International Conference, CAV '95, LNCS 939*, 1995.

[3] D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. To appear. Extended version of: "Hardware verification using monadic second-order logic," *CAV '95*, LNCS 939, 1998.

[4] 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.

[5] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, WIA '96, Lecture Notes in Computer Science, 1260*. Springer Verlag, 1996.

[6] A. Boudet and H. Comon. Diophantine equations, presburger arithmetic and finite automata. In *Trees and algebra in programming - CAAP*, volume 1059 of *LNCS*.

[7] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing surveys*, 24(3):293–318, September 1992.

[8] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

[9] J.R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

[10] J.R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. Internat. Cong. on Logic, Methodol., and Philos. of Sci.* Stanford University Press, 1962.

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

[12] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–52, 1961.

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

[14] P. Godefroid and D.E. Long. Symbolic protocol verification with Queue BDDs. In *Proc. LICS' 96*, 1996.

[15] 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.

[16] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[17] J. Jensen, M. Jørgensen, and N. Klarlund. Monadic second-order logic for parameterized verification. Technical report, BRICS Report Series RS-94-10, Department of Computer Science, University of Aarhus, 1994.

[18] 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.

[19] 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.

[20] 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.

[21] N. Klarlund. Mona & Fido: the logic-automaton connection in practice. In *CSL '97 Proceedings*, 1998. To appear in LNCS.

[22] N. Klarlund, J. Koistinen, and M. Schwartzbach. Formal design constraints. In *Proc. OOPSLA '96*, 1996.

[23] N. Klarlund, M. Nielsen, and K. Sunesen. Automated logical verification based on trace abstraction. In *Proceedings of PODC '96*, 1996.

[24] N. Klarlund, M. Nielsen, and K. 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.

[25] 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.

[26] N. Klarlund and M. Schwartzbach. A domain-specific language for regular sets of strings and trees. In *Proc. Conference on Domain Specific Languages*, 1997.

[27] N. Klarlund and M. Schwartzbach. Regularity = logic + recursive data types. Technical report, BRICS, 1997. To appear.

[28] Nils Klarlund. The restriction problem for logics with automata-theoretic semantics. Technical report, AT&T Labs Research, 1998.

[29] 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.

[30] 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.

[31] Frank Morawietz and Tom Cornell. The logic-automaton connection in linguistics. Technical report, Universität Tübingen Seminar für Sprachwissenschaft, 1998. `http://www.sfs.nphil.uni-tuebingen.de/~frank/papers.html`.

[32] Paritosh K. Pandya. Dcvalid 1.2. Technical report, Theoretical Computer Science Group, Tata Institute of Fundamental Research, 1997. Manual and system available from: `http://www.tcs.tifr.res.in/~pandya/dcvalid.html`.

[33] Thomas R. Shiple, James H. Kukula, and Rajeev K. Ranjan. A comparison of Presburger engines for EFSM reachability. In *Computer Aided Verification, CAV '98, Proceedings*, LNCS. Springer Verlag, 1998.

[34] 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.

[35] 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.

[36] 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.

Several of the articles are available from `http://www.brics.dk/mona/papers.html`.

Recent BRICS Notes Series Publications