A Domain-Specific Language for Regular Sets of Strings and Trees

Nils Klarlund

AT&T Labs–Research
klarlund@research.att.com

Michael I. Schwartzbach

BRICS, University of Aarhus
mis@brics.dk

Abstract

We propose a new high-level programming notation, called FIDO, that we have designed to concisely express regular sets of strings or trees. In particular, it can be viewed as a domain-specific language for the expression of finite-state automata on large alphabets (of sometimes astronomical size).

FIDO is based on a combination of mathematical logic and programming language concepts. This combination shares no similarities with usual logic programming languages. FIDO compiles into finite-state string or tree automata, so there is no concept of run-time. It has already been applied to a variety of problems of considerable complexity and practical interest.

In the present paper, we motivate the need for a language like FIDO, and discuss our design and its implementation.

We show how recursive data types, unification, implicit coercions, and subtyping can be merged with a variation of predicate logic, called the Monadic Second-order Logic *(M2L) on trees. FIDO is translated first into pure M2L via suitable encodings, and finally into finite-state automata through the MONA tool.*

1 Introduction

Finite-state problems are everywhere, embedded in many layers of software systems, but are often difficult to extract and solve computationally. This basic observation is the motivation for the work presented in this paper.

Recent research by us and our colleagues has exploited the Monadic Second-Order Logic (M2L) on finite strings and trees to solve interesting and challenging problems. In each case, the results are obtained by identifying an inherent regularity in the problem domain, thus reducing the problem to questions of regular string or tree languages. Successful applications today include verification of concurrent systems [9, 8], hardware verification [2], software engineering [10], and pointer verification [7]. Work in progress involves a graphical user interface for regular expressions extended with M2L and document logics for the WWW.

The rôle of M2L in this approach is to provide an extraordinarily succinct notation for complicated regular sets. Our applications have demonstrated that this notation in essence can be used to describe properties, where finite state automata, regular expressions, and grammars would be tend to be cumbersome, voluminous, or removed from the user's intuition. This is hardly surprising, since M2L is a variation on predicate logic and thus natural to use. Also, it is known to be non-elementarily more succinct than the other notations mentioned above. Thus, some formulas in M2L describe regular sets for which the size of a corresponding DFA compared to the size of the formula is not bounded by any finite stack of exponentials.

The flip side of this impressive succinctness is that M2L correspondingly has a non-elementary lower bound on its decision procedure. Surprisingly, the MONA implementation of M2L [5] can handle nontrivial formulas, some as large as 500,000 characters. This is due in part to the application of BDD techniques [4], specialized algorithms on finite-state automata [3], and careful tuning of the implementation [11]. Also, it turns out that the intermediate automata generated, even those resulting from subset constructions, are usually not big compared to the automata representing the properties reasoned about.

The successful applications of M2L and MONA reside in a common, productive niche: they require the specification of regular sets that are too complicated to describe by other means, but not so complicated as to be infeasible for our tools.

While the basic M2L formalism is simple and quite intuitive, early experience quickly indicated that this formalism in practice suffers from its primitive domain of discourse: bit-labeled strings and trees. In fact, M2L specifications are uncomfortably similar to assembly code programs in their focus on explicit manipulations of bit patterns. For M2L interpreted on trees, the situation is even worse, since the theory of two or more successors is far less familiar and intuitive than the linear sublogic.

Similarly to the early experiences with machine languages, we found that M2L "programmers" spent most of their time debugging cumbersome encodings.

Our contributions

In this paper, we propose a domain-specific programming formalism FIDO that combines mathematical logic and recursive data types in what we believe are new ways.

We suggest the following four kinds of values: finite domains, recursive data values (labeled by finite domains), positions in recursive data values, and subsets of such positions. We show that many common programming language concepts (like subtyping, coercions, and unification) make sense when the underlying semantics is based on assigning an automaton (and not a store transformer) to expressions.

This semantic property allows us to view the compilation process as calculations on values that are deterministic, finite-state automata, just as an expression evaluator calculates on numbers to arrive at a result. That is, automata are the primitive objects that are subjected to operations reflecting the semantics of the language.

This view is quite different from the method behind most state-machine formalisms used in verification (such as the Promela language [6]): a language resembling a general purpose language expresses a single finite-state machine, whose state space and transition system is constructed piecemeal from calculations that explore the state space.

Our view, however, is similar to some uses of regular expressions for text matching, except that most implemented algorithms avoid the construction of deterministic automata.

FIDO is implemented and provides, along with supporting tools, an optimizing compiler into M2L formulas. It has been used for several real-life applications and is also the source of the biggest formulas yet handled by MONA.

In this article, we motivate and explain FIDO. In particular, we discuss the type system and compilation techniques. We also give several examples (some taken from articles already published, where we have used FIDO without explaining its origin or design). Some technical considerations concerning the relationship between our data structures for tree automaton representation [3] and the compilation process will be explained elsewhere.

2 M2L and MONA

Basic M2L has a very simple syntax and semantics. Formulas are interpreted on a binary tree (or a string) labeled with bit patterns determining the values of free variables. First-order terms (t) denote positions in the tree and include first-order variables (p) and successors ($t.0$ and $t.1$). Second-order terms (T) denote sets of positions (i.e. *monadic* predicates) and include second-order variables (P), the empty set ($\emptyset$), unions ($T_1 \cup T_2$), and intersections ($T_1 \cap T_2$). The basic predicates are set membership ($t \in T$), equality ($t_1 = t_2$), ancestor relation ($t_1 < t_2$), and set inclusion $T_1 \subseteq T_2$). The logic permits the usual connectives ($\wedge$, $\vee$, $\neg$) and first and second-order quantifiers ($\forall_1$, $\exists_1$, $\forall_2$, $\exists_2$). By convention, a leaf is a position p for which $p = p.0$ and $p = p.1$. The sublogic for strings uses only the 0-successor.

The MONA tool accepts such formulas in a suitable ASCII syntax and produces a minimum DFA that accepts all trees satisfying the given formula. Thus, satisfiability of a formula is equivalent to nonemptyness of the derived automaton, and validity is equivalent to totality. The values of free variables in the formula are encoded in the alphabet of the automaton. Thus, a formula with 32 free variables yields an alphabet Σ of size 2^{32}. In the internal representation of these automata, the transition function is shared, multi-terminal Σ-BDD. With these BDD techniques, the MONA tool has processed formulas with hundreds of thousands of characters in a few minutes.

3 The Motivation

A small example will motivate the need for a high-level notation. Assume that we wish to use MONA to prove the following (not too hard) theorem: for every string in $(a+b)^*c$, any a is eventually followed by c.

To state this theorem in M2L, we must first choose an encoding of the labels a, b, and c. For this purpose we introduce two free second-order variables X_0 and X_1. The labels can be encoded according to the following (arbitrary) schema: a position p has label a if $p \notin X_0 \wedge p \notin X_1$, that is, a corresponds to the bit pattern 00. Similarly, we can assign to b the bit pattern 01 and to c the pattern 10. The property "a is eventually followed by c" becomes the formula:

$$\begin{aligned}\psi \equiv \forall_1\ p : (p \notin X_0 \wedge p \notin X_1) \Rightarrow \\ (\exists_1 q : p < q \wedge (q \in X_0 \wedge q \notin X_1))\end{aligned}$$

The regular expression $(a+b)^*c$ can in a similar way be encoded as the formula:

$$\begin{aligned}\phi \equiv \forall_1\ p : (\neg(p \in X_0 \wedge p \in X_1)) \wedge \\ ((p \in X_0 \wedge p \notin X_1) \Leftrightarrow p = p.0)\end{aligned}$$

and the theorem above is then formally stated as the implication $\phi \Rightarrow \psi$. The MONA tool will readily verify that this formula is an M2L tautology, thus proving our theorem.

A reason for M2L specifications being much more voluminous than promised should now be apparent: there is a significant overhead in encodings. Moreover, there are no automatic checks of the consistent use of bit patterns.

Support for such encodings is usually supplied by a type system. For M2L on strings, regular sets immediately suggest themselves as notions of types. It is quite common for M2L formulas to be of the implicational form $\phi \Rightarrow \psi$, where ϕ is a formula restricting the strings to a coarse regular set and ψ provides the more intricate restrictions. Thus, a high-level version of the above formula could look like:

string x: (a+b)*c;
∀**pos** p:x.(p=a ⇒ ∃**pos** q:x.(p<q ∧ q=c))

The keywords **string** and **pos** are intended to declare free variables of these two kinds. This formula can be read as: "for all positions p in the string x, if p has label a, then there exists a position q, also in x, such that p is before q and q has label c". The main formula is almost the same as the MONA version, but the proper use of labels is now supported by the compiler and can be verified by a type checker.

For M2L interpreted on trees, however, there is no intuitive analogue to regular expressions. But from programming languages we know an intuitive and successful formalism for specifying coarse regular sets of trees: recursive data types. Thus, we adopt a well-known and trusted programming concept into our high-level notation. Using this idea, we may prove our theorem as follows:

type T = a,b(next: T) | c;
string x: T;
∀**pos** p:x.(p=a ⇒ ∃**pos** q:x.(p<q ∧ q=c))

Arbitrary recursive data types may of course be expressed directly as formulas, but the translation is voluminous and best performed automatically. The translation also solves the problem that the Mona decision procedure works on formulas whose domain of discourse is only binary trees, whereas values of recursive data types are trees with a varying number of branches. (The solution is rather technical, since it involves bending the recursive data type value into the shape of a binary branching tree.)

Note that not all regular tree sets can be captured by recursive data types. Consider binary trees, in which nodes are colored red, green, or blue. The subset of trees in which at most one node is colored blue is *not* a recursive data type; however, it is easily captured by the following FIDO specification:

type RGB = red,green,blue(left, right: RGB) | leaf;
tree x: RGB;
∀**pos** p,q: x.(p=blue ∧ q=blue ⇒ p=q)

Certainly, more advanced and complicated notions of data types could similarly be adopted [1]. However, the FIDO philosophy is to rely heavily on *standard* programming language concepts to describe complex structures and operations. The ambition is that these idioms should be merged seamlessly with logical concepts that describe complex properties of such structures.

In general, we allow <u>fi</u>nite <u>do</u>mains (from which the name FIDO derives) to be the values of nodes. Finite domains are constructed conjunctively and disjunctively from enumerated and scalar types. Thus the alphabets of tree automata reading such recursive data types easily become very large.

4 The Design

While this paper is not intended as a proper language report, we will explain the more interesting or unusual concepts that the FIDO notation provides.

Domains and Data Types

Finite domains are constructed from simple scalar lists, freely combined with a product operator (&) and a union operator (|). When the union of two finite domains is formed, it is required that they are disjoint. Thus, if we define the domains:

```
type Turn = [1..2];
type PC = a,b,c,d;
type State = PC & PC & Turn;
```

then a value of the domain State may be written as State:[a,b,2]. From the more complicated definitions:

```
type A = a1,a2;
type B = b1,b2,b3;
type C = A | B;
type D = A & B & C;
type E = C & D;
```

we obtain values as: E:[a1,[a2,b3,[a2,b1]]]. In formulas, finite domain values may be unified using a syntax such as State:[pc?,a,r?], where ps and r are unification variables.

The recursive data types are quite ordinary, except that the constructors are generalized from single names to finite domains.

The finite domains could of course be encoded as (non-recursive) data types. We have chosen to have a separate concept for several reasons. First, the distinction between trees and their labels seems intuitive for many applications. Second, we can allow more operations on finite domains that on trees; for example, the introduction of unification or concatenation on trees would yield an undecidable formalism. Third, in the translation into automata, finite domains are encoded in BDDs whereas trees are encoded in the state space; often, it is necessary for the programmer to control this choice. An example is:

```
type Comp = State(next: Comp) | done;
```

which is a linear data type of sequences of state values terminated by a node labeled done. A non-linear example is:

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

denoting some binary trees. The notation [1..10] abbreviates the corresponding 10 scalars.

Variables

There are four kinds of variables in FIDO. We introduce them by examples. A *domain variable* s that ranges over states may be declared as

```
dom S: State;
```

Tree variables (recursive data type variables) x and y may be declared as:

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

Each variable defines its own space of positions. Thus, a position in x cannot be used to denote a node in y. To declare a *position variable* that may denote positions in either x or y, we write:

```
pos p: x, y;
```

A value of this variable points to a node in either x or y, but in any case, the node pointed to is either red or black. Similarly, a *set variable* S containing positions in the union of x's and y's position spaces may be declared as:

```
set S: x, y;
```

Quantification

All variables can be quantified over. For example, the formula "there is a computation that contains a loop" may involve quantification over both strings (trees), finite domains, and positions:

∃**string** x: Comp. ∃**dom** s: State.
    ∃**pos** p,q: x.(p<q ∧ p=s ∧ q=s)

Types

A type may have one of four different kinds: pos, set, dom, and tree. The pos kind corresponds to first-order terms, i.e. positions in trees; the set kind similarly encompasses second-order terms; the dom kind is new compared to M2L and describes values of finite domains; finally, the tree kind is a further extension that captures entire trees as values.

Within each kind, a type is further refined by a set of tree names and a set of data type names. For example, the type (pos,{x,y},{R,S,T}) denotes positions of nodes in either the tree x or y that are roots of subtrees of one of the data types R, S, or T. These refined types prove to be very convenient in restricting free variables in the model and in expressing relativized quantifications. Furthermore, this type structure proves crucial for optimizations in the implementation.

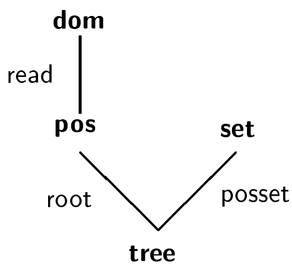
The type rules impose restrictions on all operators in the language. Generally, the rules boil down to trivial statements about finite sets. For example, if the terms s_i have types (set,X_i,D_i), then $s_1 \cap s_2$ has type (set,$X_1 \cap X_2$,$D_1 \cap D_2$). Also, if the term p has type (pos,X,D), then the term p.n has type (pos,X,{T.n | T∈ D}), where T.n is the data type reached from T along an n-successor.

Some formulas can be decided purely on the basis of the type system. For example, if p has type (pos,X_p,D_p) and s has type (set,X_S,D_S), then the formula $p \in S$ is false if $X_p \cap X_S = \emptyset$ or $D_p \cap D_S = \emptyset$. Such static decisions are exploited by the FIDO compiler.

Notational Conveniences

A formal notation has a tendency to become a quagmire of details. In the design of FIDO, we have attacked this problem in three different ways.

First, it is often convenient implicitly to *coerce* values between different kinds. This we have expressed through a simple subtype structure. Two types (κ_1,X_1,D_1) and (κ_2,X_2,D_2) are related by the subtype order if $X_1 \subseteq X_2$, $D_1 \subseteq D_2$, and κ_1 is below κ_2 in the following finite order:



The order relations have been decorated with coercions functions: posset computes the set of positions in a tree, root finds the root positions of a tree, and read computes the label of a position. This subtype structure is exploited to automatically insert coercions. Note that our subtype structure clearly is semantically coherent, so that coercions are unique [12]. If we added the coercion: singleton: **pos** → **set**, then semantic coherence would fail.

Second, we allow implicit *casts* between finite domains. For example, in the definitions:

type Fruit = apple,orange;
type Root = carrot,potato;
type Vegetable = Fruit | Root;

we will allow values of the domains Fruit and Root to be used directly as values of the domain Vegetable, even though they strictly speaking should be expressed as e.g. **cast**(Fruit:apple,Vegetable).

Third, we allow sensible *defaults* whenever possible. Thus, if a name can unambiguously be determined to have a specific meaning, then all formal qualifiers may be dismissed. For example, if the name orange is only used as a scalar in the domain Fruit, then the constant Fruit:orange may be written simply as orange.

As a specific example of these techniques, consider the previous theorem:

type T = a,b(next: T) | c;
tree x: T;
∀**pos** p:x.(p=a ⇒ ∃**pos** q:x.(p<q ∧ q=c))

We have already used a number of syntactic conveniences here. From the above specification, the compiler inserts the necessary coercions to reconstruct the more explicit code:

```
type T = a,b(next: T) | c;
tree x: T;
∀pos p:x,T.(read(p)=T:a ⇒
        ∃pos q:x,T.(p<q ∧ read(q)=T:c))
```

which is somewhat harder to read. In a real-life 12-page formula, more than 400 such pedantic corrections are automatically performed.

Decompilers

Any compiler writer must also consider the need for decompilers. In the case of FIDO and MONA, specifications are translated into a more primitive logic. This is fine, if we only want to decide validity. However, MONA also has the ability to generate counter-examples for invalid formulas. But a MONA counter-example will make little sense for a FIDO programmer, since it will have a completely different structure and be riddled with bit patterns. Consequently, the FIDO system provides a decompiler that lifts such counter-examples into the high-level syntax.

Another use of MONA, illustrated in the following section, is to generate specific automata. For this application, FIDO provides a different decompiler that expresses an automaton as a particular kind of attribute grammar at the level of recursive data types.

5 Examples

We now provide a few examples illustrating the benefits of the FIDO notation. We include applications that aim to synthesize automata as well as some that aim to verify properties. For each case we present a toy example in some detail and sketch a large, previously published application of a similar nature.

Synthesis

The following example considers (a fragment of) the HTML syntax. Not all syntactically correct HTML-specifications should be allowed. For example, a document should never contain an anchor within another anchor (to not confuse the reader). Such a constraint could be incorporated into the context-free syntax, but it would essentially double the number of non-terminals. However, we can easily capture HTML parse trees as values of a recursive data type. On these trees we can then express as a logical formula the restriction that we wish to impose:

```
type HTML = word |
            anchor(u: URL, a: HTML) |
            bold(b: HTML) |
            italic(i: HTML) |
            paragraph |
            rule |
            list(l: LIST);
type LIST = empty |
            entity(h: HTML, next: LIST);
type URL = url;

func Restrict(tree h: HTML): formula;
  ∀pos p: h,HTML.(p=anchor ⇒
  ¬(∃pos q: h,HTML.(p<q ∧ q=anchor)))
end;
tree H: HTML;

Restrict(H)
```

Furthermore, we can introduce any number of such restrictions in a completely modular manner. From this specification, the FIDO system can produce an attribute grammar working on parse trees, which could then easily be incorporated into an HTML development system. In this case, the attribute grammar has three attribute values, corresponding to zero, one, or too many nested anchors. Only trees synthesizing the values zero or one are accepted. The transitions, which are simply inherited from the tree automaton that MONA computes, are as follows:

```
HTML | word:      [] ↦ 0
HTML | anchor:    [0,0] ↦ 1
                  [0,1] ↦ 2
                  [0,2] ↦ 2
HTML | bold:      [0] ↦ 0
                  [1] ↦ 1
                  [2] ↦ 2
HTML | italic:    [0] ↦ 0
                  [1] ↦ 1
                  [2] ↦ 2
HTML | paragraph: [] ↦ 0
HTML | rule:      [] ↦ 0
HTML | list:      [0] ↦ 0
                  [1] ↦ 1
                  [2] ↦ 2
LIST | empty:     [] ↦ 0
LIST | entity:    [0,0] ↦ 0
                  [0,1] ↦ 1
                  [1,0] ↦ 1
```

```
                [1,1] ↦ 1
                [0,2] ↦ 2
                [2,0] ↦ 2
                [1,2] ↦ 2
                [2,1] ↦ 2
                [2,2] ↦ 2
URL |   url:    [] ↦ 0
```

The transition HTML | anchor: [0,0] ↦ 1 means that if the node is an anchor and each of its two subtrees synthesizes the attribute value 0, then it should synthesize the attribute value 1.

These simple ideas have been exploited in a collaboration with the Ericsson telecommunications company to formalize the constraints of design architectures [10].

Verification

Two specifications, of say distributed systems, can be compared by means of the implication or bi-implication connective. Consider a simple-minded mutual exclusion protocol for two processes with a shared memory:

```
Turn: Integer range 1..2 := 1;

task body Proc1 is
begin
  loop
    a: Non_Critical_Section_1
    b: loop exit when Turn = 1; end loop;
    c: Critical_Section_1;
    d: Turn := 2
  end loop;
end Proc1;

task body Proc2 is
begin
  loop
    a: Non_critical_Section_2;
    b: loop exit when Turn = 2; end loop;
    c: Critical_Section_2;
    d: Turn := 1;
  end loop
end Proc2
```

The FIDO specification models all valid interleaved computations and simply asks whether the safety property holds:

```
type Turn = [1..2];
type PC = a,b,c,d;
type State = PC & PC & Turn;
type Computation = State(next: Computation) | done;
string α: Computation;
func Trans(dom s,t: State): formula;
  let dom pc: PC; dom r: Turn.(
    trans(s,t)
      [a,pc?,r?] ↦ [b,pc?,r?] |
      [b,pc?,1] ↦ [c,pc?,1] |
      [b,pc?,2] ↦ [b,pc?,2] |
      [c,pc?,r?] ↦ [d,pc?,r?] |
      [d,pc?,r?] ↦ [a,pc?,2] |
      [pc?,a,r?] ↦ [pc?,b,r?] |
      [pc?,b,2] ↦ [pc?,c,2] |
      [pc?,b,1] ↦ [pc?,b,1] |
      [pc?,c,r?] ↦ [pc?,d,r?] |
      [pc?,d,r?] ↦ [pc?,a,1]
    end
  )
end;

func Valid(string x: Computation): formula;
  x=[a,a,1];
  ∀pos p: x.(
    if p.next≠done then
      let dom s,t: State.
      (p=s?; p.next=t?; Trans(s,t))
    end
  )
end;

func Mutex(string x: Computation): formula;
  ∀pos p: x.(p≠[c,c,?])
end;

Valid(α) ⇒ Mutex(α)
```

The formula **trans**(s,t) ... **end** denotes the binary relation on State domain values that hold for the pairs of values that can simultaneously match one of the listed cases.

The corresponding raw MONA formula looks like:

```
((ex1 [UNI_alpha] p: (root (p,[p]) & (all1 [UNI_alpha] q: (
(p <= q + 0) => ( ((q notin G0) & (q <= q.0 - 1)) | (((((((
q in G0) & (q notin S0)) & (q notin S1)) & (q notin S2)) &
(q notin S3)) & (q notin S4)) & (q = q.0))))))) => ( ((ex1
[UNI_x] POS26: (root (POS26,[POS26]) & ((POS26 notin G0) &
(((((POS26 notin S0) & (POS26 notin S1)) & (POS26 notin S2))
& (POS26 notin S3)) & (POS 26 notin S4))))) & (all1 [UNI_x
] POS_p: ((all1 [UNI_x] POS31: ((((POS_p in G 0) | (POS31
!= POS_p.0)) & ((POS_p notin G0) | (POS31 != POS_p))) | (PO
S31 n otin G0))) => (ex1 [UNI_x] POS41: (((POS_p notin G0)
& ((((POS_p notin G0) & (POS41 = POS_p.0)) | ((POS_p in G0)
& (POS41 = POS_p))) & (POS41 notin G0))) & (ex0 s0_pc,s1_pc
: (ex0 s0_r: ((((((((((((((POS_p in S0) <=> s0_pc) & ((
POS_p in S1) <=> s1_p c)) & (POS_p in S2)) & (POS_p in S3))
& ((POS_p in S4) <=> s0_r)) & ((((((POS41 in S0) <=> s0_pc)
& ((POS41 in S1) <=> s1_pc)) & (~(POS41 in S2))) & (~(POS
41 in S3))) & (~(POS41 in S4)))) | (((((((POS_p in S0) <=>
s0_pc) & ((POS_p in S1) <=> s1_pc)) & (~(POS_p in S2))) & (
```

```
POS_p in S3)) & ((POS_p in S4) <=> s0_r)) & ((((((POS41 in
S0) <=> s0_pc) & ((POS41 in S1) <=> s1_pc)) & (POS41 in S2)
) & (POS41 in S3)) & ((POS41 in S4) <=> s0_r)))) | (((((((P
OS_p in S0) <=> s0_pc) & ((POS_p in S1) <=> s1_pc)) & (POS_p
in S2)) & (~(POS_p in S3))) & (~(POS_p in S4))) & ((((((
POS41 in S0) <=> s0_pc) & (s1_t <=> s1_pc)) & (POS41 in S2)
) & (~(POS41 in S3))) & (~(POS41 in S4))))) | (((((((POS_p
in S0) <=> s0_pc) & (s1_s <=> s1_pc)) & (POS_p in S2)) & (
~(POS_p in S3))) & (POS_p in S4)) & ((((((POS41 in S0) <=>
s0_pc) & ((POS41 in S1) <=> s1_pc)) & (~(POS41 in S2))) & (
POS41 in S3)) & (POS41 in S4)))) | (((((((POS_p in S0) <=>
s0_pc) & ((POS_p in S1) <=> s1_pc)) & (~(POS_p in S2))) &
(~(POS_p in S3))) & ((POS_p in S4) <=> s0_r)) & ((((((POS41
in S0) <=> s0_pc) & (s1_t <=> s1_pc)) & (POS41 in S2)) &
(~(POS41 in S3))) & ((POS41 in S4) <=> s0_r)))) | ((((((POS
_p in S0) & (POS_p in S1)) & ((POS_p in S2) <=> s0_pc)) & (
(POS_p in S3) <=> s1_pc)) & ((POS_p in S4) <=> s0_r)) & (((
((~(POS41 in S0)) & (~s 1_t)) & ((POS41 in S2) <=> s0_pc))
& ((POS41 in S3) <=> s1_pc)) & (POS41 in S4)))) | ((((((~(
POS_p in S0)) & s1_s) & ((POS_p in S2) <=> s0_pc)) & ((POS_
p in S3) <=> s1_pc)) & ((POS_p in S4) <=> s0_r)) & (((((POS
41 in S0) & s 1_t) & ((POS41 in S2) <=> s0_pc)) & ((POS41 in
S3) <=> s1_pc)) & ((POS41 in S4) <=> s0_r)))) | (((((s0_s
& (~(POS_p in S1))) & ((POS_p in S2) <=> s0_pc)) & ((POS_p
in S3) <=> s1_pc)) & (POS_p in S4)) & (((((POS41 in S0) &
(~ (POS41 in S1))) & ((POS41 in S2) <=> s0_pc)) & ((POS41
in S3) <=> s1_pc)) & (POS41 in S4)))) | ((((((POS_p in S0)
& (~s1_s)) & ((POS_p in S2) <=> s0_pc)) & ((POS_p in S3)
=> s1_pc)) & (~(POS_p in S4))) & (((((~(POS41 in S0)) & (
POS41 in S1)) & ((POS41 in S2) <=> s0_pc)) & ((POS41 in S3)
=> s1_pc)) & (~(POS41 in S4))))) | ((((((~(POS_p in S0)) &
(~s1_s)) & ((POS_p in S2) <=> s0_pc)) & ((POS_p in S3) <=>
s1_pc)) & ((POS_p in S4) <=> s0_r)) & (((((POS41 in S0) &
(~(POS41 in S1))) & ((POS41 in S2) <=> s0_pc)) & ((POS41 in
S3) <=> s1_pc)) & ((POS41 in S4) <=> s0_r)))))) => (all1 [
UNI_x] POS_p: (((((POS_p in S0) | (POS_p notin S1)) | (POS_p
in S2)) | (POS_p notin S3)) | (POS_p in G0))))))))))
```

Since the simplistic mutual exclusion protocol is clearly correct, this formula is a tautology. However, if we mistakenly tried to verify that Proc2 could never enter the critical region:

func Mutex(**string** x: Computation): **formula**;
  ∀**pos** p: x.(p≠[?,c,?])
end;

then FIDO would generate the counterexample:

```
alpha = Computation:[a,a,1](
          Computation:[b,a,1](
          Computation:[b,b,1](
          Computation:[c,b,1](
          Computation:[d,b,1](
          Computation:[a,b,2](
          Computation:[a,c,2](
          Computation:done)))))));
```

which exactly describes such a computation.

For more realistic examples, internal events can be projected away by means of the existential quantifier. In [8], a detailed account is given of an application of the FIDO language to a verification problem posed by Broy and Lamport in 1994. The distributed systems are described in an interval logic, which is easily defined in FIDO. The evolution of a system over a finite segment of time is modeled as a recursive, linear data type with a constructor that define the current event. Thus position variables denote time instants. The thousands of events possible in the distributed systems that are compared are described by the types:

type Value = initVal,1;
type Loc = l0,l1;
type Ident = id0,id1;
type ValTag = MemVals,error;
type LocTag = MemLocs,error;
type TVal = Value & ValTag;
type TLoc = Loc & LocTag;
type Flag = normal,exception;
type RetFlag = BadArg,MemFailure;
type RpcFlag = RPCFailure,BadCall | RetFlag;
type Visible = observable,internal;
type ProcVal = ReadProc,WriteProc;
type ProcTag = procVal,error;
type TProc = ProcVal & ProcTag;
type NumArgs = n1,n2;
type Args = TLoc & TVal;
type Opn = rd,wrt;
type Mem = Opn & Loc & Value & Flag & Ident;
type Read = TLoc & Ident & Visible;
type Write = TLoc & TVal & Ident & Visible;
type Ret = TVal & Flag & RetFlag & Ident & Visible;
type RmtCall = TProc & NumArgs & Args & Ident;
type RpcRet = TVal & Flag & RpcFlag & Ident;
type Event = Mem | Read | Write | Ret | RmtCall |
          RpcRet | Tau;
type Comp = Event(next: Comp) | Empty;

The property to be verified requires 12 pages of FIDO specification which translates into an M2L formula of size 500,000 characters.

An entirely different use of FIDO allows us to verify many properties of PASCAL programs that use pointers [7]. By encoding a store as a string and using FIDO formulas to describe the effects of program statements, we can automatically verify some desirable properties. An example is the following program, which performs an in-situ reversal of a linked list with colored elements:

```
program reverse;
type Color = (red,blue);
     List = ^Item;
     Item = record
              case tag: Color of
              red,blue: (next: List)
            end;
var x,y,p: List;
```

```
begin
    while x<>nil do
    begin
       p:=x^.next;
       x^.next:=y;
       y:=x;
       x:=p
    end
end.
```

With our system, we can automatically verify that the resulting structure is still a linked list conforming to the type `List`. We can also verify that no pointer errors have occurred, such as dangling references or unclaimed memory cells. However, we cannot verify that the resulting list contains the same colors in reversed order. Still, our partial verification will clearly serve as a finely masked filter for many common programming errors.

The PASCAL tool adds another level of compilation, from (simple) PASCAL programs to FIDO specifications to M2L formulas and finally to finite-state automata accepting encodings of the initial stores that are counterexamples. The above program translates into 10 pages of FIDO specification which expands into a 60,000 character M2L formula. The resulting automaton is of course tiny since there are no counterexamples, but the largest intermediate result has 74 states and 297 BDD-nodes. A direct translation into MONA would essentially add all the complexities of the FIDO compiler to the implementation of the PASCAL tool.

6 The Implementation

We have implemented parsing, symbol analysis, and type checking in entirely standard ways. What is non-standard is that every subterm is compiled into a tree automaton through an intermediate representation as an M2L formula. Thus resource allocation becomes a question of managing bit pattern encodings of domain values, which are expressed in M2L formulas. We have strived to achieve a parsimonious strategy, since every bit squandered may potentially double the MONA execution time.

As a concrete example, consider the type:

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

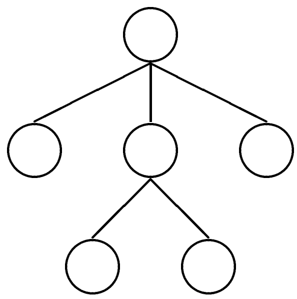
Its encoding in MONA requires seven bits in all. Two *type bits* `T0` and `T1` are used to distinguish between the types Tree and Enum and special null nodes in a tree; a single *group bit* `G0` is used to distinguish between the red-black and the leaf variants; and four *scalar bits* `S0`, `S1`, `S2`, and `S3` are used to distinguish between the values of each final domain, the largest of which is [0..10].

As an example, the formula:

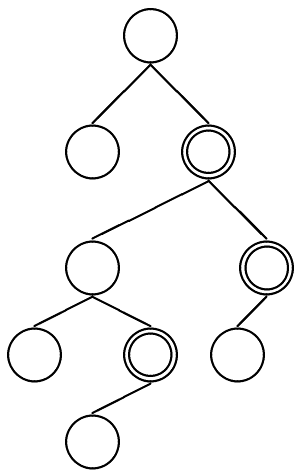
```
macro TYPE_Tree(var1 p) =
  (p in T0) & (p notin T1);
```

expresses that the type Tree is encoded by the bit pattern 10.

The null nodes are required to encode an arbitrary fan-out in a binary tree. For example, the tree:



is represented as:



where the null nodes have double lines.

A well-formed value of the type Tree is described by the MONA predicate `TREE_Tree`. It imposes the proper relationship between types and values of nodes and their descendants. A technical problem is that this predicate is most naturally described through recursion which is not available in M2L. This is solved by phrasing the requirements through a universal quantification that imposes sufficient, local well-formedness properties:

```
macro TREE_Tree(var1 p) =
  TYPE_Tree(p) &
  (all1 q: (p<=q) =>
           (NULL(q) | WF_Tree(q) | WF_Enum(q))
  );
```

The NULL and WF predicates describe the relationship between a single node and its immediate descendants:

```
macro NULL(var1 p) =
  (p notin T0) & (p notin T1) &
  (p notin G0) &
  (p notin S0) & (p notin S1) &
  (p notin S2) & (p notin S3);

macro TYPE_Enum(var1 p) =
  (p notin T0) & (p in T1);

macro GROUP_Tree_red_black(var1 p0) =
  (p notin G0);

macro GROUP_Tree_leaf(var1 p) =
  (p in G0);

macro GROUP_Tree(var1 p) =
  GROUP_Tree_red_black(p) | GROUP_Tree_leaf(p);

macro SCALAR_Enum(var1 p) =
  (p notin S3) |
  ((p notin S2) & ((p notin S1) | (p notin S0)));

macro SCALAR_Tree_red_black(var1 p) =
  true;

macro SCALAR_Tree(var1 p) =
  SCALAR_Tree_red_black(p);

macro SUCC_Enum(var1 p) =
  (p=p.0) & (p=p.1);

macro SUCC_Tree_red_black(var1 p) =
  (p<p.0) & (p<p.1) & (p.1<p.11) & (p.11=p.111) &
  NULL(p.1) & NULL(p.11) &
  TYPE_Tree(p.0) & TYPE_Tree(p.10) &
  TYPE_Enum(p.110);

macro SUCC_Tree_leaf(var1 p) =
  (p=p.0) & (p=p.1);

macro WF_Enum(var1 p) =
  TYPE_Enum(p) & SCALAR_Enum(p) & SUCC_Enum(p);
```

```
macro WF_Tree(var1 p) =
  TYPE_Tree(p) &
  ((GROUP_Tree_red_black(p) &
    SCALAR_Tree_red_black(p) &
    SUCC_Tree_red_black(p)
   ) |
   (GROUP_Tree_leaf(p) &
    (p notin S0) & SUCC_Tree_leaf(p)
   )
  );
```

Formulas are encoded in a simple inductive manner. For illustration, consider the tiny formula p∈s, where the arguments are general terms. The term p of kind **pos** generates a tuple $< p, \phi >$ where p is a first-order variable constrained by the formula ϕ. Similarly, the term s of kind **set** generates a tuple $< s, \psi >$, where s is now a second-order variable. The term p∈s then generates the formula $\exists p : \exists s : \phi \wedge \psi \wedge p \in s$. Note how existential quantification corresponds to discharging of registers. It is a fairly straightforward task to provide similar templates for all the FIDO constructs, thereby providing a compositional semantics and a recipe for a systematic translation.

As a concrete example, consider the formula:

tree x: Tree;
x.left.right.left=red

which describes the regular set of trees in which a specific node exists and is colored red. It is encoded as the following MONA formula:

```
macro DOT_right(var1 p,var1 q) =
  (TYPE_Tree(p) &
   GROUP_Tree_red_black(p) & (q=p.0)
  ) |
  (TYPE_Tree(p) &
   GROUP_Tree_leaf(p) & (q=p)
  );

macro DOT_left(var1 p,var1 q) =
  (TYPE_Tree(p) &
   GROUP_Tree_red_black(p) & (q=p.10)
  ) |
  (TYPE_Tree(p) &
   GROUP_Tree_leaf(p) & (q=p)
  );

assume ex1 p: root(p) & TREE_Tree(p);

ex0 t0_1,t1_1,g0_1,s0_1:
ex0 t0_2,t1_2,g0_2,s0_2:
```

```
(ex1 POS6:
  (ex1 POS5:
    (ex1 POS4:
      (ex1 POS3:
       root(POS3) & DOT_left(POS3,POS4)
      ) &
      DOT_right(POS4,POS5)
    ) &
    DOT_left(POS5,POS6)
   ) &
   (t1_1<=>(POS6 in T1)) &
   (t0_1<=>(POS6 in T0)) &
   (g0_1<=>(POS6 in G0)) &
   (s0_1<=>(POS6 in S0)) &
   (t0_2 & ~t1_2 & ~g0_2 & ~s0_2) &
   (g0_1 <=> g0_2) & (s0_1 <=> s0_2)
);
```

The analogy to run-time is the computation by MONA of a finite-state automaton from the generated formula. This is always guaranteed to terminate, but may be prohibitively expensive. Thus, the FIDO compiler does extensive optimizations at many levels, in most cases relying heavily on the type structure. FIDO formulas are symbolically reduced to detect simple tautologies and to eliminate unnecessary variables and quantifiers. A careful strategy is employed to allocate short bit patterns for finite domains, which includes a global analysis of concrete uses.

We have also discovered that the FIDO type structure contains a wealth of information that is not currently being exploited by the MONA implementation. An ongoing development effort will enrich the notion of tree automata to accommodate positional information that can be derived from FIDO specifications. This may in some case yield an exponential speed-up at the MONA level.

7 FIDO as a DSL

In our opinion, FIDO is a compelling example of a domain-specific language. It is focused on a clearly defined and narrow domain: *formulas in monadic second-order logic* or, equivalently, *automata on large alphabets*. It offers solutions to a classical software problem: *drowning in a swamp of low-level encodings*. It advocates a simple design principle: *go by analogy to standard programming language concepts*. It uses a well-known and trusted technology: *all the phases of a standard compiler, including optimizations at all levels*. It provides unique benefits that cannot be matched by a library in a standard programming language: *notational conveniences, type checking, and global optimizations*. And during its development, we discovered new insights about the domain: *new notions of tree automata and algorithms*.

References

[1] A. Ayari, D. Basin, and A. Podelski. LISA: A specification language based on WS2S. In *Proceedings of CSL'97*. BRICS, 1997.

[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] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *Proceedings of WIA'96*. Springer Verlag, 1996.

[4] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, August 1986.

[5] Jesper Gulmann Henriksen, Michael Jrgensen, Jakob Jensen, Nils Klarlund, Bob Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Proceedings of TACAS'95, LNCS 1019*, May 1995.

[6] G.J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, May 1997. Special issue on Formal Methods in Software Practice.

[7] J.L. Jensen, M.E. Jrgensen, N. Klarlund, and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of PLDI'97*, 1997.

[8] N. Klarlund, M. Nielsen, and K. Sunesen. A case study in automated verification based on trace abstractions. Technical Report RS-95-54, BRICS, Aarhus University, 1995.

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

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

[11] Nils Klarlund and Theis Rauhe. BDD algorithms and cache misses. Technical Report RS-96-05, BRICS, 1996. Submitted.

[12] J.C. Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development, LNCS 185*, 1985.