A MuDDy Experience–ML Bindings to a BDD Library

Ken Friis Larsen*

Department of Computer Science, University of Copenhagen
Njalsgade 132, DK-2300 Copenhagen S, Denmark
kflarsen@diku.dk

Abstract. Binary Decision Diagrams (BDDs) are a data structure used to efficiently represent boolean expressions on canonical form. BDDs are often the core data structure in model checkers. MuDDy is an ML interface (both for Standard ML and Objective Caml) to the BDD package BuDDy that is written in C. This combination of an ML interface to a high-performance C library is surprisingly fruitful. ML allows you to quickly experiment with high-level symbolic algorithms before handing over the grunt work to the C library. I show how, with a relatively little effort, you can make a domain specific language for concurrent finite state-machines embedded in Standard ML and then write various custom model-checking algorithms for this domain specific embedded language (DSEL).

1 Introduction

The aim of this paper is twofold. Firstly, I show how a simple domain specific language for specifying concurrent finite state machines can be embedded in Standard ML. The combination of a high-level functional language and a BDD package allows us to make some lightweight experiments with model checking algorithms: lightweight in the sense that the approach has a low start-up cost in terms of programmer time. Furthermore, I show that there is no loss of performance using ML compared to using C or C++.

Secondly, I would like to advocate the data structure *binary decision diagrams* (BDDs). BDDs are used to efficiently represent boolean expression on *canonical form*. The canonical form makes it possible to check whether two expressions are equivalent in constant time. Therefore, BDDs are often a core data structure in model checkers and similar tool. BDDs are naturally presented as a persistent abstract data type, hence it is a good fit for functional programming languages. Because first order logic is generally useful as a symbolic modelling language BDDs can also be used for applications other than model checking such as, for instance, program analysis. However, in this paper I concentrate on examples of simple model checking algorithms. To work with BDDs I use our ML interface,

* Supported by the Danish National Advanced Technology Foundation through the 3gERP project.

W.M. Taha (Ed.): DSL 2009, LNCS 5658, pp. 45–57, 2009.

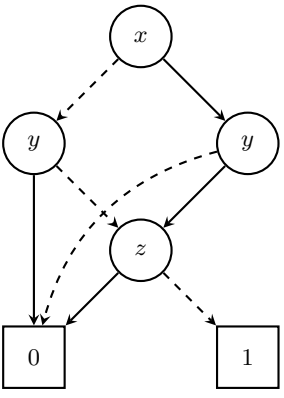

Fig. 1. Example BDD representing $(x \Leftrightarrow y) \wedge \neg z$, with variable ordering $x < y < z$. Dashed edges represent the case where the variable in a node is false and full edges represent the case where the variable is true. The two special nodes 0 (zero) and 1 (one) represents false and true respectively.

MuDDy, to the BDD C library called BuDDy. MuDDy contains both a Standard ML interface (for MLton and Moscow ML) and an Objective Caml interface. Because MuDDy contains both a Standard ML and an Objective Caml interface, I use "ML" when it does not matter which dialect is used.

The remainder of the paper is structured as follows, Section 2 contains a short introduction to BDDs, Section 3 introduces the domain specific language SGCL and shows how it can be embedded in Standard ML, Section 4 shows how to implement some model check algorithms for SGCL, Section 5 compares the performance of a reachable state-space computation programmed in Standard ML, Objective Caml, C++, and C. Finally, Section 6 reviews related work and Section 7 concludes.

2 Binary Decision Diagrams

BDDs are *directed acyclic graphs* (DAGs) used to represent boolean expressions[1]. The main feature of BDDs is that they provide constant time equivalence testing of boolean expressions, and for that reason BDDs are often used in model checkers and theorem provers. The expressions, *exp*, represented by BDDs can be described by the following grammar:

$$\begin{aligned} exp ::= {} & \mathsf{true} \mid \mathsf{false} \mid x \mid exp_1 \ con \ exp_2 \mid \neg exp \\ & \mid \ \exists(x_1, x_2, \ldots, x_n) \ exp \mid \forall(x_1, x_2, \ldots, x_n) \ exp \end{aligned}$$

where *con* is one of the usual boolean connectives, and x ranges over boolean variables, with one unusual requirement: there is a total order of all variables.

Figure 1 shows a graphical representation of a BDD. The important thing to note is that nodes that represent semantically equivalent boolean expressions are always shared.

```
signature bdd =
sig
    type bdd
    type varnum = int
    val TRUE    : bdd
    val FALSE   : bdd
    val ithvar  : varnum -> bdd
    val OR      : bdd * bdd -> bdd
    val AND     : bdd * bdd -> bdd
    val IMP     : bdd * bdd -> bdd
    type varSet
    val makeset  : varnum list -> varSet
    val exist    : varSet -> bdd -> bdd
    val forall   : varSet -> bdd -> bdd
    val equal    : bdd -> bdd -> bool
    ...
end
structure bdd :> bdd = ...
```

Fig. 2. Cut-down version of the Standard ML module `bdd` from MuDDy

2.1 The MuDDy Interface

Figure 2 shows a cut-down version of the Standard ML signature `bdd` from MuDDy (for now we shall ignore initialization of the library, see Section 2.2 for more details), the structure `bdd` implements the signature `bdd`[1]. The only surprising thing in this interface is the representation of boolean variables, the type `varnum`, as integers and not strings as you might have expected. Thus, to construct a BDD that represents x_i (the i'th variable in the total order of variables) you call the function `ithvar` with i as argument.

Using this interface it is now straight forward to build a BDD, `b`, corresponding to the tautology $((x_0 \Rightarrow x_1) \wedge x_0) \Rightarrow x_1$ (assuming that the `bdd` structure has been opened, initialised, and the proper infix declarations have been made):

```
val (x0, x1) = (ithvar 0, ithvar 1)
val b = ((x0 IMP x1) AND x0) IMP x1
```

We can now check that `b` is indeed a tautology. That is, `b` is to (the BDD) `TRUE`:

```
- equal b TRUE;
> val it = true : bool
```

2.2 Motivation and Background for MuDDy

Why does it make sense to interface to a BDD package implemented in C rather than write a BDD package directly in ML? The obvious reason is reuse of code.

[1] It is the standard Moscow ML and Objective Caml convention that a structure `A` implements signature `A` (opaque matching) I use that convention throughout the article.

When somebody already has implemented a good library then there is no reason not to reuse it (if possible).

Secondly, high performance is a necessity (if you want to handle problems of an interesting size) because BDDs are used to tackle a NP complete problem. Even a tiny constant factor has a great impact if it is in an operation that is performed exponentially many times. This is the reason why it is hard to write a high performing BDD package in a high-level programming language.

BDDs are *directed acyclic graphs* where all nodes have the same size. Thus, lots of performance can be gained by implementing custom memory management routines for BDD nodes in C.

The three main advantages of the ML wrapping are:

- The garbage collector in ML takes care of the reference counting interface to BuDDy. Thus, freeing the programmer from this burden (see Section 5.2 to see why this is important).
- Compared to the C interface, the ML API is more strongly typed, which makes it impossible to, say, confuse a set of variables with a BDD (which is represented by the same data structure in BuDDy).
- The interactive environment makes it useful in education, because the students can easily work interactively with BDDs.

The first version of MuDDy was written to support state-of-the-art BDDs in the interactive theorem-prover HOL [2]. This usage has influenced many of decisions about how the interface should be, in particular the design principle that there should be as small an overhead as possible for using MuDDy. Thus, convenience functions such as, for instance, mapping variable numbers to human-readable strings should be implemented outside of the library.

3 Small Guarded Command Language

This section introduces Small Guarded Command Language (SGCL) as an example of a small yet somewhat realistic example of domain specific language (DSL) that can be used for describing concurrent finite state-machines.

Figure 3 shows the grammar for SGCL. An SGCL program consists of a set of boolean variables, an initialisation and a set of concurrent guarded multi-assignments.

$$
\begin{array}{lll}
e & ::= & \mathit{true} \mid \mathit{false} \mid x \mid e_1 \ \mathit{op} \ e_2 \mid \neg e \\
\mathit{assignment} & ::= & x_1, \ldots, x_n \ \texttt{:=} \ e_1, \ldots, e_n \\
\mathit{command} & ::= & e \ \texttt{?} \ \mathit{assignment} \\
\mathit{program} & ::= & \mathit{assignment} \\
 & & \mathit{command}_1 \ \texttt{||} \ \ldots \ \texttt{||} \ \mathit{command}_n
\end{array}
$$

Fig. 3. Grammar for SGCL

```
type var = string
datatype boolop = AND | OR | IMP | BIIMP
datatype bexp = BVar of var
              | BBin of bexp * boolop * bexp
              | NOT  of bexp
              | TRUE | FALSE
datatype command = CMD of bexp * (var * bexp) list
datatype program = PRG of (var * bexp) list * command list
```

Fig. 4. Abstract syntax for SGCL in Standard ML

```
fun mkBBin opr (x, y) = BBin(x, opr, y)
infix /\ \/ ==> <==>
val (op /\, op \/, op ==>, op <==>) =
    (mkBBin AND, mkBBin OR, mkBBin IMP, mkBBin BIIMP)
infix ::=
val op ::= = ListPair.zip
infix ?
fun g ? ass = [CMD(g, ass)]
infix ||
val op|| = op@
val $ = BVar
```

Fig. 5. Syntactic embedding of SGCL in Standard ML

3.1 SGCL Embedded in Standard ML

To embed SGCL in Standard ML we need two parts: a representation of the abstract syntax of SGCL and an embedding of a concrete syntax for SGML. Figure 4 shows the straightforward representation of abstract syntax trees for SGCL in Standard ML.

For the syntactic embedding we make heavy use of Standard ML's `infix` declaration. Figure 5 shows the few declarations we need to make to get a much nicer syntax for SGCL rather that the raw abstract syntax trees. The biggest wart is that we need to distinguish between variables occurring on the left-hand side of an assignment and variables occurring in expressions. We use `$` to lift a variable to an expression.

Thus, if we have a command that monitors when two variable, v_1 and v_2, are both true and then set them both to false, and a command that does the reverse:

$$\begin{array}{r} v_1 \wedge v_2 \texttt{ ? } v_1, v_2 \texttt{:=} \mathit{false}, \mathit{false} \\ \texttt{|| } \neg v_1 \wedge \neg v_2 \texttt{ ? } v_1, v_2 \texttt{:=} \mathit{true}, \mathit{true} \end{array}$$

then these two commands can now be written as the following Standard ML expression:

```
   ($"v1" /\ $"v2")           ? (["v1", "v2"] ::= [TRUE, TRUE])
|| (NOT($"v1") /\ NOT($"v2")) ? (["v1", "v2"] ::= [FALSE, FALSE])
```

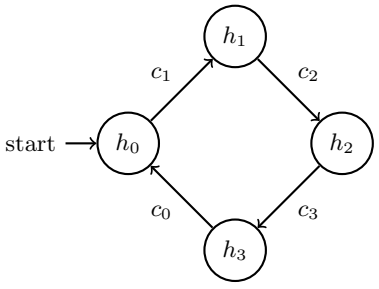

Fig. 6. Milner's Scheduler with four cyclers

Aside from minor warts like the annoying quotation marks and dollar signs, I consider this to be fully acceptable syntax for SGCL programs. It is definitely nicer to read than the LaTeX encoding of the SGCL program.

3.2 Example: Milner's Scheduler

The driving example in this section is Robin Milner's scheduler taken from [3], the modelling is loosely based on the modelling by Reif Andersen found in [4]. Milner's scheduler is a simple protocol: N cyclers cooperate on starting N tasks, one at a time. The cyclers use a token to ensure that all tasks are started in order.

Figure 6 shows Milner's Scheduler with four cyclers. Each of the cyclers i has three Boolean c_i, h_i and t_i. The Boolean c_i is used to indicate that the token is ready to be picked up by cycler i, the Boolean h_i indicates that cycler i has picked up the token and is holding the token, and t_i indicates that task i is running.

The initial state of the system is that the token is ready to be picked up by cycler 0 and all tasks are stopped. This can be represented by the following predicate:

$$c_0 \wedge \neg h_0 \wedge \neg t_0 \wedge \Big(\bigwedge_{i \in 1 \ldots (N-1)} \neg c_i \wedge \neg h_i \wedge \neg t_i \Big).$$

When a task is not running, and the token is ready, the cycler can pick up the token and start the task. The cycler can then pass on the token to the next cycler, and the task can stop. For a cycler i this can be described using two guarded commands:

$$\begin{aligned} \mathit{cycler}_i \;=\; & c_i \wedge \neg t_i \;\texttt{?}\; t_i, c_i, h_i \;\texttt{:=}\; \mathit{true}, \neg c_i, \mathit{true} \\ \texttt{||} \quad & h_i \;\texttt{?}\; c_{i+1 \bmod N}, h_i \;\texttt{:=}\; \mathit{true}, \mathit{false} \end{aligned}$$

Furthermore we need to model the environment that starts and ends task. This is simply done with one command per task that simply monitors when the right t-variable becomes true and then sets it to false. That is, for task i:

$$\mathit{task}_i = t_i \;\texttt{?}\; t_i \;\texttt{:=}\; \mathit{false}$$

```
val (c0, t0, ... , t3, h3) = ("c0", "t0", ... ,"t3", "h3")

fun cycler c t h c' =
      ($c /\ NOT($t)) ? ([t, c, h]  ::= [TRUE, NOT($c), TRUE])
   || ($h ? ([c', h] ::= [TRUE, FALSE]))

fun task t = $t ? ([t] ::= [FALSE])

val milner4 =
    PRG( [(c0, TRUE), (t0, FALSE), (h0, FALSE), ... ],
         cycler c0 t0 h0 c1
      || cycler c1 t1 h1 c2
      || cycler c2 t2 h2 c3
      || cycler c3 t3 h3 c0
      || task t0  ||  task t1
      || task t2  ||  task t3
       )
```

Fig. 7. Standard ML program that show Milner's scheduler with four cyclers

All of this can be translated to Standard ML in a straightforward manner. Figure 7 shows the Standard ML code required to model Milner's Scheduler with four cyclers. The code in Figure 7 is hardwired to four cyclers. However, it is trivial to generalise the code so that it generates a scheduler for arbitrary N.

4 Symbolic Execution of SGCL Programs

One thing that we can do with an SGCL program is to execute it symbolically to find the set of all reachable states from the initial state. That is, we want to compute some kind of representation that denotes the set of all states. One choice is to give an explicit enumeration of all the states. However, for an SGCL program with only a moderate number of variables this representation will quickly explode to enormous sizes. Instead we shall use BDDs to represent the set of reachable states and also to perform the symbolic execution.

The first step is to transform an SGCL program into a pair that consists of a BDD that represents the initial state and a BDD that represents the possible transitions from one set of states to the following set of states (post-states). To model these transitions we shall use two BDD variables, one unprimed and one primed, for each SGCL variable. The primed variables are used to model post-states.

Thus, the interesting part is that we must translate each SGCL command to a predicate, represented as a BDD, that relates a set of states to their post-states. This predicate consists of two parts, one for all the unchanged variables, the variables not assigned in the assignment, that just say that the unchanged variables in the post-state are equivalent to their values in the current state. The other part of predicate is for the variables in the assignment. Figure 8

```
(* commandToBDD: (string * {var: bdd.varnum, primed: bdd.varnum}) list
               -> command
               -> bdd.bdd
*)
fun commandToBDD allVars (CMD(guard, assignments)) =
    let val changed = List.map #1 assignments
        val unchanged =
            List.foldl (fn ((v, {var, primed}), res) =>
                           if mem v changed then res
                           else bdd.AND(bdd.BIIMP(bdd.ithvar primed,
                                                  bdd.ithvar var),
                                        res))
                       bdd.TRUE allVars
        val assigns =
            List.foldl (fn ((v,be), res) =>
                           bdd.AND(bdd.BIIMP(primed v, bexp be),
                                   res))
                       bdd.TRUE assignments
    in bdd.IMP(bexp guard, assigns) end
```

Fig. 8. The function `commandToBDD` translates an SGCL command to a BDD. It uses the helper functions `bexp` that translates an SGCL expression to a BDD and `primed` that translates an SGCL variable to the corresponding primed BDD variable.

shows the function `commandToBDD` that takes a mapping, here as a simple list, of all SGCL variables to the corresponding BDD variable numbers and a command and translates it to a BDD.

Once we have defined `commandToBDD` it's straightforward to define a function, `programToBDDs`, that translates an SGCL program to the two desired BDDs:

```
fun programToBDDs allVars (PRG(init, commands)) =
    let val init =
            List.map (fn (v,be) => bdd.BIIMP(var v, bexp be) init
    in (conj init, disj(List.map commandToBDD commands)) end
```

Again I use some helper functions: `var` translates an SGCL variable to the corresponding unprimed BDD variable, `bexp` translates an expression to a BDD, `conj` makes a conjunction of a list of BDDs, and `disj` makes a disjunction of a list of BDDs.

Now we are ready to compute the set of all reachable states for an SGCL program. Figure 9 shows the function `reachable` that computes the set of reachable states, represented as a BDD, given an initial state, `I`, and a predicate, `T`, relating unprimed variables to primed variables. The interesting part is the nested function `loop` that repetitively expands the set of reachable states until a fixpoint is found. This kind of algorithm is one of the cornerstones in most model checkers.

Once we have computed the set of reachable states, it is easy to check whether, for instance, they all satisfy a certain invariant. We just check that the BDD

```
fun reachable allVars I T =
    let val renames  =
            List.map (fn(_,{var,primed}) => (var, primed)) allVars
        val pairset  = bdd.makepairSet renames
        val unprimed = bdd.makeset(List.map #var allVars)

        open bdd infix OR
        fun loop R =
            let val post = appex T R And unprimed
                val next = R OR replace next pairset
            in  if equal R next then R else loop next end
    in  loop I end
```

Fig. 9. Compute the set of all reachable states for a SGCL program

representing the invariant imply the BDD representing the set of reachable states. Thus, if `R` is the BDD representing the set of reachable states and `Inv` is the BDD representing the invariant we want to check, then we compute:

```
val invSatisfied = bdd.IMP Inv R
```

And if `invSatisfied` is equal to `bdd.TRUE` we know that all reachable states satisfies the invariant.

5 ML versus C++ versus C

In this section we will compare model checking using MuDDy with more traditional approaches such as using a C++ or C library directly. We will show fragments of code and we will measure the performance of the different alternatives. Again we use Milner's scheduler as example.

Model checking is generally NP-complete (due to the exponential number of states). However, BDDs seems to be an efficient representation of the predicates used in Milner's scheduler. In fact, it turns out that the size of the representation of the state space "only" grows polynomially in the number of cyclers rather than exponentially.

We shall compare implementation in C, C++, Standard ML, and Objective Caml. So as to show what modelling with BDDs looks like in the different languages, I show how to construct the BDD for initial state of Milner's scheduler in C++ and C. Finally, I show running times for the different languages.

5.1 C++

In this section we show a C++ implementation using the BuDDy C++ library. The BuDDy C++ library defines a class `bdd` that represent a bdd, similar to `bdd.bdd` in MuDDy. The class defines bunches of smart operators that makes building Boolean expressions quite easy. Furthermore, all reference counting is managed by the constructor and destructor for the `bdd` class.

```
bdd initial_state(bdd* t, bdd* h, bdd* c)
{
   int i;
   bdd I = c[0] & !h[0] & !t[0];

   for(i=1; i<N; i++)
      I &= !c[i] & !h[i] & !t[i];

   return I;
}
```

The code shows that it is possible to embed a DSEL for boolean expression quite nicely in C++.

5.2 C

The C code for building the initial state space predicate lacks the constructor/destructor mechanisms for managing the reference counting. Thus, you must manually keep track of calling the reference counting functions. From hard-earned experience, I guarantee that the code is error prone, unreadable and really hard to maintain. The function `bdd_addref` references a BDD and then returns the BDD and similar for `bdd_delref`.

```
bdd initial_state(bdd* t, bdd* h, bdd* c)
{
   int i;
   bdd I, tmp1, tmp2, tmp3;

   tmp1 = bdd_addref( bdd_not(h[0]) );

   tmp2 = bdd_addref( bdd_apply(c[0], tmp1, bddop_and) );
   bdd_delref(tmp1);

   tmp1 = bdd_addref( bdd_not(t[0]) );

   I = bdd_apply(tmp1, tmp2, bddop_and);
   bdd_delref(tmp1);
   bdd_delref(tmp2);


   for(i=1; i<N; i++)
   {
      bdd_addref(I);

      tmp1 = bdd_addref( bdd_not(c[i]) );
      tmp2 = bdd_addref( bdd_not(h[i]) );

      tmp3 = bdd_addref( bdd_apply(tmp1, tmp2, bddop_and) );
      bdd_delref(tmp1);
```

```
      bdd_delref(tmp2);

      tmp1 = bdd_addref( bdd_not(t[i]) );
      tmp2 = bdd_addref( bdd_apply(tmp3, tmp1, bddop_and) );
      bdd_delref(tmp3);
      bdd_delref(tmp1);

      tmp1 = bdd_apply(I, tmp2, bddop_and);
      bdd_delref(tmp2);
      bdd_delref(I);

      I = tmp1;
   }

   return I;
}
```

5.3 Performance

In the following we benchmark the ML implementations of Milner's scheduler against their C and a C++ counterparts. As benchmark we compute the number of reachable states in Milner's scheduler for a given number of cyclers.

The source programs were obtained as follows: The C and the C++ implementations are very close to example programs provided with the BuDDy library. We have modified the programs slightly to get them as equal as possible and thereafter translated them into Standard ML code and Objective Caml code. (All implementations are included in the (upcoming) MuDDy distribution.)

You will recall that even though the number of reachable states is exponential in the number of schedulers, it turns out that the representation is only polynomial using BDDs.

Table 1 shows the running times for running the programs with different numbers of cyclers. As we can see, the running times for the ML programs are only *slightly* slower than the C and C++ programs. This is expected because most of the time in all of the programs is spent in the BuDDy library. Yet, it

Table 1. Milner's scheduler benchmark. Running time is in seconds, measured on a 1 Ghz PII with ulimit 156 Mb. The C and C++ programs was compiled using gcc 3.3 with flag `-O3`, *ocamlopt* is native-code from Objective Caml 3.04, *ocaml* is byte-code from Objective Caml 3.04, and *sml* is byte-code from Moscow ML 2.01.

No. of cyclers:	50	100	150	200
c :	1.63	4.69	13.66	31.20
cxx :	1.66	4.82	13.89	31.51
ocamlopt:	1.71	5.04	14.47	32.05
ocaml :	1.74	5.15	14.58	32.91
sml :	1.76	5.15	15.12	33.42

is reassuring to see that we can write high-level algorithms, such as finding the reachable state-space, in ML without being overly penalised.

6 Related Work

Combining BDDs and functional languages is not a new idea. For instance, Lifted-FL [5] is a domain-specific, strongly typed, lazy, functional programming language, where BDDs are available as a build-in type. Lifted-FL is used as a combined model-checking and theorem-proving tool at Intel SCL Logic Team. And the successor to Lifed-FL, the verification framework Forte[6], is built around the strongly-typed ML-like language FL. Like in Lifted-FL BDDs are built into the language, and every Boolean object is represented as a BDD.

Nancy Day et al [7] shows how BDDs and SAT solvers can be nicely wrapped in Haskell so that it feels like a native Haskell structure, hiding the mucky details of an underlying C library–similar to how MuDDy wraps BDDs in ML.

Once you have a nice wrapping of BDDs in a high-level language, you can leverage it to solve a variety of problems. Some examples that have used MuDDy are: Sørensen and Secher show how one can efficiently solve configuration problems with BDDs [8] and Sittampalam et al [9] show how BDDs, for instance, can be used for program transformations. In general, many model checking algorithms should be useful for program analysis [10].

7 Conclusion

The combination of a high-level modern language that enables DSEL with a powerful data structure like BDDs makes a pleasant environment in which to learn and experiment, for example model checking algorithms and language design. As an experiment it would be easy, for instance, to add priority specifications to SGCL.

The fact that we have not made a new domain-specific language with BDDs built into it, but have merely made an ML library, means that we can leverage all the existing tools and libraries that exist for Standard ML and Objective Caml. For instance, it enabled us to embed SGCL into Standard ML without problems.

An interesting possibility for the future would be to make an F# [11] interface for MuDDy, because F# has recently acquired powerful reflection mechanisms inspired by *reFL*^{ect} [12] (which in turn was inspired by FL). Thus, it should be possible to reap more of the advantages from FL while still staying within a general purpose language.

Acknowledgments and MuDDy History

The first version of MuDDy was written by Ken Friis Larsen while visiting Mike Gordon at Computer Lab. at University of Cambridge (UK), in autumn 1997 and spring 1998. Jakob Lichtenberg then extended MuDDy to cope with the new

BuDDy features: Finite Domain Blocks (fdds) and Boolean Vectors (bvecs). In 2001–2002 we added support for Objective Caml. In 2003, Ooege de Moor at Computer Lab. at University of Oxford sponsored a research visit to Oxford for Ken Friis Larsen which, amongst other things, resulted in the MLton binding.

It should be stressed that MuDDy is only a type safe ML wrapping around Jørn Lind-Nielsen's great BuDDy package, and that all the "hard work" is done by the BuDDy package. Jørn Lind-Nielsen has answered lots of BuDDy questions (and BDD questions), and has been willing to change the BuDDy package to make the ML wrappings more easier.

A special thanks should also go to Peter Sestoft, who has answered numerous of questions about the Moscow ML/Caml Light runtime system, and he even audited the C code at one point to help find a bug.

In addition, over the years many MuDDy users have provided bug-reports, bug-fixes, and useful feedback.

References

1. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys 24(3), 293–318 (1992)
2. Norrish, M., Slind, K.: A thread of HOL development. Computer Journal 45(1), 37–45 (2002)
3. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1989)
4. Andersen, H.R.: An introduction to binary decision diagrams (1997), `http://www.itu.dk/people/hra/bdd97-abstract.html`
5. Aagaard, M.D., Jones, R.B., Seger, C.J.H.: Lifted-FL: A pragmatic implementation of combined model checking and theorem proving. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, p. 323. Springer, Heidelberg (1999)
6. Jones, R.B., O'Leary, J.W., Seger, C.J.H., Aagaard, M.D., Melham, T.F.: Practical formal verification in microprocessor design. IEEE Design & Test of Computers 18(4), 16–25 (2001)
7. Day, N.A., Launchbury, J., Lewis, J.: Logical abstractions in Haskell. In: Proceedings of the 1999 Haskell Workshop, Utrecht University Department of Computer Science, Technical Report UU-CS-1999-28 (October 1999)
8. Sørensen, M.H., Secher, J.P.: From type inference to configuration. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 436–472. Springer, Heidelberg (2002)
9. Sittampalam, G., Moor, O.D., Larsen, K.F.: Incremental execution of transformation specifications. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 26–38. ACM Press, New York (2004)
10. Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998)
11. Syme, D., Granicz, A., Cisternino, A.: Expert F#. Apress (2007)
12. Grundy, J., Melham, T., O'Leary, J.: A reflective functional language for hardware design and theorem proving. Journal of Functional Programming 16(2), 157–196 (2006)