`HolBddLib` Version 2 Documentation

Mike Gordon

June 7, 2002

Preface

The development of `HolBddLib` has gone through two phases. The first phase consisted in experiments with different ways of linking higher order logic (HOL) terms to binary decision diagrams (BDDs). These are described in the paper *Reachability programming in HOL98 using BDDs* [6]. The first release of `HolBddLib`, now called Version 1, consisted of an ad hoc collection of tools developed for these experiments. One of the approaches we experimented with was based on a protected type of 'BDD representation judgements', analogous to the LCF protected type of theorems. Positive results of Hasan Amjad [3] have lead us to narrow attention to just this approach. `HolBddLib` Version 2, which is described here, provides a set of representation judgement rules as core infrastructure for building 'fully-expansive' or 'LCF-style' combinations of HOL theorem proving and BDD-based symbolic calculation algorithms. All higher level tools, such as model checkers, are programmed in ML as 'derived rules'.

The primitive inference rules for representation judgements are in the structure `PrimitiveBddRules`. A few example derived rules are in the structure `DerivedBddRules`. Currently the only derived rules are ones to compute reachable states and find sequences of transitions to states with given properties. It is hoped to soon add a module for checking properties expressed in the modal μ-calculus (and hence CTL).

Version 1 of `HolBddLib` was more elaborate than Version 2 because it mixed together code from a number of experiments. In Version 1 there was a function, called `termToBdd`, that tried to represent a HOL term as a BDD using a dynamically extendable global table mapping HOL terms to BDDs. `TermToBdd` constructed the BDD of a term t using any BDDs of subterms of t that were stored in the global table. `HolBddLib` Version 2 has jettisoned this imperative style in favour of purely functional rules. Some of the ideas of BDD tables are likely to return in the future, but as contexts, similar to HOL simpsets, that are passed functionally, rather than as a single global state held in references.

`HolBddLib` Version 1 only supported a single variable ordering, held in a global variable map. In Version 2, each representation judgement carries its own variable ordering, so that local scopes are possible. For convenience, `DerivedBddRules` provides a way of storing a default variable ordering in a global variable, but this is just a derived facility, not part of the kernel.

`HolBddLib` Version 2 adds assumptions to representation judgements analogous to assumptions of HOL theorems. This enables Coudert, Berthet and Madre simplification to be represented as a primitive rule (see the rule `BddSimplify` in Section 17.1). It also allows the term part of a representation judgements to be simplified using equations with assumptions (see the rule `BddEqMp` in Section 17.2).

`HolBddLib` uses Jørn Lind-Nielsen's BuDDy package as a BDD engine. The interface from BuDDy to Moscow ML, called MuDDy, is due to Ken Friis Larsen and Jakob Lichtenberg, and is described in Part I. `HolBddLib` is built on top of MuDDy and is described in Part II.

Some of the material in this document derives from University of Cambridge Computer Laboratory Technical Report No. 481, December 1999, by Mike Gordon and Ken Friis Larsen [7]. Although this report has examples that might be of tutorial use, it has much obsolete material and methodology deriving for early experiments pre-dating the release of `HolBddLib` Version 1.

Overview

In the fully expansive (or 'LCF style') approach, theorems are represented by an abstract type whose primitive operations are the axioms and inference rules of a logic. Theorem proving tools are implemented by composing together the inference rules using ML programs.

This idea can be generalised to computing valid judgements that represent other kinds of information. In particular, consider judgements (a, ρ, t, b), where a is a set of boolean terms (assumptions) that are assumed true, ρ represents a variable order, t is a boolean term all of whose free variables are boolean and b is a BDD. Such a judgement is valid if under the assumptions a, the BDD representing t with respect to ρ is b, and we will write $a\ \rho\ t\ \mapsto\ b$ when this is the case.

The derivation of 'theorems' like $a\ \rho\ t\ \mapsto\ b$ can be viewed as 'proof' in the style of LCF by defining an abstract type `term_bdd` that models judgements $a\ \rho\ t\ \mapsto\ b$ analogously to the way the type `thm` models theorems $\vdash t$.

`HolBddLib` currently contains two main structures: `PrimitiveBddRules` which defines a protected type `term_bdd` and rules for generating values of this type, and `DerivedBddRules` that contains derived rules for performing simple fixed-point calculations. There is also a theory `MachineTransitionTheory` containing the theorems on reachability and fixed points needed by the derived rules, and two small subsidiary structures `Varmap` and `PrintBdd`.

Relation to the Voss system[1]

The Voss system [13] has strongly influenced and inspired the ideas described here. Voss consists of a lazy ML-like functional language, called FL, with BDDs as a built-in datatype. Quantified boolean formulae can be input and are parsed to BDDs. The normal boolean operations $\neg$, $\wedge$, $\vee$, $\equiv$, $\forall$, $\exists$ are interpreted as BDD operations. Algorithms for model checking are easily programmed.

Joyce and Seger interfaced an early HOL system (HOL88) to Voss and in a pioneering paper showed how to verify complex systems by a combination of theorem proving deduction and symbolic trajectory evaluation (STE) [9]. The HOL-Voss system integrates HOL88 deduction with BDD computations. BDD tools are programmed in FL and can then be invoked by HOL-Voss tactics, which can make external calls into the Voss system, passing subgoals via a translation between the HOL88 and Voss term representations.

In later work Lee, Seger and Greenstreet [10] showed how various optimised BDD algorithms could be programmed in FL.

The early experiments with HOL-Voss suggested that a lighter theorem proving component was sufficient, since all that was really needed was a way of combining results obtained from STE. A system based on this idea, called VossProver, was developed by Carl Seger and his student Scott Hazelhurst. It provides operations in FL for combining assertions generated by Voss using proof rules corresponding to the laws of composition of the temporal logic assertions verified by STE [8]. VossProver was used to verify impressive integer and floating-point examples (see the DAC98 paper by Aagaard, Jones and Seger [1] for further discussion and references).

After Seger and Aagaard moved to Intel, the development of the Voss and VossProver systems evolved into a new system called Forte. Only partial details of this are in the public domain [12, 2], but a key idea is that FL is used both as a specification language and as an LCF-style metalanguage. The connection between symbolic trajectory evaluation and proof is obtained via a tactic `Eval_tac` that converts the result of executing an FL program

[1]Adapted from *Reachability programming in HOL using BDDs* [6]

performing STE into a theorem in the logic. Theorem proving in Forte is used both to split goals into smaller subgoals that are tractable for model checking, and to transform formulae so that they can be checked more efficiently.

The combination of HOL and BuDDy in Version 1 of `HolBddLib` provides a somewhat similar programming environment to Voss's FL (though with eager rather than lazy evaluation and no special support for STE). BuDDy provides BDD operations corresponding to $\neg$, $\wedge$, $\vee$, $\equiv$, $\forall$, $\exists$ and the HOL term parser plus `termToBdd` provides a way of using these to create BDDs from logical terms. Voss enables efficient computations on BDDs using functional programming. So does `HolBddLib`. However, in addition it allows FL-like BDD programming in ML to be intimately mixed with HOL deduction, so that, for example, theorem proving tools (e.g. simplifiers) can be directly applied to terms to optimise them for BDD purposes (e.g. disjunctive partitioning). This is in line with future developments discussed by Joyce and Seger [9] and it appears that the Forte system has similar capabilities.

`HolBddLib` Version 2 provides a less developed interactive programming environment than Version 1. It is more oriented to providing a clean and simple API allowing implementers to create their own 'fully-expansive' combinations of model checking and theorem proving. Such a combination could be a Voss-like verification platform.

Contents

Part I
MuDDy

MuDDy is the Moscow ML interface to BuDDy. It provides ML functions for constructing and manipulating BDDs via three structures:

- `bdd` defines the ML type `bdd` representing BDDs and associated operations derived from BuDDy;
- `fdd` provides support for blocks of BDD variables used to encode values representing elements of finite domains;
- `bvec` provides support for Boolean vectors.

The current `HolBddLib` system only uses `bdd` and so the documentation of `fdd` and `bvec` provided here is minimal (see Sections 12 and 13 below).

1 Initialisation, termination and tuning sessions

The BuDDy package must be initialised before any BDD operations are done. Initialisation is done with the ML function

```
init : int -> int -> unit
```

Evaluating `init` m n initialises BuDDy with m nodes in the nodetable and a cachesize of n. The library `HolBddLib` (Part II) initialises the nodetable to 1000000 and cachesize to be 10000. The following is a quotation from the BuDDy documentation [11].

> Good initial values are
>
> | **Example** | **nodenum** | **cachesize** |
> |---|---:|---:|
> | Small test examples | 1000 | 100 |
> | Small examples | 10000 | 1000 |
> | Medium sized examples | 100000 | 10000 |
> | Large examples | 1000000 | 10000 |

> Too few nodes will only result in reduced performance and this increases the number of garbage collections needed. If the package needs more nodes, then it will automatically increase the size of the node table.

The initial number of nodes is not critical for any BDD operation as the table will be resized whenever there are too few nodes left after a garbage collection. But it does have some impact on the efficiency of the operations.

The function

```
done : unit -> unit
```

frees all memory used by BuDDy and resets the package to its initial state.

The functions `init` and `done` should only be called once per session.

The function

```
isRunning : unit -> bool
```

tests whether BuDDy is running (i.e. `init` has been called and `done` has not been called). It is useful for checking if initialialisation is needed.

The functions `init` and `done` should only be called once in a session.

Statistical information from BuDDy is available using the function `stats`

```
stats : unit -> {produced     : int,
                 nodenum      : int,
                 maxnodenum   : int,
                 freenodes    : int,
                 minfreenodes : int,
                 varnum       : int,
                 cachesize    : int,
                 gbcnum       : int}
```

The meaning of the values of the various named fields in the record returned by evaluating `stats()` are

Field name	Meaning
`produced`	total number of new nodes ever produced
`nodenum`	currently allocated number of BDD nodes
`maxnodenum`	user defined maximum number of BDD nodes
`freenodes`	number of currently free BDD nodes
`minfreenodes`	minimum number of nodes left after a BDD garbage collection
`varnum`	number of defined BDD variables
`cachesize`	number of cache entries
`gbcnum`	number of BDD garbage collections done

The management of the node table and internal caches can be tuned using the following functions

```
setMaxincrease : int -> int
setCacheratio  : int -> int
```

Evaluating `setMaxincrease` n tells BuDDy that the maximum of new nodes added when doing an expansion of the nodetable should be n. The previous maximum is returned.

Evaluating `setCacheratio` n sets the cache ratio to n. For example, if n is 4 then the internal caches will a quarter the size of the nodetable.

2 BDDs representing `true` and `false`

The atomic BDDs representing the two truthvalues are bound to the ML identifiers `TRUE` and `FALSE`, both of type `bdd`.

Functions for mapping from ML Booleans to BDDs and vice versa are, respectively

```
fromBool : bool -> bdd
toBool   : bdd  -> bool
```

The function `toBool` returns `true` on TRUE and `false` on FALSE. It raises the exception `Domain` on non-atomic BDDs.

```
equal : bdd -> bdd -> bool
```

tests the equality of two BDDs. Thus `TRUE` is `equal` to `fromBool(true)` and `FALSE` is `equal` to `fromBool(false)`.

3 Variables

In BuDDy, BDD variables are encoded as integers (type `int` in ML) and the BDD variable ordering is the numerical ordering. Thus to build a BDD to represent a HOL term with a particular variable ordering it is necessary to map HOL variables to integers so that the numerical order corresponds to the desired variable order.

The number of variables in use must be declared using

```
setVarnum : int -> unit
```

Evaluating `setVarnum` n declares that the n variables 0, 1, ... , $n-1$ are available for use. The number of variables can be increased dynamically during a session by calling `setVarnum` with a larger number. The number of variables cannot be decreased dynamically. The function

```
getVarnum : unit -> int
```

returns the number of variables in use (i.e. the argument of the last application of `setVarnum`).

The function

```
ithvar : int -> bdd
```

maps an ML integer to a BDD that consists of just the variable corresponding to the integer and

```
nithvar : int -> bdd
```

maps an integer to the BDD representing the negation of the variable.

Note that evaluating `ithvar` n or `nithvar` n will raise the exception `Fail` (with string argument `"Unknown variable"`) if n has not been declared as in use, i.e. if `setVarnum` m has not been previously evaluated for some m greater than n.

4 Sets of variables and quantification

BuDDy provides operations on BDDs for quantifying with respect to sets of variables. The module `bdd` provides a type `varSet` to represent such sets with, respectively, a constructor and two destructors:

```
makeset : int list -> varSet
scanset : varSet   -> int vector
fromSet : varSet   -> bdd
```

The destructor `scanset` returns a vector of the variables in the set and the destructor `fromSet` returns a BDD representing the conjunction of the variables in the set.

The following functions quantify BDDs with respect to sets of variables:

```
forall : varSet -> bdd -> bdd
exist  : varSet -> bdd -> bdd
```

5 Assignments, composition, replacement and restriction

MuDDy provides a function for general purpose simultaneous substitution of arbitrary BDDs for variables in a given BDD (`veccompose`). It also provides and three optimised special cases: substituting for a single variable (`compose`), renaming variables (`replace`) and substituting with boolean constants (`restrict`).

The operation `veccompose` performs the simultaneous substitution of BDDs for variables in a BDD. The argument of `veccompose` is a value of type `composeSet` (created with a constructor `composeSet`) that specifies a list if pairs `[`$((n_1,b_1),\ldots,$, where BDD variable n is to be pre

```
composeSet : (int * bdd) list -> composeSet
veccompose : composeSet -> bdd -> bdd
```

A single variable can be replaced with a BDD using

```
compose : bdd -> bdd -> int -> bdd
```

Evaluating `compose` b_1 b_2 n substitutes b_2 for the variable n in b_1.

Variables can be renamed using the function `replace` that takes an argument of type `pairSet` representing sets of pairs of variables (with constructor `makepairSet`)

```
makepairSet : (int * int)list -> pairSet
replace     : bdd -> pairSet -> bdd
```

Evaluating `makepairSet[(`x_1`,`x'_1`), ... , (`x_n`,`x'_n`)]` creates a set of pairs specifying that x'_i be substituted for x_i (for $1 \leq i \leq n$). A renaming with `replace` will fail if it would result in distinct variables being identified (i.e. if the shape of the BDD would change).

BDDs can be restricted by instantiating variables to `TRUE` or `FALSE` using the function `restrict` that takes as argument a value of type `assignment` (which has a constructor `assignment` and destructor `getAssignment`).

```
assignment    : (int * bool)list -> assignment
getAssignment : assignment -> (int * bool) list
restrict      : bdd -> assignment -> bdd
```

Evaluating `assignment[(`v_1`,`t_1`), ... , (`v_n`,`t_n`)]` creates an assignment specifying that each v_i be instantiated to `fromBool(`t_i`)` (for $1 \leq i \leq n$).

6 Finding satisfying assignments

An assignment satisfying a BDD can be computed via BuDDy using

```
satone : bdd -> assignment
```

The exception `Domain` is raised if the argument to `satone` is unsatisfiable.

Alternatively, a model can be computed by an ML program such as:

```
val findSat =
 let fun findSatAux bdd =
      if bdd.equal bdd bdd.TRUE
       then []
       else
        if bdd.equal bdd bdd.FALSE
         then raise Domain
         else
          ((bdd.var bdd,true) :: findSatAux(bdd.high bdd)
           handle Domain =>
           (bdd.var bdd, false) :: findSatAux(bdd.low bdd))
 in
  assignment o findSatAux
 end;
```

The functions `satone` and `findSat` do not necessarily find the same satisfying assignment, if more than one exists. Also, `findSat` stops when it has found enough variable bindings to satisfy the BDD, so may not return an assignment giving values to all the variables.

7 Boolean operations on BDDs

The structure `bdd` introduces a type `bddop` corresponding to Boolean operations on BDDs. The ML function

```
apply : bdd -> bdd -> bddop -> bdd
```

applies a BDD operation to BDD values.

BuDDy provides functions for calculating in a single step the result of performing a Boolean operation and then quantifying the result with respect to several variables.

```
appall : bdd -> bdd -> bddop -> varSet -> bdd
appex  : bdd -> bdd -> bddop -> varSet -> bdd
```

The function `appall` universally quantifies the result of the Boolean operation and `appex` existentially quantifies it.

MuDDy provides ten operations of type `bddop` and for each of these an ML infix, pre-defined using `apply`, of type `bdd * bdd -> bdd`.

`bddop`	`bdd * bdd -> bdd`	Result of applying to (b_1, b_2)
`And`	`AND`	$b_1 \wedge b_2$
`Nand`	`NAND`	$\neg(b_1 \wedge b_2)$
`Or`	`OR`	$b_1 \vee b_2$
`Nor`	`NOR`	$\neg(b_1 \vee b_2)$
`Biimp`	`BIIMP`	$b_1 = b_2$
`Xor`	`XOR`	$\neg(b_1 = b_2)$
`Imp`	`IMP`	$b_1 \Rightarrow b_2$
`Invimp`	`INVIMP`	$b_2 \Rightarrow b_1$
`Lessth`	`LESSTH`	$\neg b_1 \wedge b_2$
`Diff`	`DIFF`	$b_1 \wedge \neg b_2$

MuDDy also provides a unary negation operator and ternary conditional operator.

```
NOT : bdd -> bdd
ITE :  bdd -> bdd -> bdd -> bdd
```

`NOT` b is the BDD corresponding to '$\neg b$' and `ITE` b b_1 b_2 is the BDD corresponding to '*if* b *then* b_1 *else* b_2'.

8 Inspecting and counting nodes and states

The integer labelling a BDD node and the BDDs corresponding to the high (i.e. `true`) and low (i.e. `false`) nodes are obtained, respectively, with

```
var  : bdd -> int
high : bdd -> bdd
low  : bdd -> bdd
```

Thus if b is the BDD of "*if* x *then* t_1 *else* t_2" then `var` b will return the number representing variable x, `high` b will return the BDD of t_1 and `low` b will return the BDD of t_2.

Note that `var`, `high` and `low` raise an exception if applied to `TRUE` or `FALSE`.

The entire BuDDy node table of a BDD can be copied into ML using

```
nodetable : bdd -> int * (int * int * int)vector
```

The integer returned as the first component of the pair is a pointer (starting from 0) into the second component, a vector of node descriptors. This pointer points to the root node. Each node descriptor is a triple of integers (v, l, h), where v is the node label (i.e. a number representing a variable), l points to the low (`false`) node in the vector and h points to the high (`true`) node. The first two nodes in the vector are special: they represent `true` and `false`, respectively, and arbitrarily have the structure $(0, 0, 0)$.

The number of nodes in a BDD is computed by the function

```
nodecount : bdd -> int
```

This could be defined by

```
fun nodecount bdd = Vector.length(snd(nodetable bdd));
```

However, `nodecount` defined this way is likely to run out of space on large BDDs (since it involves copying the argument BDD from BuDDy's representation into an ML vector). Thus the ML function provided by MuDDy invokes BuDDy's `nodecount` function directly and so is space-efficient.

The number of assignments *to all variables in use in the current session* that satisfy a BDD (i.e. make it true) is given by the ML function

```
satcount : bdd -> real
```

The answer is exact until the result is too big to be represented as a Moscow ML integer. Real numbers are used so that results can be returned when this happens.

The function

```
support : bdd -> varSet
```

gives the variables that a BDD depends on.

An application is to define a function that counts the number of valuations of a BDD using `satcount`.

```
statecount : bdd -> real
```

The definition of `statecount` is

```
fun statecount bdd =
 let val sat    = satcount bdd
     val total  = Real.fromInt(getVarnum())
     val sup    = scanset(support bdd)
     val numsup = Real.fromInt(Vector.length sup)
     val free   = total - numsup
 in
  if equal bdd TRUE
   then 0.0
   else sat / Math.pow(2.0, free)
 end
```

If a BDD is representing a set of states, then `statecount` gives the number of states in the set (hence the name).

9 Coudert, Berthet & Madre simplification

The ML function

```
simplify : bdd -> bdd -> bdd
```

simplifies its second argument under the assumption that the first argument is true. Thus evaluating `simplify` b_1 b_2 results in a BDD b_2', hopefully simpler than b_2, such that $b_1 \Rightarrow (b_2 = b_2')$ or, equivalently, $b_1 \wedge b_2 = b_1 \wedge b_2'$. More precisely, the relationship between b_1, b_2 and b_2' is that the BDD `IMP(`b_1`,BIIMP(`b_2`,`b_2'`))` is the BDD `TRUE` (or, equivalently, that `AND(`b_1`,`b_2`)` and `AND(`b_1`,`b_2'`)` are `equal`, i.e. the same BDD).

For more details see Henrik Reif Andersen's lecture notes on BDDs [4], where the algorithm underlying `simplify` is described and attributed to a paper by Coudert, Berthet and Madre [5].

10 Saving, hashing and printing BDDs

BDDs can be saved on disk with the functions

```
bddSave : string -> bdd -> unit
bddLoad : string -> bdd
```

The string argument is a file name.

BuDDy provides two ways of printing BDDs: (i) as the set of paths from the root node to the *true* node and (ii) to the format used by the `dot` graph drawing program[2].

The function

```
hash : bdd -> int
```

hashes a bdd to an integer.

The functions for printing BDDs are;

```
printset   : bdd -> unit
printdot   : bdd -> unit
fnprintset : string -> bdd -> unit
fnprintdot : string -> bdd -> unit
```

`printset` and `printdot` print to standard output, whilst `fnprintset` and `fnprintdot` print to a file with the supplied name.

`printset` and `fnprintset` print out a sequence of paths, each one having the form

$< m_0$`:`n_0`, ... ,`m_l`:`n_l`>`

where the n_0, ... , n_l after the colon (`:`) are `0` or `1` and indicate that the next node in the path is reached by following the low (`false`) or high (`true`) pointer, respectively.

[2]`http://www.research.att.com/sw/tools/graphviz/`

For example, evaluating

```
printset (AND(ithvar 0, OR(ithvar 1, NOT(ithvar 2))))
```

results in

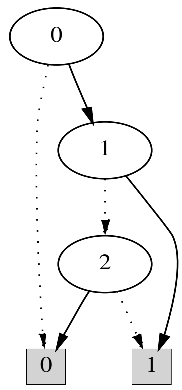
```
<0:1, 1:0, 2:0><0:1, 1:1>
```

which is best understood by looking at the diagram of the BDD drawn by `dot` that appears below.

To illustrate printing to `dot` format, the same BDD can be printed to a file `ex` by evaluating

```
fnprintdot "ex" (AND(ithvar 0, OR(ithvar 1, NOT(ithvar 2))))
```

executing `dot -Tps ex > ex.ps` (in Unix) results in the following Postscript diagram of a BDD



11 Dynamic variable reordering

BuDDy provides functions for dynamic variable reordering using a variety of methods. See the BuDDy documentation [11] for further details. The dynamic reordering types and functions provided in ML via MuDDy are in the structure `bdd` and are

```
eqtype fixed
FIXED           : fixed
FREE            : fixed

addvarblock     : varnum -> varnum -> fixed -> unit
```

```
clrvarblocks     : unit -> unit

eqtype method
WIN2             : method
WIN2ITE          : method
SIFT             : method
SIFTITE          : method
RANDOM           : method
REORDER_NONE     : method

reorder          : method -> unit
autoReorder      : method -> method
autoReorderTimes : method -> int -> method

getMethod        : unit -> method
getTimes         : unit -> int

disableReorder   : unit -> unit
enableReorder    : unit -> unit

varToLevel       : varnum -> int
varAtLevel       : int -> varnum
```

12 The MuDDy structure `fdd`

The structure `fdd` provides functions for manipulating values of finite domains. Functions are provided to allocate blocks of BDD variables to represent integer values instead of only Booleans.

Encoding is done with the least significant bits first in the BDD ordering. For example, if variables v_0, v_1, v_2, v_3 are used to encode 12, then the encoding would yield $v_0 = 0$, $v_1 = 0$, $v_2 = 1$ and $v_3 = 1$.

See the BuDDy documentation [11] for further details. See the ML structure `fdd` for the BuDDy facilities provides in ML via MuDDy:

```
type fddvar

extDomain  : int list -> fddvar list
clearAll   : unit -> unit
domainNum  : unit -> int
domainSize : fddvar -> int
varNum     : fddvar -> int
```

```
vars       : fddvar -> bdd.varnum list
ithSet     : fddvar -> bdd.varSet
domain     : fddvar -> bdd.bdd
setPairs   : (fddvar * fddvar) list -> bdd.pairSet
```

13 The MuDDy structure bvec

The structure bvec provides tools for encoding integers as arrays of BDDs, where each BDD represents one bit of an expression.

See the BuDDy documentation [11] for further details. See the ML structure bvec for the BuDDy facilities provides in ML via MuDDy.

```
type bvec

bvectrue    : fdd.precision -> bvec
bvecfalse   : fdd.precision -> bvec
con         : fdd.precision -> int -> bvec
var         : fdd.precision -> bdd.varnum -> int -> bvec
varfdd      : fdd.fddvar -> bvec

coerce      : fdd.precision -> bvec -> bvec

isConst     : bvec -> bool
getConst    : bvec -> int
lookupConst : bvec -> int option

add         : bvec * bvec -> bvec
sub         : bvec * bvec -> bvec
mul         : bvec * bvec -> bvec
mulfixed    : bvec * int -> bvec
div         : bvec * bvec -> bvec * bvec
divfixed    : bvec * int -> bvec * bvec
divi        : bvec * bvec -> bvec
divifixed   : bvec * int -> bvec

modu        : bvec * bvec -> bvec
modufixed   : bvec * int -> bvec
shl         : bvec -> bvec -> bdd.bdd -> bvec
shlfixed    : bvec -> int -> bdd.bdd -> bvec
shr         : bvec -> bvec -> bdd.bdd -> bvec
shrfixed    : bvec -> int -> bdd.bdd -> bvec
```

```
lth         : bvec * bvec -> bdd.bdd
lte         : bvec * bvec -> bdd.bdd
gth         : bvec * bvec -> bdd.bdd
gte         : bvec * bvec -> bdd.bdd
equ         : bvec * bvec -> bdd.bdd
neq         : bvec * bvec -> bdd.bdd
```

14 Storage allocation and garbage collection

The heart of the MuDDy package is mostly stub code that mirrors the BuDDy API and takes care of translating C values into SML values and vice versa.

The most tricky part is to make the Moscow ML garbage collector cooperate with the BuDDy garbage collector (we don't want either collector to try to collect the other's garbage). The cooperation is done by using the *finalized values* facility of the Moscow ML runtime system. That is, whenever a `bdd` value is returned from the BuDDy library, MuDDy register it as an external root (via `bdd_addref`) and wraps it into a finalized value.

A finalized value, in the Moscow ML runtime system, is a pair where the first component is the *destructor* (a function pointer) and the second component is the *data* (typicaly a pointer). When the Moscow ML collector collect a finalized value it apply the destructor on the data. In the case of the MuDDy package the destructor is `bdd_delref` and the data is the node-index returned by BuDDy.

Output showing the activation of the BuDDy garbage collector can be generated using the function

```
verbosegc : (string * string) option -> unit
```

Evaluating `verbosegc(SOME(`*pregc*`,`*postgc*`))` instructs BuDDy to print *pregc* when a BuDDy GC is initiated and print *postgc* when the BuDDy GC is completed.

Part II

Description of `HolBddLib`

`HolBddLib` currently consists of five modules

1. `Varmap` defines the ML type `varmap` that represents mappings, often denoted by ρ, from HOL variables to BDD variables;

2. `PrintBdd` provides rudimentary facilities for printing BDDs with respect to a varmap;

3. `PrimitiveBddRules` defines the protected type `term_bdd` representing BDD representation judgements $a\ \rho\ t\ \mapsto\ b$ with the semantics that under assumptions a, term t is represented by BDD b with respect to varmap ρ;

4. `DerivedBddRules` defines some derived rules for computing the representation of the reachable states of a transition system, and also for finding shortest paths to states satisfying a given property;

5. `MachineTransitionTheory` contains HOL reachability and fixedpoint theorems needed for the derived rules in `DerivedBddRules`.

Executing

```
load "HolBddLib";
```

loades these five modules and initialises BuDDy with a nodesize of 1000000 and cachesize of 10000.

If you want to perform your own BuDDy initialisation with different values, then instead of loading `HolBddLib`, load `bdd` and then call `bdd.init` with the parameters you want (see Section 1). `PrimitiveBddRulesTheory` and/or `MachineTransitionTheory` etc. can then be loaded.

15 The structure `Varmap`

The type `varmap` is defined by

```
type varmap = (string, int) Binarymap.dict
```

Strings are the names of HOL boolean variables and the integers associated with them are the corresponding BDD variables.

The following operations and predicates on varmaps are provided:

```
empty   : varmap
insert  : string * int -> varmap -> varmap
remove  : string -> varmap -> varmap
peek    : varmap -> string -> int option
dest    : varmap -> (string * int) list
eq      : varmap * varmap -> bool
size    : varmap -> int
extends : varmap -> varmap -> bool
unify   : varmap -> varmap -> varmap
```

with the semantics

`Varmap.empty`	the empty varmap
`Varmap.insert`	add an entry
`Varmap.remove`	delete an entry for a variable
`Varmap.peek`	lookup the value of a variable
`Varmap.dest`	convert to a list of pairs
`Varmap.eq`	pointer equality of varmaps (*not* general equality)
`Varmap.size`	number of entries
`Varmap.extends`	test if first argument included in second argument
`Varmap.unify`	compute smallest varmap that extends both arguments

16 The structure `PrintBdd`

`PrintBdd` builds on top of MuDDy's support for drawing BDDs using the `dot` program (see Section 10). Three functions are provided.

```
dotBdd              : string -> string -> bdd -> bdd
dotLabelledTermBdd : string -> string -> term_bdd -> unit
dotTermBdd          : term_bdd -> unit
```

`dotBdd` *file label bdd*
: prints the BDD *bdd* to *file*`.dot` with the label being the string *label*. The BDD variables are printed as the numbers used by BuDDy. The `dot` program is then invoked to create a postscript file *file*`.ps`. The argument BDD is returned.

`dotLabelledTermBdd` *file label tb*
: prints the BDD part of `term_bdd` *tb* with the nodes labelled with the variables specified in the varmap part of *tb*. A file *file*`.ps` is created, and the BDD is labelled with the string *label*.

`dotTermBdd` *tb*
: prints the BDD part of `term_bdd` *tb* with the nodes labelled with the variables specified in the varmap part of *tb*. A file `ScratchBdd.ps` is created, and the BDD is labelled by default with a representation of the term part of *tb*. The default labels can be suppressed (i.e. set to be always the empty string) by assigning `false` to the global reference `dotTermBddFlag`.

17 The structure `PrimitiveBddRules`

The structure `PrimitiveBddRules` defines the type `term_bdd` by

```
type assums = term HOLset.set;
datatype term_bdd = TermBdd of assums * varmap * term * bdd;
```

The constructor `TermBdd` is not exported, so the only way to construct values of type `term_bdd` is using the following inference rules (which are described in more detail in the rest of this section).

```
BddExtendVarmap           : varmap->term_bdd->term_bdd
BddFreevarsContractVarmap : term->term_bdd->term_bdd
BddSupportContractVarmap  : term->term_bdd->term_bdd
BddVar                    : bool->varmap->term->term_bdd
BddCon                    : bool->varmap->term_bdd
BddNot                    : term_bdd->term_bdd
BddIte                    : term_bdd*term_bdd*term_bdd->term_bdd
BddOp                     : bddop*term_bdd*term_bdd->term_bdd
BddForall                 : term list->term_bdd->term_bdd
BddExists                 : term list->term_bdd->term_bdd
BddAppall                 : term list->bddop*term_bdd*term_bdd->term_bdd
BddAppex                  : term list->bddop*term_bdd*term_bdd->term_bdd
```

```
BddCompose                : term_bdd*term_bdd->term_bdd->term_bdd
BddListCompose            : (term_bdd*term_bdd)list->term_bdd->term_bdd
BddRestrict               : (term_bdd*term_bdd)list->term_bdd->term_bdd
BddReplace                : (term_bdd*term_bdd)list->term_bdd->term_bdd
BddEqMp                   : thm->term_bdd->term_bdd
BddSimplify               : term_bdd*term_bdd->term_bdd
BddFindModel              : term_bdd->term_bdd
```

Destructor functions `dest_term_bdd`, `getAssums`, `getVarmap`, `getTerm` and `getBdd` for values of type `term_bdd` are described in Section 17.3

There is also a single oracle function `BddThmOracle` that derives the HOL theorem $a \vdash t$ from the representation judgement $a\ \rho\ t\ \mapsto$ `TRUE` (details are in Section 17.2).

Many of the rules assume that the varmaps in their `term_bdd` arguments are all equal. To apply these rules to hypotheses with different varmaps it may be possible to use `BddExtendVarmap`, `BddFreevarsContractVarmap` or `BddSupportContractVarmap` to make the varmaps equal. It is expected that derived rules to enable judgements with different varmaps to be combined will be implemented, however, as the soundness conditions for these are potentially subtle, such rules have not been included in the 'trusted kernel'.

Currently we have no formal treatment of notions of soundness or completeness for the rules in `PrimitiveBddRules`, though this is being thought about. We think the rules are 'obviously sound', but such intuitions are known to be unreliable! Our intuition about completeness is weaker: it is probable that as more experience with derived rules is obtained, the need for additional primitive rules will appear. Support for 'local scopes' (combining judgements with different variable orders) is an area that may reveal incompleteness in the current rules.

17.1 Rules for generating representation judgements

The notation $a_1 \cup a_2$ denotes the union of a_1 and a_2 Assumptions of representation judgements are identified up to α-conversion (as are assumptions of HOL theorems). The implementation is $a_1 \cup a_2$ = `HOLset.union` $a_1\ a_2$. The empty set of assumptions is denoted by $\{\}$, a set of assumptions containing terms $t_1, \ldots, t_n$ is denoted by $\{t_1, \ldots, t_n\}$ and $\{\}\ \rho\ t\ \mapsto\ b$ is abbreviated to $\rho\ t\ \mapsto\ b$.

Extending and contracting the varmap

`BddExtendVarmap : varmap -> term_bdd -> term_bdd`

$$\frac{\texttt{Varmap.extends}\ \rho_1\ \rho_2 \qquad a\ \rho_1\ t\ \mapsto\ b}{a\ \rho_2\ t\ \mapsto\ b}$$

Raises `BddExtendVarmapError` if ρ_2 doesn not extend ρ_1

`BddFreevarsContractVarmap : term -> term_bdd -> term_bdd`

$$\frac{a\ \rho\ t\ \mapsto\ b \qquad v \text{ not free in } t}{a\ \big(\texttt{Varmap.remove}\ \texttt{"}v\texttt{"}\ \rho\big)\ t\ \mapsto\ b}$$

Raises `BddFreevarsContractVarmapError` if v not free in t

`BddSupportContractVarmap : term -> term_bdd -> term_bdd`

$$\frac{a\ \rho\ t\ \mapsto\ b \qquad \rho(v) \text{ doesn't occur in } b}{a\ \big(\texttt{Varmap.remove}\ \texttt{"}v\texttt{"}\ \rho\big)\ t\ \mapsto\ b}$$

Raises `BddSupportContractVarmapError` if $\rho(v)$ not in the support of b

Variables and constants

`BddVar : bool -> varmap -> term -> term_bdd`

$$\frac{\rho(v) = n}{\rho\ v\ \mapsto\ \texttt{ithvar}\ n}\ \texttt{BddVar true}$$

$$\frac{\rho(v) = n}{\rho\ \neg v\ \mapsto\ \texttt{nithvar}\ n}\ \texttt{BddVar false}$$

Raises `BddVarError` if v not in the domain of ρ

```
BddCon : bool -> varmap -> term_bdd
```

$$\frac{}{\rho\ \texttt{T}\ \mapsto\ \texttt{TRUE}}\ \texttt{BddCon true}$$

$$\frac{}{\rho\ \texttt{F}\ \mapsto\ \texttt{FALSE}}\ \texttt{BddCon false}$$

Always succeeds

Boolean operations

```
BddNot : term_bdd -> term_bdd
```

$$\frac{a\ \rho\ t\ \mapsto\ b}{a\ \rho\ \neg t\ \mapsto\ \texttt{NOT}\ b}$$

Always succeeds

```
BddIte : term_bdd * term_bdd * term_bdd -> term_bdd
```

$$\frac{a\ \rho\ t\ \mapsto\ b \qquad a_1\ \rho\ t_1\ \mapsto\ b_1 \qquad a_2\ \rho\ t_2\ \mapsto\ b_2}{a \cup a_1 \cup a_2\ \rho\ \texttt{(if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2\texttt{)}\ \mapsto\ \texttt{ITE}\ b\ b_1\ b_2}$$

Raises `BddIteError` if the varmaps of the hypotheses are not all pointer equal

```
BddOp : bddop * term_bdd * term_bdd -> term_bdd
```

$$\frac{a_1\ \rho\ t_1\ \mapsto\ b_1 \qquad a_2\ \rho\ t_2\ \mapsto\ b_2}{a_1 \cup a_2\ \rho\ \texttt{(termApply}\ t_1\ t_2\ \mathit{bddop}\texttt{)}\ \mapsto\ \texttt{apply}\ b_1\ b_2\ \mathit{bddop}}$$

`termApply` t_1 t_2 *bddop* applies the HOL operation corresponding to the BuDDy BDD operation *bddop* to terms t_1 and t_2 (see Section 17.3). The exception `BddOpError` is raised if the varmaps of the hypotheses are not pointer equal

Quantification

`BddForall : term list -> term_bdd -> term_bdd`

$$\frac{a\ \rho\ t\ \mapsto\ b \qquad \rho(v_1) = n_1,\ \ldots\ ,\ \rho(v_i) = n_i}{a\ \rho\ (\forall v_1\ \cdots\ v_i.\ t)\ \mapsto\ \texttt{forall (makeset[}n_1, \ldots, n_i\texttt{])}\ b}$$

Raises `BddForallError` if any of the terms in the term list argument are not boolean variables in the domain of ρ, or occur free in any assumption

`BddExists : term list -> term_bdd -> term_bdd`

$$\frac{a\ \rho\ t\ \mapsto\ b \qquad \rho(v_1) = n_1,\ \ldots\ ,\ \rho(v_i) = n_i}{a\ \rho\ (\exists v_1\ \cdots\ v_i.\ t)\ \mapsto\ \texttt{exist (makeset[}n_1, \ldots, n_i\texttt{])}\ b}$$

Raises `BddExistsError` if any of the terms in the term list argument are not boolean variables in the domain of ρ, or occur free in any assumption

`BddAppall : term list -> bddop * term_bdd * term_bdd -> term_bdd`

$$\frac{a_1\ \rho\ t_1\ \mapsto\ b_1 \qquad a_2\ \rho\ t_2\ \mapsto\ b_2 \qquad \rho(v_1) = n_1,\ \ldots\ ,\ \rho(v_i) = n_i}{\begin{array}{l} a_1 \cup a_2\ \rho\ (\forall v_1\ \cdots\ v_i.\ \texttt{termApply}\ t_1\ t_2\ \mathit{bddop}) \\ \mapsto \\ \texttt{appall}\ b_1\ b_2\ \mathit{bddop}\ \texttt{(makeset[}n_1, \ldots, n_i\texttt{])}\ b \end{array}}$$

Raises `BddAppallError` if the varmaps in the hypotheses are not pointer equal, or if any of the terms in the term list argument are not boolean variables in the domain of ρ, or occur free in any assumption

BddAppex : term list -> bddop * term_bdd * term_bdd -> term_bdd

$$\frac{a_1\ \rho\ t_1\ \mapsto\ b_1 \qquad a_2\ \rho\ t_2\ \mapsto\ b_2 \qquad \rho(v_1)=n_1,\ \ldots\ ,\ \rho(v_i)=n_i}{\begin{array}{l} a_1\cup a_2\ \rho\ (\exists v_1\ \cdots\ v_i.\ \texttt{termApply}\ t_1\ t_2\ bddop) \\ \mapsto \\ \texttt{appex}\ b_1\ b_2\ bddop\ \texttt{(makeset[}n_1,\ldots,n_i\texttt{])}\ b \end{array}}$$

Raises `BddAppexError` if the varmaps of the hypotheses are not pointer equal, or if any of the terms in the term list argument are not boolean variables in the domain of ρ, or occur free in any assumption

Composition, repacement and restriction

BddCompose : term_bdd * term_bdd -> term_bdd -> term_bdd

$$\frac{(a_1\ \rho\ v_1\ \mapsto\ b_1,\ \ a_2\ \rho\ t_1\ \mapsto\ b_1') \qquad a\ \rho\ t\ \mapsto\ b}{a_1\cup a_2\cup a\ \rho\ \left(\texttt{subst[}v_1\ \texttt{|->}\ t_1\texttt{]}\ t\right)\ \mapsto\ \texttt{compose(var}\ b_1\texttt{,}\ b_1'\texttt{)}\ b}$$

Raises `BddComposeError` if varmaps in the hypotheses are not pointer equal, or the term v_1 is not a variable

BddListCompose : (term_bdd * term_bdd) list -> term_bdd -> term_bdd

$$\frac{\begin{array}{c} \texttt{[}(a_1\ \rho\ v_1\ \mapsto\ b_1,\ \ a_1'\ \rho\ t_1\ \mapsto\ b_1'), \\ \vdots \\ (a_i\ \rho\ v_i\ \mapsto\ b_i,\ \ a_i'\ \rho\ t_i\ \mapsto\ b_i')\texttt{]} \qquad a\ \rho\ t\ \mapsto\ b \end{array}}{\begin{array}{l} a_1\cup a_1'\ \cup\ \cdots\ \cup\ a_i\cup a_i'\cup a \\ \rho \\ \texttt{subst[}v_1\ \texttt{|->}\ t_1\texttt{,}\ \ldots\ \texttt{,}\ v_i\ \texttt{|->}\ t_i\texttt{]}\ t \\ \mapsto \\ \texttt{veccompose(composeSet[(var}\ b_1\texttt{,}\ b_1'\texttt{),}\ \ldots\ \texttt{,}\ \texttt{(var}\ b_i\texttt{,}\ b_i'\texttt{)])}b \end{array}}$$

Raises `BddListComposeError` if the varmaps in the hypotheses are not all pointer equal, or if any of the terms $v_1,\ldots,v_i$ are repeated or are not variables

`BddRestrict :  (term_bdd * term_bdd) list -> term_bdd -> term_bdd`

$$\frac{\texttt{[}(a_1\ \rho\ v_1\ \mapsto\ b_1,\ \ a'_1\ \rho\ c_1\ \mapsto\ b'_1), \quad \vdots \quad (a_i\ \rho\ v_i\ \mapsto\ b_i,\ \ a'_i\ \rho\ c_i\ \mapsto\ b'_i)\texttt{]} \qquad a\ \rho\ t\ \mapsto\ b}{\begin{array}{l} a_1 \cup a'_1\ \cup\ \cdots\ \cup\ a_i \cup a'_i \cup a \\ \rho \\ \texttt{subst[}v_1\ \texttt{|->}\ c_1\texttt{,}\ \ldots\ \texttt{,}\ v_i\ \texttt{|->}\ c_i\texttt{]}\ t \\ \mapsto \\ \texttt{restrict}\ b\ \texttt{(assignment[(var}\ b_1\texttt{,}\ \hat{c_1}\texttt{),}\ \ldots\ \texttt{, (var}\ b_i\texttt{,}\ \hat{c_i}\texttt{)])} \end{array}}$$

Where each of $c_1, \ldots, c_i$ is either the constant `F` or the constant `F`, and $\hat{\texttt{T}}$ denotes the ML value `true` and $\hat{\texttt{F}}$ denotes `false`. The exception `BddRestrictError` is raised if the varmaps in the hypotheses are not all pointer equal, or if any of the terms $v_1, \ldots, v_i$ are repeated or are not variables, or if any of $c_1, \ldots, c_i$ are not equal to `T` or `F`

`BddReplace :  (term_bdd * term_bdd) list -> term_bdd -> term_bdd`

$$\frac{\texttt{[}(a_1\ \rho\ v_1\ \mapsto\ b_1,\ \ a'_1\ \rho\ v'_1\ \mapsto\ b'_1), \quad \vdots \quad (a_i\ \rho\ v_i\ \mapsto\ b_i,\ \ a'_i\ \rho\ v'_i\ \mapsto\ b'_i)\texttt{]} \qquad a\ \rho\ t\ \mapsto\ b}{\begin{array}{l} a_1 \cup a'_1\ \cup\ \cdots\ \cup\ a_i \cup a'_i \cup a \\ \rho \\ \texttt{subst[}v_1\ \texttt{|->}\ v'_1\texttt{,}\ \ldots\ \texttt{,}\ v_i\ \texttt{|->}\ v'_i\texttt{]}\ t \\ \mapsto \\ \texttt{replace}\ b\ \texttt{(makepairSet[(var}\ b_1\texttt{, var}\ b'_1\texttt{),}\ \ldots\ \texttt{, (var}\ b_i\texttt{, var}\ b'_i\texttt{)])} \end{array}}$$

Raises `BddReplaceError` if the varmaps in the hypotheses are not all pointer equal, or if any of the terms $v_1, \ldots, v_i$ are repeated or are not variables, or if any of the terms $v'_1, \ldots, v'_i$ are repeated or are not variables

Coudert, Berthet & Madre simplification

```
BddSimplify : term_bdd * term_bdd -> term_bdd
```

$$\frac{a_1\ \rho\ t_1\ \mapsto\ b_1 \qquad a_2\ \rho\ t_2\ \mapsto\ b_2}{a_1 \cup a_2 \cup \{t_1\}\ \rho\ t_2\ \mapsto\ \texttt{simplify}\ b_1\ b_2}$$

The exception `BddSimplifyError` is raised if the varmaps in the hypotheses are not pointer equal

Finding a satisfying assignment

```
BddFindModel : term_bdd -> term_bdd
```

$$\frac{a\ \rho\ t\ \mapsto\ b}{a \cup \{v_1 = c_1, \ldots, v_p = c_p\}\ \rho\ t\ \mapsto\ \texttt{TRUE}}$$

The set $\{v_1 = c_1, \ldots, v_p = c_p\}$ is a satisfying assignment for t (c_i is T or F for $1 \leq i \leq p$). Exception `BddFindModelError` is raised if `satone` can't find a satisfying assignment.

17.2 Linking representation judgements to theorems

```
BddThmOracle : term_bdd -> thm
```

$$\frac{a\ \rho\ t\ \mapsto\ \texttt{TRUE}}{\texttt{[oracles: HolBdd]}\ a \vdash t}$$

Allows HOL theorems to be 'proved' by BDD calculation using BuDDy. Such theorems, and any theorems deduced from them, are tagged with `HolBdd` and so can be easily identified.

```
BddEqMp : thm -> term_bdd -> term_bdd
```

$$\frac{a_1 \vdash t_1 = t_2 \qquad a_2\ \rho\ t_1\ \mapsto\ b}{a_1 \cup a_2\ \rho\ t_2\ \mapsto\ b}$$

Enables the term part of a representation judgement to be replaced by a logically equivalent term. Raises `BddEqMpError` if the left hand side of the equation isn't α-convertable to the term part of the representation judgement

17.3 Miscellaneous functions

```
dest_term_bdd : term_bdd -> assums * varmap * term * bdd
getAssums     : term_bdd -> assums
getVarmap     : term_bdd -> varmap
getTerm       : term_bdd -> term
getBdd        : term_bdd -> bdd
```

`dest_term_bdd` $(a\ \rho\ t\ \mapsto\ b) = (\rho, t, b)$
`getVarmap` $(a\ \rho\ t\ \mapsto\ b) = \rho$
`getTerm` $(a\ \rho\ t\ \mapsto\ b) = t$
`getBdd` $(a\ \rho\ t\ \mapsto\ b) = b$

```
inSupport : int -> bdd -> bool
```

`inSupport` n b checks if the BDD variable n occurs in the BDD b

```
termApply : term -> term -> bddop -> term
```

`termApply` t_1 t_2 *bddop* applies the HOL operation corresponding to *bddop* to t_1 and t_2.

```
fun termApply t1 t2 bddop =
 case bddop of
    And    => mk_conj(t1,t2)
  | Biimp  => mk_eq(t1,t2)
  | Diff   => mk_conj(t1, mk_neg t2)
  | Imp    => mk_imp(t1,t2)
  | Invimp => mk_imp(t2,t1)
  | Lessth => mk_conj(mk_neg t1, t2)
  | Nand   => mk_neg(mk_conj(t1,t2))
  | Nor    => mk_neg(mk_disj(t1,t2))
  | Or     => mk_disj(t1,t2)
  | Xor    => mk_neg(mk_eq(t1,t2));
```

18 The structure `DerivedBddRules`

The documentation is this section is preliminary, reflecting the current status of the module `PrimitiveBddRules`. What follows is an edited copy of the source file `PrimitiveBddRules.sml` in which the comments are preserved, but most of the ML source code has been eliminated (some is left, if it is thought to be of pedagogical or documentation value).

```
(******************************************************************************)
(* Test equality of BDD component of two term_bdds and return true or false  *)
(******************************************************************************)

fun BddEqualTest tb1 tb2 = bdd.equal (getBdd tb1) (getBdd tb2);

(******************************************************************************)
(* Test if the BDD part is TRUE or FALSE                                      *)
(******************************************************************************)

fun isTRUE  tb = bdd.equal (getBdd tb) bdd.TRUE
and isFALSE tb = bdd.equal (getBdd tb) bdd.FALSE;

(******************************************************************************)
(* Count number of states (code from Ken Larsen)                              *)
(******************************************************************************)

statecount : bdd -> real

(******************************************************************************)
(* Destruct a term corresponding to a BuDDY BDD binary operation (bddop).    *)
(* Fail if not such a term.                                                   *)
(******************************************************************************)

exception dest_BddOpError;

dest_BddOp : term -> bddop * term * term

(******************************************************************************)
(* Function that always raises exception fail                                 *)
(* (useful as argument (leaffn) to GenTermToTermBdd)                          *)
(******************************************************************************)

exception fail;

fun failfn _ = raise fail;

(******************************************************************************)
(* Scan a term and construct a term_bdd using the primitive operations       *)
(* when applicable, and a supplied function on leaves when all else fails    *)
(******************************************************************************)

GenTermToTermBdd : (term -> term_bdd) -> varmap -> term -> term_bdd
```

```
(******************************************************************************)
(* Extend a varmap with a list of variables                                   *)
(* (allocating new BDD variables, if necessary)                               *)
(******************************************************************************)

extendVarmap : term list -> varmap -> varmap

(******************************************************************************)
(* Convert a BDD to a nested conditional term with respect to a varmap        *)
(******************************************************************************)

exception bddToTermError;

bddToTerm : varmap -> bdd -> term

(******************************************************************************)
(*              ass vm tm |--> b                                              *)
(*  ---------------------------------------------                             *)
(*  [oracles: HolBdd] ass |- tm = ^(bddToTerm vm b)                           *)
(******************************************************************************)

TermBddToEqThm : term_bdd -> thm

(******************************************************************************)
(* Global assignable varmap                                                   *)
(******************************************************************************)

val global_varmap = ref(Varmap.empty);

fun showVarmap () = Varmap.dest(!global_varmap);

(******************************************************************************)
(* Add variables to global_varmap and then call GenTermToTermBdd              *)
(* using the global function !termToTermBddFun on leaves                      *)
(******************************************************************************)

exception termToTermBddError;

val termToTermBddFun = ref(fn (tm:term) => (raise termToTermBddError));

fun termToTermBdd tm =
 let val vl = rev(all_vars tm)     (* all_vars returns vars in reverse order *)
     val vm = extendVarmap vl (!global_varmap)
     val _  = global_varmap := vm
 in GenTermToTermBdd (!termToTermBddFun) vm tm end;
```

```
(***************************************************************************)
(* MkIterThms ReachBy_rec``R((v1,...,vn),(v1',...,vn'))`` ``B(v1,...,vn)`` = *)
(*  ([|- ReachBy R B 0 (v1,...,vn) = B(v1,...,vn),                          *)
(*    |- !n. ReachBy R B (SUC n) (v1,...,vn) =                              *)
(*                ReachBy R B n (v1,...,vn)                                 *)
(*                \/                                                        *)
(*                ?v1'...vn'. ReachBy R B n (v1',...,vn')                   *)
(*                            /\                                            *)
(*                            R ((v1',...,vn'),(v1,...,vn))]                *)
(*                                                                          *)
(* MkIterThms ReachIn_rec``R((v1,...,vn),(v1',...,vn'))`` ``B(v1,...,vn)`` = *)
(*  ([|- ReachIn R B 0 (v1,...,vn) = B(v1,...,vn),                          *)
(*    |- !n. ReachIn R B (SUC n) (v1,...,vn) =                              *)
(*                ?v1'...vn'. ReachIn R B n (v1',...,vn')                   *)
(*                            /\                                            *)
(*                            R ((v1',...,vn'),(v1,...,vn))]                *)
(***************************************************************************)

MkIterThms : thm -> term -> term -> thm * thm

(***************************************************************************)
(* Perform disjunctive partitioning. Assume R is of the form               *)
(*                                                                          *)
(*  R((x,y,z),(x',y',z'))=                                                  *)
(*       ((x' = E1(x,y,z)) /\ (y' = y)         /\ (z' = z))                 *)
(*    \/ ((x' = x)         /\ (y' = E2(x,y,z)) /\ (z' = z))                 *)
(*    \/ ((x' = x)         /\ (y' = y)         /\ (z' = E3(x,y,z)))         *)
(*                                                                          *)
(* Then, for example, the equation:                                         *)
(*                                                                          *)
(*   ReachBy R B (SUC n) (x,y,z) =                                          *)
(*     ReachBy R B n (x,y,z)                                                *)
(*     \/                                                                   *)
(*     (?x_ y_ z_. ReachBy n R B (x_,y_,z_) /\ R((x_,y_,z_),(x,y,z))))      *)
(*                                                                          *)
(* is simplified to:                                                        *)
(*                                                                          *)
(*   ReachBy R B (SUC n) (x,y,z) =                                          *)
(*     ReachBy R B n (x,y,z)                                                *)
(*     \/ (?x_. ReachBy R B n (x_,y,z) /\ (x = E1(x_,y,z))                  *)
(*     \/ (?y_. ReachBy R B n (x,y_,z) /\ (y = E2(x,y_,z))                  *)
(*     \/ (?z_. ReachBy R B n (x,y,z_) /\ (z = E3(x,y,z_))                  *)
(***************************************************************************)
```

```
val MakeSimpRecThm = SIMP_RULE bool_ss [LEFT_AND_OVER_OR,EXISTS_OR_THM]);

(*****************************************************************************)
(*  MkPrevThm (|- R((v1,...,vn),(v1',...,vn')) = ...) =                     *)
(*    |- Prev R (Eq (v1',...,vn'))) (v1,...,vn) = ...                        *)
(*****************************************************************************)

MkPrevThm : thm -> thm

(*****************************************************************************)
(*  asl |- t1 = t2   ass vm t1' |--> b                                       *)
(*  ---------------------------------                                        *)
(*      (asl U ass) vm t2' |--> b'                                           *)
(*                                                                           *)
(* where t1 can be instantiated to t1' and t2' is the corresponding         *)
(* instance of t2                                                            *)
(*****************************************************************************)

fun BddApThm th tb =
 let val (_,vm,t1',b) = dest_term_bdd tb
 in  BddEqMp (REWR_CONV th t1') tb
      handle HOL_ERR _ => hol_err "REWR_CONV failed" "BddApthm"
 end;

(*****************************************************************************)
(*  ass vm t |--> b                                                          *)
(*  ----------------                                                         *)
(*  ass vm tm |--> b'                                                        *)
(*                                                                           *)
(* where boolean variables in t can be renamed to get tm and b' is          *)
(* the corresponding replacement of BDD variables in b                      *)
(*****************************************************************************)

exception BddApReplaceError;

BddApReplace : term_bdd -> term -> term_bdd

(*****************************************************************************)
(*   ass vm t  |--> b                                                        *)
(*   -----------------                                                       *)
(*   ass vm tm |--> b'                                                       *)
(*                                                                           *)
(* Generates the BDD of a supplied term if it can be obtained by restricting *)
(* a given term_bdd                                                          *)
(*****************************************************************************)
```

```
exception BddApRestrictError;

BddApRestrict : term_bdd -> term -> term_bdd

(*****************************************************************************)
(* BddSubst applies a substitution [(oldtb1,newtb1),...,(oldtni,newtbi)]     *)
(* to a term_bdd, where oldtbp (1 <= p <= i) must be of the form             *)
(* ass vm vp |--> bp where vp is a variable, and the varmaps are distinct    *)
(*                                                                           *)
(* The preliminary version below separates the substitution into a          *)
(* restriction (variables mapped to T or F) followed by a variable           *)
(* renaming (replacement).  A more elaborate scheme will be implemented      *)
(* using BuDDy's bdd_veccompose.                                             *)
(*****************************************************************************)

(*****************************************************************************)
(* Split a substitution [(oldtb1,newtb1),...,(oldtni,newtbi)]                *)
(* into a restriction and variable renaming, failing if this isn't possible  *)
(*****************************************************************************)

val split_subst =
 List.partition
  (fn (tb,tb')=> let val tm' = getTerm tb'
                 in (tm'=T) orelse (tm'=F) end);

(*****************************************************************************)
(*                    [(ass1 vm v1 |--> b1 , ass1' vm tm1 |--> b1'),         *)
(*                                  .                                        *)
(*                                  .                                        *)
(*                                  .                                        *)
(*                     (assi vm vi |--> bi , assi' vm tmi |--> bi')]         *)
(*                    ass vm tm |--> b                                       *)
(*  ------------------------------------------------------------------------ *)
(*   (as1 U ass1' U ... U assi U assi' U ass)                                *)
(*   vm                                                                      *)
(*   (subst[v1 |-> tm1, ... , vi |-> tmi]tm)                                 *)
(*   |-->                                                                    *)
(*   <BDD resulting from restrict followed by replace>                       *)
(*****************************************************************************)

fun BddSubst tbl tb =
 let val (res,rep) = split_subst tbl
 in BddReplace rep (BddRestrict res tb) end;
```

```
(*****************************************************************************)
(*  ass vm t |--> b                                                          *)
(*  -----------------                                                        *)
(*  ass vm tm |--> b'                                                        *)
(*                                                                           *)
(* where boolean variables in t can be instantiated to get tm and b' is      *)
(* the corresponding replacement of BDD variables in b                       *)
(*****************************************************************************)

exception BddApSubstError;

BddApSubst = fn : term_bdd -> term -> term_bdd

(*****************************************************************************)
(*          asl |- t1 = t2                                                   *)
(*   ------------------------------                                          *)
(*   (addList ass [])  vm t1 |--> b                                          *)
(*                                                                           *)
(* Fails if t2 is not built from variables using bddops                      *)
(*****************************************************************************)

fun eqToTermBdd leaffn vm th =
 let val th' = SPEC_ALL th
     val tm  = rhs(concl th')
 in BddEqMp (SYM th') (GenTermToTermBdd leaffn vm tm) end;

(*****************************************************************************)
(* Convert an ml positive integer to a HOL numeral                           *)
(*****************************************************************************)

fun intToTerm n = numSyntax.mk_numeral(Arbnum.fromInt n);

(*****************************************************************************)
(*  ass vm tm |--> b   conv tm  =  asl |- tm = tm'                           *)
(*  ----------------------------------------------                           *)
(*        (addList ass asl) vm tm' |--> b                                    *)
(*****************************************************************************)

fun BddApConv conv tb = BddEqMp (conv(getTerm tb)) tb;
```

```
(*****************************************************************************)
(*   |- t1 = t2                                                              *)
(*   ----------                                                              *)
(*     |- t1                                                                 *)
(*                                                                           *)
(* if the BDD of t2 (using GenTermToTermBdd) is TRUE                         *)
(*****************************************************************************)

BddRhsOracle :  (term -> term_bdd) -> varmap -> thm -> thm

(*****************************************************************************)
(* Iterate a function f : int -> 'a -> 'a                                    *)
(* from an initial value, applying it successively to 0,1,2,... until        *)
(*                                                                           *)
(*   p : 'a -> bool                                                          *)
(*                                                                           *)
(* is true (at least one iteration is always performed)                      *)
(*****************************************************************************)

fun iterate p f =
 let fun iter n x =
      let val x'  = f n x
      in if p x' then x' else iter (n+1) x' end
 in iter 0 end;

(*****************************************************************************)
(*   |- f 0 s = ... s ...     |- !n. f (SUC n) s = ... f n ... s ...         *)
(*   ----------------------------------------------------------------        *)
(*       (vm ``f i s`` |--> bi, vm ``f (SUC i) s`` |--> bsuci)               *)
(*                                                                           *)
(* where i is the first number such that |- f (SUC i) s = f i s              *)
(* and the function report is applied to the iteration level and current     *)
(* term_bdd and can be used for tracing.                                     *)
(*                                                                           *)
(* A state of the iteration is a pair (tb,tb') consisting of the             *)
(* previous term_bdd tb and the current one tb'. The initial state           *)
(* is (somewhat arbitarily) taken to be (tb0,tb0).                           *)
(*****************************************************************************)

exception computeFixedpointError;

computeFixedpoint : (int -> term_bdd -> 'a) -> varmap -> thm * thm -> term_bdd
```

```
(******************************************************************************)
(*              ass vm tm |--> b                                             *)
(*  ---------------------------------------------                            *)
(*  [((ass1 vm v1 |--> b1),(ass1' vm c1 |--> b1')),                          *)
(*                      .                                                    *)
(*                      .                                                    *)
(*                      .                                                    *)
(*   ((assi vm vi |--> bi),(assi' vm ci |--> bi')]                           *)
(*                                                                           *)
(* with the property that                                                    *)
(*                                                                           *)
(* BddRestrict [((ass1 vm v1 |--> b1),(ass1' vm c1 |--> b1')),               *)
(*                                .                                          *)
(*                                .                                          *)
(*                                .                        ,                 *)
(*              ((assi vm vi |--> bi),(assi' vm ci |--> bi'))]               *)
(*             (ass vm tm |--> b)                                            *)
(* =                                                                         *)
(* (ass1 U ass1' U ... U assi U assi' U ass)                                 *)
(* vm                                                                        *)
(* (subst[v1|->ci,...,vi|->ci]tm)                                            *)
(* |--> TRUE                                                                 *)
(******************************************************************************)

exception BddSatoneError;

BddSatone : term_bdd -> (term_bdd * term_bdd) list

(******************************************************************************)
(*        |- p s = ... s ...                                                 *)
(*        |- f 0 s  = ... s ...                                              *)
(*        |- f (SUC n) s = ... f n ... s ...                                 *)
(*  -------------------------------------------------------                  *)
(*  [{} vm ``f i s`` |--> bi,  ... , {} vm ``f 0 s`` |--> b0]                *)
(*                                                                           *)
(* where i is the first number such that |- f i s ==> p s                    *)
(******************************************************************************)

exception computeTraceError;

computeTrace : (int->term_bdd->'a) -> varmap -> thm -> thm*thm -> term_bdd list
```

```
(******************************************************************************)
(*  traceBack vm                                                              *)
(*   [{} vm ``f i s`` |--> bi,  ... , {} vm ``f 0 s`` |--> b0]                *)
(*   (|- p s = ... s ...)                                                     *)
(*   (|- R((v1,...,vn),(v1',...,vn')) = ...)                                  *)
(*                                                                            *)
(* computes a list of pairs of the form (with j = 0,1,...,i-1)                *)
(*                                                                            *)
(* ((vm ``ReachIn R B j s_vec /\ Prev R (Eq c_vec) (v1,...,vn)`` |--> bdd),  *)
(*  [((vm v1 |--> b1),(vm c1 |--> b1')),                                      *)
(*                  .                                                         *)
(*                  .                                                         *)
(*                  .                 ,                                       *)
(*   ((vm vn |--> bn),(vm cn |--> bn'))])                                     *)
(*                                                                            *)
(* where s_vec = (v1,...,vn) and c_vec = (c1,...,cn) where ci is T or F      *)
(* and the second element specifies a state satisfying the first element     *)
(* and in which state variable vj has value cj (0 <= j <= n).                 *)
(* The last element of the list has the form                                  *)
(* (({} vm ``ReachIn R B j s_vec /\ p(v1,...,vn)`` |--> bdd),                 *)
(*  [(({} vm v1 |--> b1),{} vm c1 |--> b1')),                                 *)
(*                      .                                                     *)
(*                      .                                                     *)
(*                      .                 ,                                   *)
(*   (({} vm vn |--> bn),({} vm cn |--> bn'))])                               *)
(*                                                                            *)
(* If [s0,...,si] is the sequence of states, then                             *)
(* R(s0,s1), R(s1,s2),...,R(s(i-1),sj) and sj satisfies bj and p si          *)
(******************************************************************************)

traceBack : varmap
             -> term_bdd list
             -> thm -> thm -> (term_bdd * (term_bdd * term_bdd) list) list

(******************************************************************************)
(*  findTrace                                                                 *)
(*   (|- R((v1,...,vn),(v1',...,vn')) = ...)                                  *)
(*   (|- P(v1,...,vn) = ...)                                                  *)
(*   (|- Q(v1,...,vn) = ...)                                                  *)
(*  =                                                                         *)
(*   ((|- P s_0), [(|- R(s_0,s_1)),...,(|- R(s_(n-1),s_n))], (|- Q s_n))     *)
(******************************************************************************)

findTrace : varmap -> thm -> thm -> thm -> thm * thm list * thm
```

```
(******************************************************************************)
(* If t is satifiable (i.e. b is not FALSE)                                   *)
(*                                                                            *)
(*            a vm t |--> b                                                   *)
(*      --------------------------                                            *)
(*      a U {v1=c1,...,vn=cn} |- t                                            *)
(*                                                                            *)
(* Similar to BddFindModel followed by BddThmOracle, but checks the          *)
(* assignment found by satone using proof, so is pure                         *)
(* (i.e. result not tagged with HolBdd)                                       *)
(*                                                                            *)
(******************************************************************************)

findModel : term_bdd -> thm
```

19 The structure MachineTransitionTheory

The theory MachineTransitionTheory contained the HOL theoremes used by the derived rules in DerivedBddRules. The signature file (slightly edited) is given below.

```
signature MachineTransitionTheory =
sig
  type thm = Thm.thm

  (*  Definitions  *)
    val ChoosePath_def : thm
    val Eq_def : thm
    val FinPath_arg_munge_def : thm
    val FinPath_tupled_primitive_def : thm
    val FnPair_def : thm
    val IsTrace_arg_munge_def : thm
    val IsTrace_tupled_primitive_def : thm
    val Live_def : thm
    val MooreTrans_def : thm
    val Moore_def : thm
    val Next_def : thm
    val Path_def : thm
    val Prev_def : thm
    val ReachBy_def : thm
    val ReachIn_def : thm
    val Reachable_def : thm
    val Stable_def : thm
```

```
  val Total_def : thm
  val Totalise_def : thm

(*  Theorems  *)
  val ABS_EXISTS_THM : thm
  val ABS_ONE_ONE : thm
  val COND_SIMP : thm
  val EQ_COND : thm
  val EXISTS_IMP_EQ : thm
  val EXISTS_REP : thm
  val FORALL_REP : thm
  val FinFunEq : thm
  val FinPathLemma : thm
  val FinPathPathExists : thm
  val FinPathThm : thm
  val FinPath_def : thm
  val FinPath_ind : thm
  val FnPairAbs : thm
  val FnPairExists : thm
  val FnPairForall : thm
  val IsTrace_def : thm
  val IsTrace_ind : thm
  val ModelCheckAlways : thm
  val ModelCheckAlwaysCor1 : thm
  val ModelCheckAlwaysCor2 : thm
  val MoorePath : thm
  val MooreReachable : thm
  val MooreReachable1 : thm
  val MooreReachable2 : thm
  val MooreReachableCor1 : thm
  val MooreReachableExists : thm
  val MooreTransEq : thm
  val ReachBy_ReachIn : thm
  val ReachBy_fixedpoint : thm
  val ReachBy_rec : thm
  val ReachInFinPath : thm
  val ReachInPath : thm
  val ReachIn_rec : thm
  val ReachIn_revrec : thm
  val ReachableFinPath : thm
  val ReachableMooreTrans : thm
  val ReachablePath : thm
  val ReachablePathThm : thm
  val ReachableTotalise : thm
  val Reachable_ReachBy : thm
```

```
    val Reachable_Stable : thm
    val TotalImpTotalise : thm
    val TotalImpTotaliseLemma : thm
    val TotalMooreTrans : thm
    val TotalTotalise : thm
    val TotaliseReachBy : thm
    val TotalpathExists : thm
    val TraceReachIn : thm

  val MachineTransition_grammars : type_grammar.grammar * term_grammar.grammar
(*
   [list] Parent theory of "MachineTransition"

   [option] Parent theory of "MachineTransition"

   [ChoosePath_def]
   Definition
   |- (!R s. ChoosePath R s 0 = s) /\
      !R s n. ChoosePath R s (SUC n) = @s'. R (ChoosePath R s n,s')

   [Eq_def]
   Definition
   |- !state0 state. Eq state0 state = (state0 = state)

   [FinPath_arg_munge_def]
   Definition
   |- !x x1 x2. FinPath x x1 x2 = FinPath_tupled (x,x1,x2)

   [FinPath_tupled_primitive_def]
   Definition
   |- FinPath_tupled =
      WFREC (@R'. WF R' /\ !n f s R. R' ((R,s),f,n) ((R,s),f,SUC n))
        (\FinPath_tupled a.
           case a of
              (v,v1) ->
                case v of
                   (v2,v3) ->
                     case v1 of
                        (v4,v5) ->
                          case v5 of 0 -> v4 0 = v3
                          || SUC v6 -> FinPath_tupled ((v2,v3),v4,v6) /\
                                       v2 (v4 v6,v4 (v6 + 1)))

   [FnPair_def]  Definition  |- !f g x. FnPair f g x = (f x,g x)
```

```
[IsTrace_arg_munge_def]
Definition
|- !x x1 x2 x3. IsTrace x x1 x2 x3 = IsTrace_tupled (x,x1,x2,x3)

[IsTrace_tupled_primitive_def]
Definition
|- IsTrace_tupled =
   WFREC
     (@R'. WF R' /\ !s0 B tr Q s1 R. R' (R,Eq s1,Q,s1::tr) (R,B,Q,s0::s1::tr))
     (\IsTrace_tupled a.
        case a of
           (v,v1) ->
             case v1 of
                (v2,v3) ->
                  case v3 of
                     (v4,v5) ->
                       case v5 of
                          [] -> F
                       || v6::v7 ->
                            case v7 of
                               [] -> v2 v6 /\ v4 v6
                            || v8::v9 -> v2 v6 /\ v (v6,v8) /\
                               IsTrace_tupled (v,Eq v8,v4,v8::v9))

[Live_def]
Definition
|- !R. Live R = !state. ?state'. R (state,state')

[MooreTrans_def]
Definition
|- !nextfn input state input' state'.
     MooreTrans nextfn ((input,state),input',state') =
     (state' = nextfn (input,state))

[Moore_def]
Definition
|- !nextfn inputs states.
     Moore nextfn (inputs,states) =
     !t. states (t + 1) = nextfn (inputs t,states t)

[Next_def]
Definition
|- !R B state. Next R B state = ?state_. B state_ /\ R (state_,state)

[Path_def]
```

```
Definition
|- !R s f. Path (R,s) f = (f 0 = s) /\ !n. R (f n,f (n + 1))

[Prev_def]
Definition
|- !R Q state. Prev R Q state = ?state'. R (state,state') /\ Q state'

[ReachBy_def]
Definition
|- !R B n state. ReachBy R B n state = ?m. m <= n /\ ReachIn R B m state

[ReachIn_def]
Definition
|- (!R B. ReachIn R B 0 = B) /\
   !R B n. ReachIn R B (SUC n) = Next R (ReachIn R B n)

[Reachable_def]
Definition
|- !R B state. Reachable R B state = ?n. ReachIn R B n state

[Stable_def]
Definition
|- !R state. Stable R state = !state'. R (state,state') ==> (state' = state)

[Total_def]  Definition  |- !R. Total R = !s. ?s'. R (s,s')

[Totalise_def]
Definition
|- !R s s'.
     Totalise R (s,s') = R (s,s') \/ ~(?s''. R (s,s'')) /\ (s = s')

[ABS_EXISTS_THM]
Theorem
|- !P rep.
     TYPE_DEFINITION P rep ==>
     ?abs. (!a. abs (rep a) = a) /\ !r. P r = (rep (abs r) = r)

[ABS_ONE_ONE]
Theorem
|- !abs rep.
     (!a. abs (rep a) = a) /\ (!r. range r = (rep (abs r) = r)) ==>
     !r. range r /\ range r' ==> ((abs r = abs r') = (r = r'))

[COND_SIMP]
Theorem
```

```
|- ((if b then F else F) = F) /\ ((if b then F else T) = ~b) /\
   ((if b then T else F) = b) /\ ((if b then T else T) = T) /\
   ((if b then x else x) = x) /\ ((if b then b' else ~b') = (b = b')) /\
   ((if b then ~b' else b') = (b = ~b'))

[EQ_COND]
Theorem
|- ((x = (if b then y else z)) = (if b then x = y else x = z)) /\
   (((if b then y else z) = x) = (if b then y = x else z = x))

[EXISTS_IMP_EQ]  Theorem  |- (?x. P x) ==> Q = !x. P x ==> Q

[EXISTS_REP]
Theorem
|- !abs rep P Q.
     (!a. abs (rep a) = a) /\ (!r. P r = (rep (abs r) = r)) ==>
     ((?a. Q a) = ?r. P r /\ Q (abs r))

[FORALL_REP]
Theorem
|- !abs rep P Q.
     (!a. abs (rep a) = a) /\ (!r. P r = (rep (abs r) = r)) ==>
     ((!a. Q a) = !r. P r ==> Q (abs r))

[FinFunEq]
Theorem
|- (!m. m <= n + 1 ==> (f1 m = f2 m)) =
   (!m. m <= n ==> (f1 m = f2 m)) /\ (f1 (n + 1) = f2 (n + 1))

[FinPathLemma]
Theorem
|- !f1 f2 n.
     (!m. m <= n ==> (f1 m = f2 m)) ==>
     (FinPath (R,s) f1 n = FinPath (R,s) f2 n)

[FinPathPathExists]
Theorem
|- !R B f s n.
     Total R /\ FinPath (R,s) f n ==>
     ?g. (!m. m <= n ==> (f m = g m)) /\ Path (R,s) g

[FinPathThm]
Theorem
|- !n. FinPath (R,s) f n = (f 0 = s) /\ !m. m < n ==> R (f m,f (m + 1))
```

```
[FinPath_def]
Theorem
|- (FinPath (R,s) f 0 = (f 0 = s)) /\
   (FinPath (R,s) f (SUC n) = FinPath (R,s) f n /\ R (f n,f (n + 1)))

[FinPath_ind]
Theorem
|- !P.
     (!R s f. P (R,s) f 0) /\
     (!R s f n. P (R,s) f n ==> P (R,s) f (SUC n)) ==>
     !v v1 v2 v3. P (v,v1) v2 v3

[FnPairAbs]
Theorem
|- (!tr. FnPair (\n. FST (tr n)) (\n. SND (tr n)) = tr) /\
   !tr1 tr2. (\n. (tr1 n,tr2 n)) = FnPair tr1 tr2

[FnPairExists]
Theorem
|- !P. (?tr. P tr) = ?tr1 tr2. P (FnPair tr1 tr2)



[FnPairForall]
Theorem
|- !P. (!tr. P tr) = !tr1 tr2. P (FnPair tr1 tr2)

[IsTrace_def]
Theorem
|- (IsTrace R B Q [] = F) /\ (IsTrace R B Q [s] = B s /\ Q s) /\
   (IsTrace R B Q (s0::s1::tr) =
    B s0 /\ R (s0,s1) /\ IsTrace R (Eq s1) Q (s1::tr))

[IsTrace_ind]
Theorem
|- !P.
     (!R B Q. P R B Q []) /\ (!R B Q s. P R B Q [s]) /\
     (!R B Q s0 s1 tr.
        P R (Eq s1) Q (s1::tr) ==> P R B Q (s0::s1::tr)) ==>
     !v v1 v2 v3. P v v1 v2 v3

[ModelCheckAlways]
Theorem
|- !R B P.
     (!s. Reachable R B s ==> P s) ==>
```

```
     !tr. B (tr 0) /\ (!t. R (tr t,tr (t + 1))) ==> !t. P (tr t)

[ModelCheckAlwaysCor1]
Theorem
|- (!s1 s2. Reachable R B (s1,s2) ==> P s1) ==>
   !tr. B (tr 0) /\ (!t. R (tr t,tr (t + 1))) ==> !t. P (FST (tr t))

[ModelCheckAlwaysCor2]
Theorem
|- !R B P.
     (!s1 s2. Reachable R B (s1,s2) ==> P s1) ==>
     !tr1.
       (?tr2.
          B (tr1 0,tr2 0) /\
          !t. R ((tr1 t,tr2 t),tr1 (t + 1),tr2 (t + 1))) ==>
       !t. P (tr1 t)

[MoorePath]
Theorem
|- Moore nextfn (inputs,states) =
   Path (MooreTrans nextfn,inputs 0,states 0) (\t. (inputs t,states t))

[MooreReachable]
Theorem
|- !B nextfn P.
     (!inputs states.
        B (inputs 0,states 0) /\ Moore nextfn (inputs,states) ==>
        !t. P (inputs t,states t)) =
     !s. Reachable (MooreTrans nextfn) B s ==> P s

[MooreReachable1]
Theorem
|- (!inputs states.
      B (inputs 0,states 0) /\ Moore nextfn (inputs,states) ==>
      !t. P (inputs t,states t)) ==>
   !s. Reachable (MooreTrans nextfn) B s ==> P s




[MooreReachable2]
Theorem
|- (!s. Reachable (MooreTrans nextfn) B s ==> P s) ==>
   !inputs states.
```

```
     B (inputs 0,states 0) /\ Moore nextfn (inputs,states) ==>
     !t. P (inputs t,states t)

[MooreReachableCor1]
Theorem
|- !B nextfn.
     (!inputs states.
        B (inputs 0,states 0) /\
        (!t. states (t + 1) = nextfn (inputs t,states t)) ==>
        !t. P (inputs t,states t)) =
     !s. Reachable (\((i,s),i',s'). s' = nextfn (i,s)) B s ==> P s

[MooreReachableExists]
Theorem
|- (?inputs states.
      (B (inputs 0,states 0) /\ Moore nextfn (inputs,states)) /\
      ?t. P (inputs t,states t)) =
   ?s. Reachable (MooreTrans nextfn) B s /\ P s

[MooreTransEq]
Theorem
|- MooreTrans nextfn =
   (\((input,state),input',state'). state' = nextfn (input,state))

[ReachBy_ReachIn]
Theorem
|- (!R B state. ReachBy R B 0 state = B state) /\
   !R B n state.
     ReachBy R B (SUC n) state =
     ReachBy R B n state \/ ReachIn R B (SUC n) state

[ReachBy_fixedpoint]
Theorem
|- !R B n.
     (ReachBy R B n = ReachBy R B (SUC n)) ==>
     (Reachable R B = ReachBy R B n)

[ReachBy_rec]
Theorem
|- (!R B state. ReachBy R B 0 state = B state) /\
   !R B n state.
     ReachBy R B (SUC n) state =
     ReachBy R B n state \/
     ?state_. ReachBy R B n state_ /\ R (state_,state)
```

```
[ReachInFinPath]
Theorem
|- !R B n s. ReachIn R B n s = ?f s0. B s0 /\ FinPath (R,s0) f n /\ (s = f n)

[ReachInPath]
Theorem
|- !R B n s.
     Total R ==> (ReachIn R B n s = ?f s0. B s0 /\ Path (R,s0) f /\ (s = f n))

[ReachIn_rec]
Theorem
|- (!R B state. ReachIn R B 0 state = B state) /\
   !R B n state.
     ReachIn R B (SUC n) state =
     ?state_. ReachIn R B n state_ /\ R (state_,state)

[ReachIn_revrec]
Theorem
|- (!R B state. ReachIn R B 0 state = B state) /\
   !R B n state.
     ReachIn R B (SUC n) state =
     ?state1 state2.
       B state1 /\ R (state1,state2) /\ ReachIn R (Eq state2) n state

[ReachableFinPath]
Theorem
|- !R B s.
     Reachable R B s = ?f s0 n. B s0 /\ FinPath (R,s0) f n /\ (s = f n)

[ReachableMooreTrans]
Theorem
|- !B s.
     Reachable (MooreTrans nextfn) B s =
     ?f s0. B s0 /\ Path (MooreTrans nextfn,s0) f /\ ?n. s = f n

[ReachablePath]
Theorem
|- !R B s.
     Total R ==>
     (Reachable R B s = ?f s0. B s0 /\ Path (R,s0) f /\ ?n. s = f n)

[ReachablePathThm]
Theorem
|- !R B s.
     Reachable R B s =
```

```
      ?f s0. B s0 /\ Path (Totalise R,s0) f /\ ?n. s = f n

   [ReachableTotalise]  Theorem  |- Reachable (Totalise R) = Reachable R

   [Reachable_ReachBy]
   Theorem
   |- Reachable R B state = ?n. ReachBy R B n state

   [Reachable_Stable]
   Theorem
   |- Live R /\ (!state. ReachIn R B n state ==> Stable R state) ==>
      !state. Reachable R B state /\ Stable R state = ReachIn R B n state

   [TotalImpTotalise]  Theorem  |- Total R ==> (Totalise R = R)

   [TotalImpTotaliseLemma]
   Theorem
   |- Total R ==> !s s'. R (s,s') = Totalise R (s,s')

   [TotalMooreTrans]  Theorem  |- Total (MooreTrans nextfn)

   [TotalTotalise]  Theorem  |- Total (Totalise R)

   [TotaliseReachBy]
   Theorem  |- !n s. ReachBy (Totalise R) B n s = ReachBy R B n s

   [TotalpathExists]
   Theorem  |- Total R ==> !s. Path (R,s) (ChoosePath R s)

   [TraceReachIn]
   Theorem
   |- !R B tr. B (tr 0) /\ (!n. R (tr n,tr (n + 1))) ==> !n. ReachIn R B n (tr n)

*)
end
```

Acknowledgements

HolBddLib would not have been possible without BuDDy from Jørn Lind-Nielsen and MuDDy from Ken Friis Larsen and Jakob Lichtenberg.

This research was initially supported by EPSRC grant GR/K57343 *Checking Equivalence Between Synthesised Logic and Non-Synthesisable Behavioural*

Prototypes, EPSRC grant GR/L35973 entitled *A Hardware Compilation Workbench*, EPSRC grant GR/L74262 entitled *A uniform semantics for Verilog and VHDL suitable for both simulation and formal verification* and ESPRIT Framework IV LTR 26241 project Prosper (*Proof and Specification Assisted Design Environments*). Currently the research is supported by EPSRC grant GR/R27105/01 entitled *Fully Expansive Proof and Algorithmic Verification*[3].

At the beginning of the research, data from Atanas Parashkevov and Bill Roscoe on the BDD and state space sizes arising from Peg Solitaire was useful for evaluating and testing the first version of `HolBddLib`.

Michael Norrish and Konrad Slind have provided invaluable help with Hol98, which they are currently developing.

Mark Aagaard provided some of the information on Voss and its successors described in the preface.

Paul Jackson, Jesper Møller and Konrad Slind provided detailed comments and suggestions on a first draft of the University of Cambridge Computer Laboratory Technical Report No. 481.

References

[1] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Design Automation Conference (DAC)*, pages 538–541. ACM/IEEE, July 1998.

[2] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In *Theorem Proving in Higher Order Logics (TPHOLs99)*, number 1690 in Lecture Notes in Computer Science, pages 323–340. Springer-Verlag, 1999.

[3] Hasan Amjad. BDD representation judgements in HOL: A performance evaluation. Category B paper presented at the *14th International Conference on Theorem Proving and Higher Order Logics*. Edinburgh, Scotland, UK, August 2001. Available from `http://www.cl.cam.ac.uk/~ha227/`.

[3] `http://www.cl.cam.ac.uk/~mjcg/HolCheck/`

[4] Henrik Reif Andersen. An Introduction to Binary Decision Diagrams, October 1997. Lecture notes for 49285 Advanced Algorithms E97. Available from: `http://www.it.dtu.dk/~hra`.

[5] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 365–373. Springer-Verlag, 1989.

[6] Michael J. C. Gordon. Reachability programming in HOL using BDDs. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 180–197. Springer-Verlag, 2000.

[7] Mike Gordon and Ken Friis Larsen. Combining the Hol98 proof assistant with the BuDDy BDD package. Technical Report 481, University of Cambridge Computer Laboratory, December 1999.

[8] Scott Hazelhurst and Carl-Johan H. Seger. Symbolic trajectory evaluation. In Thomas Kropf, editor, *Formal Hardware Verification*, chapter 1, pages 3–78. Springer-Verlag, 1997.

[9] J. Joyce and C. Seger. The HOL-Voss System: Model-Checking inside a General-Purpose Theorem-Prover. In J. J. Joyce and C.-J. H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG'93, Vancouver, B.C., August 11-13 1993*, volume 780 of *Lecture Notes in Computer Science*, pages 185–198. Spinger-Verlag, 1994.

[10] Trevor W. S. Lee, Mark R. Greenstreet, and Carl-Johan Seger. Automatic verification of asynchronous circuits. Technical Report UBC TR 93-40, The University of British Columbia, November 1993.

[11] Jørn Lind-Nielsen's BuDDy package is documented at `http://www.itu.dk/research/buddy/`.

[12] John O'Leary, Xudong Zhao, Robert Gerth, and Carl-Johan H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel*

Technology Journal, First Quarter 1999. Online at `http://developer.intel.com/technology/itj/`.

[13] Carl-Johan H. Seger. Voss - a formal hardware verification system: User's guide. Technical Report UBC TR 93-45, The University of British Columbia, December 1993.