Technical Report

UCAM-CL-TR-480
ISSN 1476-2986

Number 480

Programming combinations of deduction and BDD-based symbolic calculation

Mike Gordon

December 1999

ISSN 1476-2986

Programming combinations of deduction and BDD-based symbolic calculation

Mike Gordon*

December 15, 1999

Abstract

Theorem provers descended from LCF allow their users to write complex proof tools that provide high assurance that false theorems will not be proved. This paper describes some experiments in extending the 'LCF approach' to enable BDD-based symbolic algorithms to be programmed with a similar assurance. The deduction is supplied by the HOL system and the BDD algorithms by the BuDDy package.

1 Introduction

LCF-style theorem provers [12] extend the ML programming language [19] with two types: *term* representing logical terms[1] and *thm* representing theorems. Milner's key idea was to make *thm* an abstract type whose only theorem-creating operations correspond to rules of inference of a logic. Users can program complex proof procedures in ML by calling the primitive operations of *thm*. The ML type discipline ensures that theorem-values can only be created via sequences of primitive inferences. The set of theorems corresponds to a subset of the set of terms, namely those terms that are proveable. The notation $\vdash t$ means that term t is a theorem. There is a long-standing controversy about whether the LCF-approach can achieve good enough efficiency. In a surprising number of cases it can [6]. However, programming

*Univerity of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, United Kingdom, Email: `mjcg@cl.cam.ac.uk`

[1]Boolean terms represent predicate calculus formulae.

decision procedures and theorem provers in an LCF-style is more demanding than just implementing them as algorithms. Thus, even if a pure LCF-style solution is possible, it might be more cost-effective to use a simpler approach. Modern LCF-style provers (e.g. HOL [11] and Isabelle [22]) allow 'oracles' to create theorems. Such oracles can either be ML programs that operate directly on the data-structures underlying theorems, or they might be external tools implemented in other languages (e.g. C).

Many successful automatic verification algorithms are based on computing with Boolean terms represented as binary decision diagrams (BDDs for short) [7]. There is a lot of work on combining general higher order logic theorem proving with BDD-based verification oracles (e.g. [23, 5]). The approach usually taken is to regard the oracle as a 'black box' invoked as a separate process using translations between the formalisms of the two systems.

The work described here differs in that it aims to provide 'LCF-style' general infrastructure enabling users to implement their own BDD-based verification algorithms inside HOL. Thus, rather than link to an external black box model checker one can program up a bespoke checker with an LCF-like assurance of soundness.

Preliminary experiments described here suggest that not only can standard state exploration algorithms be efficiently and securely programmed in HOL, but that their tight integration makes it easy to combine algorithmic and deductive methods to do things that are not possible with either component in isolation. An example is using theorem proving to rewrite terms so that the resulting BDDs are more tractable, then performing BDD calculations, then converting the result back to a term and simplifying it.

The rest of this paper is structured as follows: first the combination of the HOL theorem prover and the BuDDy BDD package is outlined, then some basic programming techniques using the combination are described, then some elementary reachability concepts are reviewed, and finally three case studies are summarised: a treatment of the classical Missionaries and Cannibals problem, a simple autopilot (previously analysed using PVS) and an asynchronous arbiter (previously analysed with Voss).

2 HOL+BuDDy

As part of a project on formal verification of Verilog programs, Ken Larsen interfaced Jørn Lind-Nielsen's BuDDy BDD package[2], which is written in C, to Moscow ML[3]. The BuDDy package provides state-of-the-art implementations of standard BDD algorithms [4]. The Moscow ML interface to BuDDy provides ML functions for constructing and manipulating BDDs. The storage management of ML and BuDDy is linked: for example, whenever a BDD is garbage collected by ML, its reference count in BuDDy is decreased and hence it may be subsequently garbage collected by the BDD package. BDDs are made available in Moscow ML via an ML type *bdd* representing nodes in BuDDy's BDD space together with operations for creating and manipulating ML values representing values of type *bdd*.

Building on the Moscow ML interface to BuDDy, a connection between BDDs and higher order logic has been implemented [13]. How this works is discussed in some detail later.[4] Two functions are provided:

`termToBdd` : $term \to bdd$
`bddToTerm` : $bdd \to term$

The function `termToBdd` tries to represent a term as a BDD using a variable ordering held in an extensible ML datastructure called the *variable map* and a database of previously computed BDDs called the *BDD table*. An exception is raised if a BDD cannot be computed. An explicit variable order can be declared. Without such a declaration, variables are given the order in which they are encountered. A side-effect of a call to `termToBdd` is to add any previously unseen variables to the variable map. The function `bddToTerm` is total and creates a nested conditional corresponding to a BDD. The functions `termToBdd` and `bddToTerm` must be used carefully as they can cause enormous structures to be generated that exhaust available space.

The oracle function

`bddOracle` : $term \to thm$

returns the theorem $\vdash t$, if `termToBdd` t successfully evaluates to the BDD representing T and returns $\vdash \neg t$ if `termToBdd` t successfully evaluates to the BDD representing F. *This function is the only way that HOL theorems*

[2]`http://cs.it.dtu.dk/buddy/`

[3]`http://www.dina.kvl.dk/~sestoft/mosml.html`

[4]The system described here is undergoing development and rationalisation and so the details in this paper (such as identifier names) might change.

can be created via BuDDy. Theorems created using `bddOracle` are marked as having been created by BuDDy and this mark is propagated to any theorems deduced from it, so that the provenance of theorems is explicit (i.e. whether they are proved using only the rules of higher order logic, or proved from theorems created using the BDD oracle).

To use BuDDy to check whether t_1 and t_2 are equivalent, it is sufficient to check that `bddOracle` $(t_1 = t_2)$ evaluates to $\vdash t_1 = t_2$. The ML function `EqCheck` : *term* × *term* → *bool*, which is used later, is pre-defined to do this.

The domain of `termToBdd` includes quantified Boolean formulae (QBFs), that is terms built out of variables and the constants T and F using propositional operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\equiv$, $\forall$, $\exists$ (the quantified formula $\forall b{:}bool.\ P(b)$ is equivalent to $P(\mathsf{T}) \wedge P(\mathsf{F})$ and $\exists b{:}bool.\ P(b)$ to $P(\mathsf{T}) \vee P(\mathsf{F})$).

Specifications of systems are not normally single formulae, but are structured hierarchically using components. For example, hardware designs are often represented by hieraries of equations, such as

$$\begin{array}{lll} \mathsf{D}(\overline{u}) & = & \exists \overline{v}.\ \mathsf{D}_1(\overline{u},\overline{v}) \ \wedge\ \cdots\ \wedge\ \mathsf{D}_n(\overline{u},\overline{v}) \\ \mathsf{D}_1(\overline{u},\overline{v}) & = & \exists\ \cdots \\ & \vdots & \\ \mathsf{D}_n(\overline{u},\overline{v}) & = & \exists\ \cdots \end{array}$$

where $\overline{v}$ is the tuple of variables internal to D. If the definition of $\mathsf{D}(\overline{u})$ can be unfolded to a Boolean formula (e.g. by rewriting with the definitions of D_i and their components), then the BDD of $\mathsf{D}(\overline{u})$ can be computed. However, this can be a bad idea: (i) it might be that the term that results from unfolding the definitions of all the components of D is too big to fit inside HOL, even though the BDD of the term can be represented in BuDDy, and (ii) the BDDs of the components might have already been computed (e.g. with optimised variable orderings) and then it could be more efficient to compute the BDD of D by applying standard BDD algorithms to the BDDs of D_i than to compute D's BDD from scratch.

A new entry is added to the BDD table using the function

`addEquation` : *thm* → *term* × *bdd*

Evaluating `addEquation`($\vdash t_1 = t_2$) applies `termToBdd` to t_2 to compute a BDD, b_2 say, and then stores the association $\{t_1 \mapsto b_2\}$ in the BDD table. The pair (t_1, b_2) is returned.[5] Normally `addEquation` is used on definitional equations: t_1 is defined as equal to t_2, but the implemented mechanism

[5]The call of `termToBddd` may extend the variable map. An exception is raised by

currently supports arbitrary equations, as long as the right hand side, t_2, can be converted to a BDD (i.e. is built out of subterms that have already have BDDs in the table).

It is anticipated that in some demanding CAD applications, it may be necessary to compute a BDD directly inside BuDDy rather than going via terms. A mechanism for introducing new constants to name such BDDs is planned, but not yet implemented. This will provide a facility for making definitions where the right hand side of the defining equation is an externally created BDD rather than a term.

The implementation of `termToBdd` uses the BDD map to look up previously computed BDDs. In the ML pseudo code below, note that `termToBdd` *t* first checks whether *t* already has a BDD and returns the pre-computed BDD if it exists. If *t* doesn't have a pre-computed BDD, then `termToBdd` checks whether some other term that can be instantiated to *t* has a BDD, and if so the BDD of this term is appropriately instantiated (using BuDDy operations). The implementation of the BDD table is designed so that lookups and search for matches are efficient. The algorithm used by `termToBdd`, in outline, is

```
fun termToBdd $t$ =
 if $t$ is in the BDD table
  then return corresponding BDD else
 if $t$ matches a term in the BDD table
  then return corresponding BDD instance else
 if $t$ is a new variable
  then add variable to the variable map and then
       return BDD corresponding to new variable else
 if $t$ is an existing variable in the variable map
  then return corresponding BDD else
 if $t = (\lambda v.\ t_1)t_2$
  then apply BDD composition to termToBdd $t_1$, termToBdd $t_2$, $v$ else
 if $t = \neg t_1$
  then apply BDD negation to termToBdd $t_1$ else
 if $t = t_1\ op\ t_2$
  then apply BDD algorithm for op to termToBdd $t_1$, termToBdd $t_2$ else
     ⋮
  raise holToBddError
```

BuDDy provides algorithms for certain combinations of logical operators (e.g. quantified conjunctions and disjunctions). The implemented `termToBdd` detects such combinations and uses the optimised algorithm.

`addEquation` if it is not applied to an equation or if `termToBdd` fails.

The BDD associated with a term in the BDD table can be removed using the function

`deleteBdd` : *term*→*bdd*

Removing terms from the BDD table may enable the BuDDy garbage collector to reclaim space. Not all side effects resulting from adding a term to the BDD table are undone by `deleteBdd` – in particular, any extensions to the global variable ordering made when the term was added will persist.

2.1 Relation to Voss

Carl Seger's Voss system [24] has been particularly influential on the work 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) [15]. The HOL-Voss system integrates HOL 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 HOL and Voss term representations.

In later work Lee, Seger and Greenstreet [16] showed how various optimised BDD algorithms could be programmed in FL. The arbiter example in Section 5.2 is taken from this work.

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 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 [14]. VossProver was used to verify impressive integer and floating-point examples (see the the DAC98 paper by Aagaard, Jones and Seger [1] for further discussion and references). After Seger and Aagaard moved to Intel, the development of Voss and VossProver evolved into a system called Forte that "is an LCF-style implementation of a higher-order classical logic" and "seamlessly integrates several types of model-checking engines with lightweight theorem proving and extensive debugging capabilities, creating a productive high-capacity formal verification environment". Only partial details of this are in the public domain [20, 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 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. Research with Forte has resulted in major hardware verification case studies.

The combination of HOL and BuDDy provides a similar programming environment to Voss's FL (though with eager rather than lazy evaluation). 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 HOL+BuDDy. 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 (see the description of 'early quantification' in the Section 5.2). This is in line with future developments discussed by Joyce and Seger [15] and it appears that the Forte system has similar capabilities. The approach described here of adding Voss-like facilities into HOL is dual to adding deductive theorem proving into Voss.

3 Transition systems

Many systems can be modelled by a state space, a set of initial states and a state-transition relation. The state space can be represented as a type, *states* say, the transition relation as a predicate $\mathcal{R} : \mathit{states} \times \mathit{states} \to \mathit{bool}$, and the set of initial states as a predicate $\mathcal{B} : \mathit{states} \to \mathit{bool}$. Here, *bool* is the type of the two truth-values T and F. The term $\mathcal{R}(s, s')$ means s' is a successor to s, and $\mathcal{B}\ s$ means s is an initial state. This representation in higher order logic is similat to that used for integrating model-checking into PVS [23].

An example of a state transition system is a single machine defined by a next-state function $\delta : \mathit{states} \times \mathit{inputs} \to \mathit{states}$. The state transition relation

$\mathcal{R}_\delta$ for such a machine is defined by $\mathcal{R}_\delta(s, s') = \exists inp.\ s' = \delta(s, inp)$.

A deterministic machine gives rise to a non-deterministic transition relation via existential quantification over inputs. This is called *input non-determinism.* The autopilot example in Section 5.3 is an example of a single machine.

A more interesting class of state transition systems corresponds to n machines running in parallel. Assume n machines, each with states $states_1, \ldots, states_n$. The combined state is the Cartesian product: $states = states_1 \times \cdots \times states_n$. Assume n transition functions: $\delta_i : states \times inputs \rightarrow states_i \quad (1 \leq i \leq n)$. Write $\overline{v}$ for the state $(v_1, \ldots, v_n)$. The transition relation of the asynchronous parallel composition of the n machines is defined by

$$
\begin{array}{rcl}
\mathcal{R}(\overline{v}, \overline{v}') & = & \exists inp.\ \ v_1' = \delta_1(\overline{v}, inp) \ \wedge\ v_2' = v_2 \ \wedge \cdots \wedge\ v_n' = v_n \\
& & \qquad\ \ \vee \\
& & \qquad\ \ v_1' = v_1 \ \wedge\ v_2' = \delta_2(\overline{v}, inp) \ \wedge \cdots \wedge\ v_n' = v_n \\
& & \qquad\ \ \vee \\
& & \qquad\ \ \vdots \\
& & \qquad\ \ \vee \\
& & \qquad\ \ v_1' = v_1 \ \wedge\ v_2' = v_2 \ \wedge \cdots \wedge\ v_n' = \delta_n(\overline{v}, inp)
\end{array}
$$

An $\mathcal{R}$-step is a δ_i-step for some i.

In any state transition system a state s is reachable in one $\mathcal{R}$-step from a state in $\mathcal{B}$ if $\exists u.\ \mathcal{B}\ u \ \wedge\ \mathcal{R}(u, s)$. The set of states reachable in at most n steps is defined recursively by

$$\mathsf{ReachBy}\ 0\ \mathcal{R}\ \mathcal{B}\ s \ = \ \mathcal{B}\ s$$

$$
\begin{array}{l}
\mathsf{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}\ s \ = \\
\quad \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s \ \vee\ \exists u.\ \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ u \ \wedge\ \mathcal{R}(u, s)
\end{array}
$$

The set of reachable states is then defined as the set of states reachable in some number of steps: $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s \ = \ \exists n.\ \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s$.

4 Programming with BDDs in HOL

One way of applying BDDs to transition systems [18, 10] is to represent the Boolean terms $\mathcal{R}(s, s')$ and $\mathcal{B}\ s$ as BDDs, where the state vector s and next state vector s' are tuples of Boolean variables. The BDD of $\mathsf{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}\ s$ is computed from that of $\mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s$ using standard BDD operations. As i increases $\mathsf{ReachBy}\ i\ \mathcal{R}\ \mathcal{B}\ s$ represent increasing sets, so if the set of states is finite then there must be an n such that

$$\mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s\ =\ \mathsf{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}\ s$$

It easily follows in HOL from the definitions of ReachBy and Reach that

$$\vdash (\mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s\ =\ \mathsf{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}\ s) \Rightarrow$$
$$(\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s\ =\ \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s)$$

Thus if $\mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s\ =\ \mathsf{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}\ s$ then it follows by Modus Ponens that $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s\ =\ \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s$.

The implementation in HOL+BuDDy of a function to compute $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s$ is described below. This takes a definition of a transition relation in the form of a theorem $\vdash \mathcal{R}(s,s')\ =\ t_1$ and a definition of a set of initial states in the form of a theorem $\vdash \mathcal{B}\ s\ =\ t_2$ and returns a pair (t,b), where t is the term $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s$ and b is its BDD. As a side effect the BDD table is updated with b as the BDD of t.

In the simplified ML definition that follows, the HOL inference rule `SPEC` performs $\forall$-instantiation (i.e. `SPEC` t $(\vdash \forall x.\ P(x))\ =\ \vdash P(t)$) and the HOL inference rule `MP` does Modus Ponens (i.e. `MP` $(\vdash A{\Rightarrow}B)$ $(\vdash A)\ =\ \vdash B$)

```
fun ComputeReachableStates(⊢ $\mathcal{R}(s,s')$ = $t_1$, ⊢ $\mathcal{B}$ $s$ = $t_2$) =
 let val ReachByThm =
      ⊢ ∀$n$. ReachBy ($n$+1) $\mathcal{R}$ $\mathcal{B}$ $s$ =
            ReachBy $n$ $\mathcal{R}$ $\mathcal{B}$ $s$ ∨
            ∃$u$. ReachBy $n$ $\mathcal{R}$ $\mathcal{B}$ $u$ ∧ $\mathcal{R}(u,s)$
     val FixedPointThm =
      ⊢ ∀$n$. (ReachBy $n$ $\mathcal{R}$ $\mathcal{B}$ $s$ = ReachBy ($n$+1) $\mathcal{R}$ $\mathcal{B}$ $s$)
            ⇒
           (Reach $\mathcal{R}$ $\mathcal{B}$ $s$ = ReachBy $n$ $\mathcal{R}$ $\mathcal{B}$ $s$)
     fun Iterate $n$ =
      (addEquation (SPEC $n$ ReachByThm);
       if EqCheck(ReachBy $n$ $\mathcal{R}$ $\mathcal{B}$ $s$, ReachBy ($n$+1) $\mathcal{R}$ $\mathcal{B}$ $s$)
       then addEquation
            (MP (SPEC $n$ FixedPointThm)
                (bddOracle
                  ReachBy $n$ $\mathcal{R}$ $\mathcal{B}$ $s$ = ReachBy ($n$+1) $\mathcal{R}$ $\mathcal{B}$ $s$))
       else (deleteBdd(ReachBy $n$ $\mathcal{R}$ $\mathcal{B}$ $s$);
            Iterate($n$+1)))
 in addEquation ⊢ $\mathcal{R}(s,s')$ = $t_1$;
    addEquation ⊢ $\mathcal{B}$ $s$ = $t_2$;
    addEquation ⊢ ReachBy 0 $\mathcal{R}$ $\mathcal{B}$ $s$ = $\mathcal{B}$ $s$;
    Iterate 0
 end
```

This function is an intimate mixture of BuDDy and HOL operations. The starting point is the HOL definitions of the transition relation and set of initial states. The right hand side of these definitions are converted to BDDs (using `addEquation`) and then a standard fixed-point computation is done using BuDDy. This mixes the application of HOL inference rules with BuDDy BDD operations. For successive values of n, starting from 0, the BDD of $\mathsf{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}\ s$ is computed by adding `SPEC` n `ReachByThm` to the BDD table, which forces a computation of $\mathsf{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}\ s$ from the already computed BDD of $\mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s$ and the BDD of $\mathcal{R}(s, s')$. When the `EqCheck` succeeds the fixed point has been reached and then `bddOracle` is used to create a HOL theorem:

$$\vdash \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s\ =\ \mathsf{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}\ s$$

and then by Modus Ponens with `FixedPointThm` the equation

$$\vdash \mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s\ =\ \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s$$

is derived. Finally, this equation is added to the BDD table, so that the BDD of $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s$ is available.

A common verification problem is to show some property $\mathcal{P}$ is true in all reachable states. This can be verified by checking whether $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s \Rightarrow \mathcal{P}\ s$ is a tautology. Because the BDD of $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s$ is already in the BDD table, its BDD can just be looked up. Only the BDD of $\mathcal{P}\ s$ needs to be computed from scratch (and maybe even this is not even necessary as $\mathcal{P}\ s$ might already have sub-terms that match terms in the BDD table).

Proving the equality of $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s$ and `bddToTerm(termToBdd` $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s$) using `bddOracle` results in a theorem giving an explicit formula for the set of reachable states. This formula might be huge, but examples illustrating its use are given in Sections 5.1 and 5.3 below. Since the BDD of $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s$ is already in the BDD table, `termToBdd` will just perform a lookup.

Because the fixed point may have been taken in a Boolean encoding of the original problem (see the autopilot example below), further HOL deduction may be needed to convert it back to the unencoded datatypes.

The Function `ComputeReachableStates` cannot produce wrong results as long as the HOL system, the BuDDy system and the implementation of `termToBdd` and `addEquation` are correct. This is in the spirit of LCF: the architecture of the system makes it impossible for the user to prove false theorems.

4.1 Counter-examples and other sequences of states

Suppose $\mathsf{Reach}\ \mathcal{R}\ \mathcal{B}\ s \Rightarrow \mathcal{P}\ s$ is false. Then BDD calculations can be used to find a shortest sequence of state transitions that lead to a reachable state in which $\mathcal{P}$ fails to hold. Such debugging is perhaps even more useful than verification.

The first step is to find a shortest path to a counter-example. Using a similar fixed-point calculation to the one above, BDDs can be successively generated for $\mathsf{ReachBy}\ i\ \mathcal{R}\ \mathcal{B}\ s$ $(i = 0, 1, \ldots)$ and for each i it can be checked whether $\mathcal{P}\ s$ holds, that is whether the BDD of $\mathsf{ReachBy}\ i\ \mathcal{R}\ \mathcal{B}\ s \Rightarrow \mathcal{P}\ s$ is true. Eventually a smallest n such that $\mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ s \Rightarrow \mathcal{P}\ s$ is not true will be found.

The HOL+BuDDy system provides an ML function

`findModel` : *bdd→term*

which finds values for variables in a BDD that make it true (an exception is raised if the BDD is a contradiction). The term returned by `findModel` is a conjunction of variables or negated variables. If $t[v_1, \ldots, v_m]$ is a term containing free variables v_1, …, v_m, then, abbreviating $\overline{v} = v_1, \ldots, v_m$, `findModel(termToBdd` $t[\overline{v}]$`)` will return a conjunction of v_i or $\neg v_i$ (for some values of i), $u[\overline{v}]$ say, such that $\vdash u[\overline{v}] \Rightarrow t[\overline{v}]$. From this it follows that $\vdash t[\overline{c}]$, where $\overline{c} = c_1, \ldots, c_m$ and c_i is T if v_i occurs positively in $u[\overline{v}]$ and is F otherwise.

Using `findModel`, a vector of values $\overline{c}_n$ such that $\mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ \overline{c}_n \wedge \neg(\mathcal{P}\ \overline{c}_n)$ can be found. A sequence of vectors representing states starting from an initial state and ending in $\overline{c}_n$ can then be generated by tracing backwards from $\overline{c}_n$. Define

$$\begin{aligned} \mathsf{Prev}\ \mathcal{R}\ \mathcal{P}\ s &= \exists s'.\ \mathcal{R}(s, s') \ \wedge\ \mathcal{P}\ s' \\ \mathsf{Eq}\ s_1\ s_2 &= (s_1 = s_2) \end{aligned}$$

then iteratively construct a sequence $\overline{c}_n, \ldots, \overline{c}_0$, where, given $\overline{c}_i$, `findModel` is used as above to compute $\overline{c}_{i-1}$ from the BDD of

$\mathsf{ReachBy}\ (i{-}1)\ \mathcal{R}\ \mathcal{B}\ s\ \wedge\ \mathsf{Prev}\ \mathcal{R}\ (\mathsf{Eq}\ \overline{c}_i)\ s$

The BDDs for $\mathsf{ReachBy}\ i\ \mathcal{R}\ \mathcal{B}\ s$ (for $i = 0, 1, \ldots, n$) were previously computed when searching for s_n, so they are already available via the BDD table. The BDD for $\mathsf{Prev}\ \mathcal{R}\ (\mathsf{Eq}\ \overline{c}_i)\ s$ is computed from the BDDs for $\mathcal{R}(s, s')$ and $\mathsf{Eq}(s_1, s_2)$ by `termToBdd`.

The sequence $\overline{c}_0, \ldots, \overline{c}_n$ is a trace from an initial state to a counter-example. The following properties can be deduced

$\vdash \mathcal{B}\ \overline{c}_0$

$\vdash \mathsf{ReachBy}\ i\ \mathcal{R}\ \mathcal{B}\ \overline{c}_i\ \wedge\ \mathcal{R}(\overline{c}_i, \overline{c}_{i+1}) \qquad (0 < i < n)$

$\vdash \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ \overline{c}_n\ \wedge\ \neg(\mathcal{P}\ \overline{c}_n)$

For debugging, it seems overkill to verify that the computed sequence of states to a counter-example is correct, since the purpose of the counter-example is just to point at bugs. However, something similar to the procedure just described could possibly be used to synthesize program code to achieve a specification, and with this kind of application it may be useful to generate a proof that it is correct, especially if the final code is obtained by transforming the output of the raw BDD calculation into some different format. The Missionaries and Cannibals example below is suggestive: a schedule of boat trips is a kind of program.

5 Examples

In this section three rather different examples are outlined in varying degrees of detail. The Missionaries and Cannibals Problem is from the early Artificial Intelligence problem solving literature [3]. The autopilot is a simplified example of avionics system design validation, originally devised as a tutorial example for the PVS theorem prover [8]. The arbiter is a hardware verification case study [17, 16] used to illustrate BDD programming with Voss.

5.1 Missionaries and Cannibals Problem

The original Missionaries and Cannibals Problem (MCP) is:

> *Three missionaries and three cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten.*
>
> *How shall they cross?* [6]

This problem can be generalised to have n missionaries and n cannibals and a boat of capacity k. Call this problem $MCP(n, k)$. The problem stated above is then $MCP(3, 2)$. If $k > 2$, then in the generalised problem it is assumed that cannibals must not outnumber missionaries in the boat. Assume without loss of generality that in the initial state, the missionaries, the cannibals and

[6]Problem statement from `www-formal.stanford.edu/jmc/elaboration/node2.html`

the boat are on the left bank. Let a state (m, c, b) be represented by the number m of missionaries on the left bank, the number c of cannibals on the left bank and the position b of the boat (where b being true means 'boat at left bank').

The initial state is represented by (n, n, T) and the goal state by $(0, 0, \mathsf{F})$. A predicate B_{MCP} characterizing the initial state is thus defined by

$$\mathsf{B}_{MCP}\ n\ (m, c, b) \ = \ m{=}n \ \wedge \ c{=}n \ \wedge \ b$$

The possible state transitions are

MoveRight:	*move u missionaries and v cannibals to right bank*
MoveLeft:	*move u missionaries and v cannibals to left bank*

Consider MoveRight. The boat must be on the left bank, so b. The people embarking on the boat must be a subset of those who were on the left bank, so $u \leq m \wedge v \leq c$. Cannibals must not eat missionaries in the boat, so $\neg(0 < u < v)$. There must be at least one person to operate the boat, so $0 < u{+}v$. The capacity of the boat must not be exceeded, so $u{+}v \leq k$. If all these pre-conditions are met, then a trip can take place and in the resulting state $m' = m{-}u \ \wedge \ c' = c{-}v \ \wedge \ \neg b'$. Thus MoveRight is defined by

$$\begin{array}{l} \mathsf{MoveRight}\ n\ k\ (u, v)\ ((m, c, b), (m', c', b')) \ = \\ b \ \wedge \ u \leq m \ \wedge \ v \leq c \ \wedge \ \neg(0 < u \ \wedge \ u < v) \ \wedge \ 0 < u{+}v \ \wedge \ u{+}v \leq k \\ \wedge \\ m' = m{-}u \ \wedge \ c' = c{-}v \ \wedge \ \neg b' \end{array}$$

Similarly MoveLeft is defined by

$$\begin{array}{l} \mathsf{MoveLeft}\ n\ k\ (u, v)\ ((m, c, b), (m', c', b')) \ = \\ \neg b \ \wedge \ u \leq n{-}m \ \wedge \ v \leq n{-}c \ \wedge \ \neg(0 < u \ \wedge \ u < v) \ \wedge \ 0 < u{+}v \ \wedge \ u{+}v \leq k \\ \wedge \\ m' = m{+}u \ \wedge \ c' = c{+}v \ \wedge \ b' \end{array}$$

Cannibals can also eat missionaries if there are more of them on either the left bank $(0 < m < c)$ or the right bank $(0 < (n{-}m) < (n{-}c))$ Define $\mathsf{Eat}\ n\ (m, c)$ to be true if this can happen.

$$\mathsf{Eat}\ n\ (m, c, b) \ = \ (0 < m \ \wedge \ m < c) \ \vee \ (0 < (n{-}m) \ \wedge \ (n{-}m) < (n{-}c))$$

Thus the transition relation is

$$\begin{array}{l} \mathsf{R}_{MCP}\ n\ k\ (s, s') \ = \\ \neg\mathsf{Eat}\ n\ s' \ \wedge \ (\exists u\ v.\ u \leq n \ \wedge \ v \leq n \ \wedge \ \mathsf{MoveRight}\ n\ k\ (u, v)\ (s, s')) \\ \qquad\qquad\quad \wedge \ (\exists u\ v.\ u \leq n \ \wedge \ v \leq n \ \wedge \ \mathsf{MoveLeft}\ n\ k\ (u, v)\ (s, s')) \end{array}$$

The missionaries and cannibals problem $MCP(n,k)$ is solvable if the goal state $(0,0,\mathsf{F})$ is reachable from the initial state, that is if

$\vdash \mathsf{Reach}\ (\mathsf{R}_{MCP}\ n\ k)\ (\mathsf{B}_{MCP}\ n)\ (0,0,\mathsf{F})$

A few things are immediately obvious: $MCP(n,0)$ is not solvable, because the boat is useless; $MCP(n,1)$ is also not solvable because one can never get more than one person to the right bank (the first person to go to the right bank will have to return as the boat needs at least one occupant). It is also clear that $MCP(n,k)$ is solvable if $k \geq 4$, since one missionary and one cannibal can act as boatmen and ferry pairs of missionaries and cannibals from left to right.

Let us consider $MCP(n,k)$ where n and k are less than 16, (i.e. representable with 4 bits). It is easy to convert the problem as stated above to a finite state one by restricting m, c, u and v to 4-bit words.[7] If w is such a word, denote its least significant bit by w_0 and its most significant bit by w_3.

`ComputeReachableState` automatically gives

```
⊢ Reach
    (R_MCP (n3,n2,n1,n0) (k3,k2,k1,k0))
    (B_MCP (n3,n2,n1,n0) (k3,k2,k1,k0))
    ((0,0,0,0),(0,0,0,0),F) =
    if n3 then k3 ∨ k2
         else (if n2 then (if n1 then k3 ∨ k2 else k3 ∨ k2 ∨ k1 ∧ k0)
                    else (if n1 then k3 ∨ k2 ∨ k1
                               else n0 ∧ (k3 ∨ k2 ∨ k1)))
```

Abstracting to numbers

$$\begin{array}{l} n < 16\ \wedge\ k < 16 \Rightarrow \\ \quad \mathsf{Reach}\ (\mathsf{R}_{MCP}\ n\ k)\ (\mathsf{B}_{MCP}\ n)\ (0,0,\mathsf{F})\ = \\ \quad 0 < n\ \wedge\ 0 < k\ \wedge\ (k = 2 \Rightarrow n < 4)\ \wedge\ (k = 3 \Rightarrow n < 6) \end{array}$$

This can be checked with n and k interpreted as 4-bit words. Work is in progress to provide infrastructure to make it easy to prove it for numbers (it's just pure HOL deduction).

From this it appears that there are four interesting solvable cases: $MCP(2,2)$, $MCP(3,2)$, $MCP(4,3)$ and $MCP(5,3)$. Finding solutions is just like finding counterexamples. Using the method described earlier, HOL+BuDDy computes the following list of theorems for $MCP(2,2)$

[7]To avoid overflow the condition $u{+}v \leq k$ should be replaced by $u \leq k{-}v\ \wedge\ v \leq k$, and $0 < u{+}v$ by $0 < u\ \vee\ 0 < v$.

```
[⊢ℬ (W4 2) (W4 2, W4 2, T),
 ⊢Prev (ℛ (W4 2) (W4 2)) (Eq (W4 1, W4 1, F)) (W4 2, W4 2, T),
 ⊢Prev (ℛ (W4 2) (W4 2)) (Eq (W4 2, W4 1, T)) (W4 1, W4 1, F),
 ⊢Prev (ℛ (W4 2) (W4 2)) (Eq (W4 0, W4 1, F)) (W4 2, W4 1, T),
 ⊢Prev (ℛ (W4 2) (W4 2)) (Eq (W4 0, W4 2, T)) (W4 0, W4 1, F),
 ⊢Prev (ℛ (W4 2) (W4 2)) (Eq (W4 0, W4 0, F)) (W4 0, W4 2, T)]
```

Here W4 n is the 4-bit binary representation (n_3, n_3, n_1, n_0). This list of theorems specifies the solution[8] 22T→11F→21T→01F→02T→00F. The other solutions found by HOL+BuDDy are

$MCP(3,2)$:
33T→22F→32T→30F→31T→11F→22T→02F→03T→01F→02T→00F
$MCP(4,3)$
44T→41F→42T→22F→33T→03F→04T→01F→02T→00F
$MCP(5,3)$
55T→52F→53T→50F→52T→22F→33T→03F→04T→01F→02T→00F

The reachability calculations only check that for $n < 16$ there are no solutions of $MCP(n, 2)$ when $n \geq 4$ and no solutions of $MCP(n, 3)$ when $n \geq 6$.

Anuj Dawar, on a flight back from Italy, came up with the following argument showing that there are no solutions when $k < 4$ and $n \geq 2k$. Suppose M_L and C_L are the numbers of missionaries and cannibals on the left bank, respectively, and M_R and C_R are the numbers of missionaries and cannibals on the right bank, respectively. Consider the last time M_R changes from being 0. In the resulting state all the missionaries cannot have arrived at the right bank, since there are more missionaries than the capacity k of the boat $(n \geq 2k)$. In fact, at most k missionaries can have arrived, so $M_R \leq k$. Thus there must be some missionaries remaining on the left bank, and hence fewer cannibals on the left bank. All the cannibals cannot be on the right bank, because if they were they would eat the missionaries that have just arrived. Thus there are missionaries and cannibals on both banks, hence $M_L = C_L$ and $M_R = C_R$.

Now, on the return trip of the boat, not all M_R missionaries may travel, as this would contradict the assumption that M_R is henceforth non-zero. If any missionaries travel, at least that number of cannibals must travel so that the remaining ones are not outnumbered, and if any cannibals travel, at least that number of missionaries must travel so that missionaries on the first bank are

[8] mcb abbreviates (m, c, b).

not outnumbered. This means that exactly one missionary and one cannibal can travel, (since $k < 4$).

This leaves $M_R < k$, and therefore $M_L > n-k$, i.e. $M_L > k$ (since $n \geq 2k$). Thus, all the remaining M_L missionaries cannot travel on the next trip. But then, an equal number of missionaries and cannibals must travel, which means exactly one missionary and one cannibal travel, but this just reverses the last move.

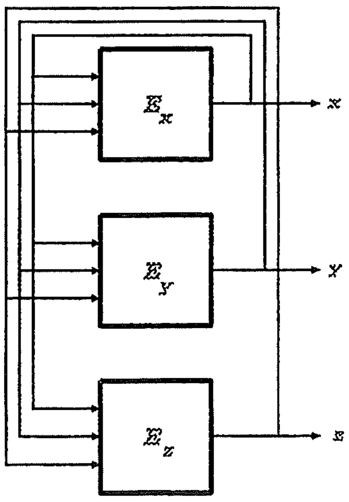
A nice exercise would be to perform this proof in HOL, thereby getting a complete machine checked analysis of $MCP(n, k)$, with all the finite number of solutions computed using BDDs and the other cases shown to be impossible by deduction. This would prove for all n and k that

$$\begin{aligned}&\mathsf{Reach}\ (\mathsf{R}_{MCP}\ n\ k)\ (\mathsf{B}_{MCP}\ n)\ (0, 0, \mathsf{F})\ = \\ &0 < n\ \wedge\ 0 < k\ \wedge\ (k = 2 \Rightarrow n < 4)\ \wedge\ (k = 3 \Rightarrow n < 6)\end{aligned}$$

5.2 The Arbiter example

This is an asynchronous circuit for mutual exclusion modelled as the asynchronous parallel composition of 22 machines. Each machine corresponds to a circuit component. There is a state variable corresponding to the output of each component, hence there are 22 state variables. The transition relation is the disjunction of 22 terms, each specifying the behaviour of one of the machines. Such a specification gives the next value of the state variable corresponding to the machine's output in terms of its current inputs, together with a 'frame axiom' specifying that the other 21 state variables don't change.

For simplicity, consider three machines running asynchronously in parallel



This is modelled by

$$
\begin{array}{l}
\mathcal{R}((x,y,z),(x',y',z')) = \\
\quad (x' = E_x(x,y,z) \ \wedge \ y' = y \ \wedge \ z' = z) \vee \\
\quad (x' = x \ \wedge \ y' = E_y(x,y,z) \ \wedge \ z' = z) \vee \\
\quad (x' = x \ \wedge \ y' = y \ \wedge \ z' = E_z(x,y,z))
\end{array}
$$

Let $\mathcal{S}(\hat{x},\hat{y},\hat{z})$ abbreviate $\mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ (\hat{x},\hat{y},\hat{z})$ then

$$
\begin{array}{l}
\exists\hat{x}\ \hat{y}\ \hat{z}.\ \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ (\hat{x},\hat{y},\hat{z}) \ \wedge \ \mathcal{R}((\hat{x},\hat{y},\hat{z}),(x,y,z)) \\
\quad = \ \exists\hat{x}\ \hat{y}\ \hat{z}.\ \mathcal{S}(\hat{x},\hat{y},\hat{z}) \ \wedge \ \mathcal{R}((\hat{x},\hat{y},\hat{z}),(x,y,z)) \\
\quad = \ \exists\hat{x}\ \hat{y}\ \hat{z}.\ \mathcal{S}(\hat{x},\hat{y},\hat{z}) \ \wedge \ ((x = E_x(\hat{x},\hat{y},\hat{z}) \ \wedge \ y = \hat{y} \ \wedge \ z = \hat{z}) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (x = \hat{x} \ \wedge \ y = E_y(\hat{x},\hat{y},\hat{z}) \ \wedge \ z = \hat{z}) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (x = \hat{x} \ \wedge \ y = \hat{y} \ \wedge \ z = E_z(\hat{x},\hat{y},\hat{z}))) \\
\quad = \ (\exists\hat{x}\ \hat{y}\ \hat{z}.\ \mathcal{S}(\hat{x},\hat{y},\hat{z}) \ \wedge \ x = E_x(\hat{x},\hat{y},\hat{z}) \ \wedge \ y = \hat{y} \ \wedge \ z = \hat{z}) \vee \\
\qquad\ (\exists\hat{x}\ \hat{y}\ \hat{z}.\ \mathcal{S}(\hat{x},\hat{y},\hat{z}) \ \wedge \ x = \hat{x} \ \wedge \ y = E_y(\hat{x},\hat{y},\hat{z}) \ \wedge \ z = \hat{z}) \vee \\
\qquad\ (\exists\hat{x}\ \hat{y}\ \hat{z}.\ \mathcal{S}(\hat{x},\hat{y},\hat{z}) \ \wedge \ x = \hat{x} \ \wedge \ y = \hat{y} \ \wedge \ z = E_z(\hat{x},\hat{y},\hat{z})) \\
\quad = \ ((\exists\hat{x}.\ \mathcal{S}(\hat{x},y,z) \wedge x{=}E_x(\hat{x},y,z)) \ \wedge \ (\exists\hat{y}.\ y{=}\hat{y}) \ \wedge \ (\exists\hat{z}.\ z{=}\hat{z})) \vee \\
\qquad\ ((\exists\hat{x}.\ x{=}\hat{x}) \ \wedge \ (\exists\hat{y}.\ \mathcal{S}(x,\hat{y},z) \wedge y{=}E_y(x,\hat{y},z)) \ \wedge \ (\exists\hat{z}.\ z{=}\hat{z})) \vee \\
\qquad\ ((\exists\hat{x}.\ x{=}\hat{x}) \ \wedge \ (\exists\hat{y}.\ y{=}\hat{y}) \ \wedge \ (\exists\hat{z}.\ \mathcal{S}(x,y,\hat{z}) \wedge z{=}E_z(x,y,\hat{z}))) \\
\quad = \ (\exists\hat{x}.\ \mathcal{S}(\hat{x},y,z) \ \wedge \ x = E_x(\hat{x},y,z)) \vee \\
\qquad\ (\exists\hat{y}.\ \mathcal{S}(x,\hat{y},z) \ \wedge \ y = E_y(x,\hat{y},z)) \vee \\
\qquad\ (\exists\hat{z}.\ \mathcal{S}(x,y,\hat{z}) \ \wedge \ z = E_z(x,y,\hat{z}))
\end{array}
$$

The BDD of $\exists\hat{x}\ \hat{y}\ \hat{z}.\ \mathsf{ReachBy}\ n\ \mathcal{R}\ \mathcal{B}\ (\hat{x},\hat{y},\hat{z}) \ \wedge \ \mathcal{R}((\hat{x},\hat{y},\hat{z}),(x,y,z))$ can thus be computed without ever computing the BDD of $\mathcal{R}((\hat{x},\hat{y},\hat{z}),(x,y,z))$. This technique is called *early quantification*[9] [18, page 45] and can be done by the HOL simplifier. The definition of the function `ComputeReachableStates` is simply modified to use early quantification by replacing `ReachByThm` with the simplified equation for $\mathsf{ReachBy}\ (n{+}1)\ \mathcal{R}\ \mathcal{B}\ s$.

The usual implementation of early quantification is by writing programs that directly construct the BDD of the simplified term. The logical transformation corresponding to early quantification are thus encoded in BDD building code. The approach here is to deductively simplify the next-state relation prior to invoking `termToBdd`. The advantages are that the simplification is guaranteed sound.

It took HOL+BuDDy 4891 seconds to build the BDD of the transition rela-

[9]In the theorem proving literature early quantification is called 'miniscoping' and can reduce proof length more than exponentially [21].

tion for the arbiter, which has 8,399,564 nodes (using the variable order given in Lee's thesis). Using early quantification to simplifiy ReachBy $(n{+}1)$ $\mathcal{R}$ $\mathcal{B}$ s resulted in the biggest BDD that needed to be computed having 5338 nodes. The BDD of Reach $\mathcal{R}$ $\mathcal{B}$ s has 1768 nodes. It takes 46 seconds[10] to compute the set of reachable states, of which 36 seconds is HOL applying the early quantification simplification and the 10 additional seconds to perform the 62 iterations needed to reach the fixed-point.

In Lee's thesis [17] two arbiters are given: a correct one that is verified and an incorrect one for which a counter-example is computed. The algorithm for computing the counter-example trace employs hand coded Voss FL programs to compute weakest pre-conditions [16].

Early quantification can also be used to find paths to counter-examples: Prev $\mathcal{R}$ $\mathcal{P}$ s can be deductively simplified, so that its BDD can be computed without having to compute the BDD of the transition relation $\mathcal{R}$. Using early quantification applied to Prev $\mathcal{R}$ $\mathcal{P}$ s we were able to generate the same counter-example as Lee without any special weakest pre-condition BDD algorithms.

5.3 A simple autopilot

The autopilot example has been used to illustrate requirements capture and formal validation with PVS [9]. It has also been modelled and validated with ACL2 [25]. The treatment in higher order logic outlined here derives from a 'port' of the PVS specification and proofs to HOL by Mark Staples and Konrad Slind. The autopilot is a single machine specified by a next state function Next : *states*$\rightarrow$(*events*$\rightarrow$*states*), where the types *states* and *events* (i.e. inputs) are specified abstractly as enumerated types. There are 576 states and 10 events. An initial set of states is specified by a predicate Initial : *states*$\rightarrow$*bool*. The goal of the verification is to establish that all reachable states satisfy a predicate Valid : *states*$\rightarrow$*bool*. A transition relation Trans is defined in the usual way: Trans$(s, s') \; = \; \exists e.\ s' = $ Next s e.

It is easy to show by deduction that $\vdash \forall s.$ Reach Trans Initial $s \Rightarrow$ Valid s. PVS, ACL2 and HOL can all verify this automaticallly. Thus just for verification there is no need to use automatic state exploration methods. However, using them one can derive additional information that is hard to get from a theorem prover. For example, it is possible to automatically compute a counter-

[10]Times are for my year-old office PC running Linux.

example to $\forall s.$ Valid $s \Rightarrow$ Reach Trans Initial s and to prove an equation giving an explicit term for Reach Trans Initial s.

The specification of the autopilot is expressed in terms of high-level datatypes. To apply BDD algorithms, these datatypes need to be encoded as tuples of Booleans. This can all be done by deduction inside HOL. The details cannot be explained without the complete definitions making up the autopilot, but we shall try to give an outline.

The abstract type *states* can be encoded as a subset of *rep_states*, where

$$\mathit{rep\_states} =$$
$$\mathit{bool}\times\mathit{bool}\times\mathit{bool}\times(\mathit{bool}\times\mathit{bool})\times\mathit{bool}\times\mathit{bool}\times\mathit{bool}\times(\mathit{bool}\times\mathit{bool})$$

The representation is characterized by functions

$$\begin{array}{ll}\mathsf{Rep\_states} : & \mathit{states}\rightarrow\mathit{rep\_states}\\ \mathsf{Abs\_states} : & \mathit{rep\_states}\rightarrow\mathit{states}\\ \mathsf{Dom\_states} : & \mathit{rep\_states}\rightarrow\mathit{bool}\end{array}$$

that satisfy

$$\vdash \forall a.\ \mathsf{Abs\_states}(\mathsf{Rep\_states}\ a) = a$$
$$\vdash \forall r.\ \mathsf{Dom\_states}\ r = (\mathsf{Rep\_states}(\mathsf{Abs\_states}\ r) = r)$$

The type *rep_states* and the representation, abstraction and domain functions are defined automatically from the HOL definitions of the high-level datatypes.

Versions of Trans, Initial and Valid that operate on Booleans are defined by

$$\vdash \mathsf{Rep\_Trans}(r,r') =$$
$$\mathsf{Dom\_states}\ r \wedge \mathsf{Dom\_states}\ r' \wedge \mathsf{Trans}(\mathsf{Abs\_states}\ r, \mathsf{Abs\_states}\ r')$$
$$\vdash \mathsf{Rep\_Initial}\ r =$$
$$\mathsf{Dom\_states}\ r \wedge \mathsf{Initial}(\mathsf{Abs\_states}\ r)$$
$$\vdash \mathsf{Rep\_Valid}\ r =$$
$$\mathsf{Dom\_states}\ r \wedge \mathsf{Valid}(\mathsf{Abs\_states}\ r)$$

Equations showing the validity of the representation can be derived by deductive proof:

$$\vdash \mathsf{Trans}(s,s') = \mathsf{Rep\_Trans}(\mathsf{Rep\_states}\ s, \mathsf{Rep\_states}\ s')$$
$$\vdash \mathsf{Initial}\ s = \mathsf{Rep\_Initial}(\mathsf{Rep\_states}\ s)$$
$$\vdash \mathsf{Valid}\ s = \mathsf{Rep\_Valid}(\mathsf{Rep\_states}\ s)$$
$$\vdash \mathsf{Reach\ Trans\ Initial}\ s = \mathsf{Reach\ Rep\_Trans\ Rep\_Initial}\ (\mathsf{Rep\_states}\ s)$$

To find a counter-example to $\forall s.$ Valid $s \Rightarrow$ Reach Trans Initial s the following higher-order theorem is used

$\vdash \forall P.\ (\forall s.\ P\ s)\ =\ \forall r.\ \mathsf{Dom\_states}\ r \Rightarrow P(\mathsf{Abs\_states}\ r)$

This converts the problem of finding a counter-example to the abstract term to the problem of finding a counter-example to the following Boolean term

```
Dom_states (v0, v1, v2, (v30, v31), v4, v5, v6, (v70, v71))
 ⇒
  Rep_Valid (v0, v1, v2, (v30, v31), v4, v5, v6, (v70, v71))
   ⇒ Reach
        Rep_Trans
        Rep_Initial
        (v0, v1, v2, (v30, v31), v4, v5, v6, (v70, v71))
```

The counter-example found by `findModel` is

$\neg v_{71} \wedge \neg v_{70} \wedge \neg v_{31} \wedge v_{30} \wedge \neg v_2 \wedge \neg v_0$

which abstracts via Abs_states to

```
(Altitude s = away) ∧
(AltEng s = mode_engaged) ∧
(FpaSel s = off) ∧
(AttCws s = off)
```

This is not comprehensible without the context of all the definitions making up the autopilot, but it perhaps gives an idea.

Similar methods yield an explicit formula for Reach Trans Initial s, namely (again details incomprehensible)

```
⊢ Reach Trans Initial s =
    if (AttCws s = engaged)
     then (FpaSel s = off) ∧ (AltEng s = mode_off)
     else
      if (FpaSel s = engaged)
       then (AltEng s = mode_off) ∨
              ((AltEng s = armed) ∧
               (AltDisp s = pre_selected) ∧
               (Altitude s = away))
       else (AltEng s = mode_engaged) ∧
              ((Altitude s = near_pre_selected) ∨
               (Altitude s = at_pre_selected))
```

The autopilot example is so simple that there is no significant gain in using

BDDs for invariant verification.[11] The added value of BDDs are the new kinds of analysis that become possible: automatic generation of counter-examples and explicit terms for reachability fixed-points.
Specialised systems such as SMV [18] have specification languages that provide hardwired translations from user-oriented datatypes to Boolean formulae. With HOL+BuDDy, the user can define bespoke types in higher order logic and then program up problem specific data encodings and use LCF-style high-assurance deduction methods to transform problems into the domain of BDD algorithms.

6 Conclusions

Our goal has been to extend the scope of LCF-style theorem proving to include the ability to program derived rules and tactics that make use of external algorithms. Here we have concentrated on BDD-based symbolic state exploration. The results seem promising, but more case studies are needed. Because the main verification calculations are done in an external BDD engine (BuDDy) the efficiency is good. The relatively slow HOL code (compared with C) only controls the invocation of BuDDy operations and so is out of the critical performance loops.
HOL+BuDDy provides a secure platform to experiment with intimate mixtures of deduction and BDD-based symbolic calculation. It could be especially appropriate for experimenting with intricate optimisations, since soundness is ensured.
Whilst we do not claim that user-programmed tight-integration is always the best way to connect external tools, we think that the work described here provides evidence of its potential benefits for some applications.

Acknowledgements

The implementation of the interface between Moscow ML and BuDDy was done by Ken Larsen with the support of EPSRC grant GR/K57343 entitled *Checking equivalence between synthesized logic and non-synthesizable behavioural prototypes*. Ken Larsen also implemented a prototype HOL or-

[11]The proofs take seconds with a theorem prover against fractions of a second with BDDs. The fixed-point is reached after 8 iterations. The BDDs at successive stages have sizes (number of nodes): 11, 34, 65, 82, 66, 56, 45, 20. The BDD of Trans has size 21,720.

acle that was the starting point for the work reported here. Special thanks go to Jørn Lind-Nielsen for making his BuDDy code freely available to us. This work was also supported by ESPRIT Framework IV LTR 26241 project entitled *Proof and Specification Assisted Design Environments.*
The author would like to thank Anuj Dawar, Tom Melham, Michael Norrish, Konrad Slind and Mark Staples for discussions and help during this research. Mark Aagaard, Mark Greenstreet, Joe Hurd, Paul Jackson, Ken Larsen, John Matthews and Larry Paulson read drafts of this paper, pointed out errors and made many useful suggestions (some of which will be implemented in the next version).

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)*, July 1999.

[3] Saul Amarel. On representation of problems of reasoning about action. In Donald Michie, editor, *Machine Intelligence 3*, pages 131–171. Edinburgh University Press, 1971.

[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] David Basin and Stefan Friedrich. Combining WS1S and HOL. In *Frontiers of Combining Systems, Second International Workshop, Amsterdam, September 1998*, Applied Logic Series. Kluwer Academic Publishers, 1998. To appear.

[6] R. J. Boulton. *Efficiency in a Fully-Expansive Theorem Prover.* PhD thesis, University of Cambridge Computer Laboratory, New Museums

Site, Pembroke Street, Cambridge CB2 3QG, U.K., May 1994. Technical Report 337.

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

[8] Ricky W. Butler. An introduction to requirements capture using PVS: Specification of a simple autopilot. NASA Technical Memorandum 110255, NASA Langley Research Center, Hampton, VA, May 1996.

[9] Ricky W. Butler. An introduction to requirements capture using PVS: Specification of a simple autopilot. Technical Report TR 110255, NASA, May 1996.

[10] 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*, LNCS 407, pages 365–373. Springer-Verlag, 1989.

[11] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.

[12] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[13] Mike Gordon and Ken Larsen. Combining the Hol98 proof assistant with the BuDDy BDD package. Technical report, University of Cambridge Computer Laboratory, 1999. To appear.

[14] 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.

[15] 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.

[16] 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.

[17] Wing Sang Lee. ST to FL translation for hardware design verification. Master's thesis, The University of British Columbia, April 1994.

[18] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[19] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[20] 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/`.

[21] Larry Paulson, private communication.

[22] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer, 1994. LNCS 828.

[23] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[24] 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.

[25] William D. Young. The specification of a simple autopilot in ACL2. Technical Report CLI Internal Note #327, Computational Logic Inc, July 1996.