

PROGRAMMING COMBINATIONS OF DEDUCTION AND BDD-BASED SYMBOLIC CALCULATION

MICHAEL J. C. GORDON

Abstract

A generalisation of Milner’s ‘LCF approach’ is described. This allows algorithms based on binary decision diagrams (BDDs) to be programmed as derived proof rules in a calculus of representation judgements. The derivation of representation judgements becomes an LCF-style proof by defining an abstract type for judgements analogous to the LCF type of theorems. The primitive inference rules for representation judgements correspond to the operations provided by an efficient BDD package coded in C (BuDDy). Proof can combine traditional inference with steps inferring representation judgements. The resulting system provides a platform to support a tight and principled integration of theorem proving and model checking. The methods are illustrated by using them to solve all instances of a generalised Missionaries and Cannibals problem.

1. *Introduction*

LCF-style theorem provers [8] use the ML programming language [17] to define a type *thm* representing theorems of a deductive system. Milner’s key idea [8] 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 values of type *thm* can only be created via sequences of primitive inferences. LCF-style theorem provers are sometimes called ‘fully expansive’, as they expand out all deductions into sequences of primitive inference steps. The notation ‘ $\vdash t$ ’ means that term *t* is a theorem.

Many successful automatic verification algorithms are based on computing with boolean terms represented as *reduced ordered binary decision diagrams* (ROBDDs, or just BDDs for short) [3]. The LCF-approach is extended here to include proving judgements $a \rho t \mapsto b$ as well as conventional theorems $\vdash t$. The components of a judgement $a \rho t \mapsto b$ are a set of boolean terms *a* that are assumed true, a finite map ρ from logical variables to BDD variables, a boolean term *t* (all of whose free variables are boolean) and a BDD *b*. Such a judgement is true if and only if, under the assumptions *a*, the BDD representing *t* with respect to ρ is *b*.

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 $\vdash t$.

In this paper, symbolic calculation is represented as LCF-style proofs of BDD representation judgements. It is still too early to say whether adequate efficiency can be obtained

(though preliminary experiments suggest optimism), but the methodology encourages interesting synergies between theorem proving and model checking, and allows users to program bespoke checkers with an LCF-like assurance of soundness.

The rest of this paper is structured as follows: first the combination of the HOL theorem prover and the BuDDy BDD package using representation judgements $a \rho t \mapsto b$ is outlined, then some basic programming techniques using judgements are described. Next, some elementary reachability concepts are reviewed, and their computation via judgements is explained. Finally, a case study is presented: an analysis of a generalisation of the classical Missionaries and Cannibals problem.

2. BDD representation judgements

A BDD representation judgement is a 4-tuple (a, ρ, t, b) , where a is a finite set of boolean terms called the assumptions, ρ is a finite map from boolean variables (a subset of HOL terms) to natural numbers (representing BDD variables), t is a boolean term and b is a BDD. Such a judgement is true if and only if, under the assumption that all the terms in a are true, the term t is represented by the BDD b , assuming that each variable v in t corresponds to the BDD variable $\rho(v)$. The notation $a \rho t \mapsto b$ indicates that (a, ρ, t, b) follows from the rules given in Sections 2.1, 2.2, 2.3 and 2.4 below. The map ρ is called a variable map. A common case occurs when the set a of assumptions is the empty set $\{\}$, and in this case the notation $\rho t \mapsto b$ is used, rather than $\{\} \rho t \mapsto b$.

The definition of truth just given is informal. It is intended that the rules for deducing representation judgements should be sufficiently simple to be ‘obviously sound’; nevertheless, it is hoped eventually to formalise the truth conditions for judgements and prove their soundness, but this is work for the future.

BDDs in HOL are provided by the BuDDy package due to Jørn Lind-Nielsen, which is implemented in C. The HOL interface to BuDDy is via a Moscow ML API called MuDDy, due to Ken Friis Larsen and Jakob Lichtenberg. Both BuDDy and MuDDy are supported from the IT University, Copenhagen. MuDDy defines an ML type *bdd* of pointers into BuDDy’s BDD space. It also defines ML functions corresponding to operations in the BuDDy API. Some of these functions initialise and configure the BDD package, and others are operations for combining BDDs. For example, MuDDy binds the ML variables TRUE and FALSE (both of ML type *bdd*) to the BDDs representing truth and falsity, respectively. The ML function `ithvar : int → bdd` maps an ML integer i to a BDD node labelled with i (in BuDDy, BDD variables are non-negative integers, and the arithmetical order is the variable order). The functions `NOT : bdd → bdd` and `AND : bdd × bdd → bdd` compute the negation and conjunction of BDDs. Only a small subset of MuDDy is described here, and some of the details are simplified to make the exposition cleaner. Details of the implemented system are available online [7].

The rules for representation judgements can be classified into four categories: rules for constants and variables, rules for combining or manipulating judgements, rules linking judgements and theorems, and rules for changing variable maps.

2.1. Rules for constants and variables

The HOL logical constants T and F are terms denoting truth and falsity, respectively, and have ML type *term*. The ML values TRUE and FALSE of ML type *bdd* are the corresponding BDDs.

In the notation for rules used below, hypotheses and side conditions are written above a horizontal line, with the conclusion below the line. For example, the rule `BddForall` has one judgement as a hypothesis, and a side-condition for each quantified variable.

The inference rules `BddT` and `BddF` link the logical and BDD truth-values. They have no hypotheses or side-conditions.

$$\text{BddT} \frac{}{\rho \text{ T} \mapsto \text{TRUE}} \quad \text{BddF} \frac{}{\rho \text{ F} \mapsto \text{FALSE}}$$

Logic variables in HOL's higher-order logic are a subset of terms, but BDD variables in `BuDDy` are represented by natural numbers (ML integers in `MuDDy`). Logic variables are associated with BDD variables by a partial function from terms to integers called a *variable map* (or 'varmap', for short). A varmap ρ defines a variable ordering: $x < y$ if $\rho(x) < \rho(y)$.

The inference rule `BddVar` for inferring judgements $\rho v \mapsto b$, where v is a variable, has the side-condition that v is bound in the supplied variable map:

$$\text{BddVar} \frac{\rho(v) = n}{\rho v \mapsto \text{ithvar } n}.$$

Evaluating `BddVar` ρv in ML will return $\rho v \mapsto \text{ithvar } n$ if v is bound to n by ρ . An ML exception is raised if v is not in the domain of ρ .

The sets of assumptions generated by `BddT`, `BddF` and `BddVar` are empty.

2.2. Rules for combining or manipulating BDD representation judgements

Here are some example inference rules for proving judgements $a \rho t \mapsto b$, immediately preceded by their ML names (the unary operation `NOT` negates a BDD, and the binary operators `OR`, `IMP` and `BIIMP` construct the disjunction, implication and equivalence of BDDs).

$$\begin{aligned} \text{BddNot} & \frac{a \rho t \mapsto b}{a \rho \neg t \mapsto \text{NOT } b} \\ \text{BddAnd} & \frac{a_1 \rho t_1 \mapsto b_1 \quad a_2 \rho t_2 \mapsto b_2}{a_1 \cup a_2 \rho t_1 \wedge t_2 \mapsto b_1 \text{ AND } b_2} \\ \text{BddOr} & \frac{a_1 \rho t_1 \mapsto b_1 \quad a_2 \rho t_2 \mapsto b_2}{a_1 \cup a_2 \rho t_1 \vee t_2 \mapsto b_1 \text{ OR } b_2} \\ \text{BddImp} & \frac{a_1 \rho t_1 \mapsto b_1 \quad a_2 \rho t_2 \mapsto b_2}{a_1 \cup a_2 \rho t_1 \Rightarrow t_2 \mapsto b_1 \text{ IMP } b_2} \\ \text{BddEq} & \frac{a_1 \rho t_1 \mapsto b_1 \quad a_2 \rho t_2 \mapsto b_2}{a_1 \cup a_2 \rho t_1 = t_2 \mapsto b_1 \text{ BIIMP } b_2} \end{aligned}$$

The functions `Forall` and `Exists` of type $(\text{int list}) \rightarrow \text{bdd} \rightarrow \text{bdd}$ quantify BDDs; thus

$$\begin{aligned} \text{BddForall} & \frac{a \rho t \mapsto b \quad \rho(v_1) = n_1 \quad \dots \quad \rho(v_p) = n_p}{a \rho \forall v_1 \dots v_p. t \mapsto \text{Forall}[n_1, \dots, n_p] b}, \\ \text{BddExists} & \frac{a \rho t \mapsto b \quad \rho(v_1) = n_1 \quad \dots \quad \rho(v_p) = n_p}{a \rho \exists v_1 \dots v_p. t \mapsto \text{Exists}[n_1, \dots, n_p] b}. \end{aligned}$$

The BDDs of quantifications of conjunctions can be built by calling `AND` followed by `forall` or `exists`, but it is more efficient to use the optimised algorithms `forallAnd` and `existsAnd` provided by `BuDDy`.

`BddforallAnd`

$$\frac{a_1 \rho t_1 \mapsto b_1 \quad a_2 \rho t_2 \mapsto b_2 \quad \rho(v_1) = n_1 \quad \dots \quad \rho(v_p) = n_p}{a_1 \cup a_2 \rho \forall v_1 \dots v_p. t_1 \wedge t_2 \mapsto \text{forallAnd } [n_1, \dots, n_p] b_1 b_2}$$

`BddexistsAnd`

$$\frac{a_1 \rho t_1 \mapsto b_1 \quad a_2 \rho t_2 \mapsto b_2 \quad \rho(v_1) = n_1 \quad \dots \quad \rho(v_p) = n_p}{a_1 \cup a_2 \rho \exists v_1 \dots v_p. t_1 \wedge t_2 \mapsto \text{existsAnd } [n_1, \dots, n_p] b_1 b_2}$$

`BuDDy/MuDDy` provides other optimised combinations of quantification and boolean operations, which provide the basis for other representation judgement rules.

The rule `BddSubst` links substitution on logical formulae with substitution on BDDs. Let $t\{v_1 \leftarrow t_1, \dots, v_p \leftarrow t_p\}$ denote the result of simultaneously replacing occurrences of free variables v_1, \dots, v_p in a term t with terms t_1, \dots, t_p , respectively, renaming any bound variables in t to avoid capture. Let $b\{n_1 \leftarrow b_1, \dots, n_p \leftarrow b_p\}$ denote the result of simultaneously replacing BDD variables n_1, \dots, n_p in a BDD b with BDDs b_1, \dots, b_p , respectively (such replacements are supported by `MuDDy`). The rule `BddSubst` takes a list of pairs representing a substitution and a judgement $a \rho t \mapsto b$, and returns the result of performing a term substitution on t and the corresponding BDD substitution on b . The rule as an ML function has ML type $(\text{term_bdd} \times \text{term_bdd})\text{list} \rightarrow \text{term_bdd} \rightarrow \text{term_bdd}$.

`BddSubst`

$$\frac{[(\rho v_1 \mapsto b_{v_1}, \rho t_1 \mapsto b_{t_1}), \dots, (\rho v_p \mapsto b_{v_p}, \rho t_p \mapsto b_{t_p})] \quad a \rho t \mapsto b}{a \rho t\{v_1 \leftarrow t_1, \dots, v_p \leftarrow t_p\} \mapsto b\{\rho(v_1) \leftarrow b_{v_1}, \dots, \rho(v_p) \leftarrow b_{v_p}\}}$$

Two rules that illustrates the use of the a component of a judgement are `BddSimplify` and `BddFindModel`. The first of these provides access to a classic algorithm of Coudert, Berthet and Madre [5], provided by `MuDDy` as a function `simplify : bdd → bdd → bdd` that simplifies its second argument under the assumption that the first argument is true; that is, `simplify b1 b2` results in a BDD, say b'_2 , such that $b_1 \Rightarrow (b_2 = b'_2)$ and in the intended applications b'_2 is simpler than b_2 .

$$\text{BddSimplify} \frac{a_1 \rho t_1 \mapsto b_1 \quad a_2 \rho t_2 \mapsto b_2}{a_1 \cup a_2 \cup \{t_1\} \rho t_2 \mapsto \text{simplify } b_1 b_2}$$

The rule `BddFindModel` uses a BDD algorithm to find a set $\{v_1 = c_1, \dots, v_p = c_p\}$ of assumptions that makes a term t true (where c_i is a boolean constant `T` or `F` for $1 \leq i \leq p$).

$$\text{BddFindModel} \frac{a \rho t \mapsto b}{a \cup \{v_1 = c_1, \dots, v_p = c_p\} \rho t \mapsto \text{TRUE}}$$

An exception is raised if t is unsatisfiable.

2.3. Rules linking BDD representation judgements to HOL theorems

The rule `BddEqMp` expresses the fact that logically equivalent terms have the same BDD.

$$\text{BddEqMp} \frac{a_1 \vdash t_1 = t_2 \quad a_2 \rho t_1 \mapsto b}{a_1 \cup a_2 \rho t_2 \mapsto b}$$

The rule `TermBddOracle` checks whether the BDD part of a judgement is `TRUE`, and if so creates a theorem whose conclusion is the term part.

$$\text{TermBddOracle} \frac{a \rho t \mapsto \text{TRUE}}{a \vdash t}$$

If `TermBddOracle` is applied to a judgement whose BDD part is not `TRUE`, then an ML exception is raised.

The rule `TermBddToEqThm` below converts the BDD part b of a judgement $a \rho t \mapsto b$ to a conditional term $t_{\rho b}$ built out of logic variables corresponding to the BDD variables in b according to ρ , and with the BDD structure represented by a nested conditional.

$$\text{TermBddToEqThm} \frac{a \rho t \mapsto b}{a \vdash t = t_{\rho b}}$$

For example, `TermBddToEqThm` applied to $\rho x \wedge y \wedge \neg z \mapsto b$ yields:

$$\vdash x \wedge y \vee \neg z = \text{if } x \text{ then (if } y \text{ then T else (if } z \text{ then F else T))} \\ \text{else (if } z \text{ then F else T).}$$

Clearly, the right-hand sides of the equations generated by `TermBddToEqThm` may be huge, and the sharing in BDDs is lost. Thus this rule must be used with care.

The rules `TermBddOracle` and `TermBddToEqThm` are the only way that theorems of higher-order logic can be created via BDDs. Theorems ‘proved’ with these are tagged so that users can see that BDD calculations have been used. The route from BDD representation judgements to HOL theorems could be limited to just `TermBddOracle` by having `TermBddToEqThm` use `TermBddOracle` to validate its result.

2.4. Rules for extending or contracting variable maps

The rules given above all have the same variable map ρ in the hypotheses and conclusion. To use one of the preceding rules to combine judgements with different varmaps, one needs first to extend or contract the different varmaps so that they are the same. The following rules enable varmaps to be extended and contracted.

Let ‘extends $\rho_1 \rho_2$ ’ mean that ρ_2 agrees with ρ_1 on all the variables that ρ_1 is defined on, and let ‘remove $v \rho$ ’ denote the result of removing the variable v from the domain of ρ .

$$\text{BddExtendVarmap} \frac{\text{extends } \rho_1 \rho_2 \quad a \rho_1 t \mapsto b}{a \rho_2 t \mapsto b}$$

$$\text{BddFreevarsContractVarmap} \frac{a \rho t \mapsto b \quad v \text{ is not free in } t}{a (\text{remove } v \rho) t \mapsto b}$$

$$\text{BddSupportContractVarmap} \frac{a \rho t \mapsto b \quad \rho(v) \text{ does not occur in } b}{a (\text{remove } v \rho) t \mapsto b}$$

To apply a rule like `BddAnd` to two judgements $a_1 \rho_1 t_1 \mapsto b_1$ and $a_2 \rho_2 t_2 \mapsto b_2$, one uses `BddFreevarsContractVarmap` and `BddSupportContractVarmap` to remove redundant bindings from ρ_1 and ρ_2 to get judgements, say $a_1 \rho'_1 t_1 \mapsto b_1$ and $a_2 \rho'_2 t_2 \mapsto b_2$. One then finds a varmap ρ such that extends $\rho'_1 \rho$ and extends $\rho_2 \rho$, and uses `BddExtendVarmap` to derive $a_1 \rho t_1 \mapsto b_1$ and $a_2 \rho t_2 \mapsto b_2$, which have the same varmap, ρ , and so can be combined using `BddAnd`. A generalised rule that automatically performs the contractions and extensions is easily programmed; a common case, however, is to have a fixed varmap, and the basic rules are optimised for this. If ρ_1 and ρ_2 are incompatible on a variable, v say, occurring in t_1 or t_2 (that is, $\rho_1(v) \neq \rho_2(v)$), then the desired ρ cannot be found.

It is intended to provide support for having incompatible variable orders, though the management of such ‘local scopes’ is likely to be tricky, and the details have not been worked out in relation to representation judgements. Additional rules may be needed.

3. *Derived BDD representation rules*

The rules in the preceding section (and some others not presented here) form a fixed set defining the primitive operations on the abstract type `term_bdd`. The primitives correspond to the BuDDy operations provided by the MuDDy API, together with rules for manipulating varmaps and rules linking representation judgements to logical formulae. The claim that this system provides LCF-like assurance assumes that the fixed set of primitive rules is correctly implemented (which requires BuDDy and MuDDy be correct). Starting from these rules, derived rules can then be programmed in ML in the LCF style.

Suppose that t is a quantified boolean formula (QBF) built out of boolean variables and constants using $\neg, \wedge, \vee, \Rightarrow, =, \forall$ and \exists . One can program an ML function `termToTermBdd` with ML type `term \rightarrow term_bdd` that, when applied to t , computes a ρ and judgement $\rho t \mapsto b$ (with empty assumptions). The variable map is computed by using term-processing functions to get the set of variables in t , then repeatedly using `ithvar` to construct a BDD variable for each variable in t , and finally packaging up the correspondence between logic variables and BDD variables into a variable map ρ . A recursive descent algorithm can then use the primitive rules in Sections 2.1 and 2.2 to deduce a BDD representation judgement for t , and thus implement the following rule.

$$\text{termToTermBdd} \frac{t \text{ is a quantified boolean formula}}{\rho t \mapsto b}$$

A more general version of `termToTermBdd` can be easily implemented; this enables the variable order to be explicitly specified.

3.1. *Computing the set of reachable states*

In higher-order logic, sets are conveniently represented by their characteristic predicates. Suppose that B is a predicate representing a set of initial states; then a state s is in the set represented by B if and only if $B s$ (the application of B to s is true). Suppose that R is a predicate on pairs of states representing the transition relation of a system, so that $R(s, s')$ is true if and only if the system can make a transition from state s to state s' .

A *transition system* is defined by a pair (R, B) consisting of a transition relation and a set of initial states. A state s is *reachable in m steps* in such a transition system if and only if there exists a sequence of states s_0, s_1, \dots, s_m such that $B s_0$ is true (that is, s_0 is an initial

state), $s_m = s$ and $R(s_i, s_{i+1})$ is true for $i = 0, \dots, m-1$. Such a sequence is a *trace* from B to s . The predicate $\text{ReachBy } R \ B \ i$ representing the set of states reachable in i or fewer steps is defined recursively by:

$$\begin{aligned} &\vdash (\text{ReachBy } R \ B \ 0 \ s = B \ s) \\ &\quad \wedge \\ &\quad \forall i. \text{ReachBy } R \ B \ (i+1) \ s = \text{ReachBy } R \ B \ i \ s \vee \exists u. \text{ReachBy } R \ B \ i \ u \wedge R(u, s). \end{aligned}$$

A state s is *reachable* if and only if it is reachable in a finite number of steps; that is, if and only if $\exists i. \text{ReachBy } R \ B \ i \ s$. The predicate Reachable characterising the set of reachable states is thus defined by:

$$\vdash \text{Reachable } R \ B \ s = \exists i. \text{ReachBy } R \ B \ i \ s.$$

The following fixed-point theorem is straightforward to prove:

$$\begin{aligned} &\vdash \forall i. (\forall s. \text{ReachBy } R \ B \ i \ s = \text{ReachBy } R \ B \ (i+1) \ s) \\ &\quad \Rightarrow \\ &\quad \forall s. \text{Reachable } R \ B \ s = \text{ReachBy } R \ B \ i \ s. \end{aligned}$$

Thus, to compute the set of reachable states, it is sufficient to find a number of iterations i such that $\forall s. \text{ReachBy } R \ B \ i \ s = \text{ReachBy } R \ B \ (i+1) \ s$.

To use BDDs, states are encoded as vectors of booleans (see Section 3.4). Suppose that the constants B and R have been defined by theorems:

$$\begin{aligned} &\vdash B(v_1, \dots, v_n) = \text{QBF}_1, \\ &\vdash R((v_1, \dots, v_n), (v'_1, \dots, v'_n)) = \text{QBF}_2, \end{aligned}$$

where QBF_1 and QBF_2 are quantified boolean formulae that can be directly represented as BDDs. With `termToTermBdd` followed by `BddEqMp`, one can create representation judgements:

$$\begin{aligned} &\rho(B(v_1, \dots, v_n)) \mapsto b_B \\ &\rho(R((v_1, \dots, v_n), (v'_1, \dots, v'_n))) \mapsto b_R \end{aligned}$$

A simple iteration [6] then implements a derived rule that can take these as hypotheses:

`computeReachable`

$$\frac{\rho(B(v_1, \dots, v_n)) \mapsto b_B \quad \rho(R((v_1, \dots, v_n), (v'_1, \dots, v'_n))) \mapsto b_R}{\rho(\text{Reachable } R \ B \ (v_1, \dots, v_n)) \mapsto b_{\text{ReachRB}}}$$

The iteration consists in computing

$$\begin{aligned} &\rho(\text{ReachBy } R \ B \ 0 \ (v_1, \dots, v_n)) \mapsto b_0 \\ &\rho(\text{ReachBy } R \ B \ 1 \ (v_1, \dots, v_n)) \mapsto b_1 \\ &\rho(\text{ReachBy } R \ B \ 2 \ (v_1, \dots, v_n)) \mapsto b_2 \\ &\quad \vdots \\ &\rho(\text{ReachBy } R \ B \ i \ (v_1, \dots, v_n)) \mapsto b_i \\ &\rho(\text{ReachBy } R \ B \ (i+1) \ (v_1, \dots, v_n)) \mapsto b_{i+1} \end{aligned}$$

until

$$\vdash \text{ReachBy } R \ B \ i \ (v_1, \dots, v_n) = \text{ReachBy } R \ B \ (i+1) \ (v_1, \dots, v_n).$$

This is tested for by using `BddEq` followed by `TermBddOracle`. When such an i is reached, the conclusion of `computeReachable` is deduced by applying `BddEqMp` and the preceding fixed-point theorem.

Applying `TermBddToEqThm` to the result of `computeReachable` results in a theorem:

$$\vdash \text{Reachable RB } (v_1, \dots, v_n) = t_{\text{ReachRB}},$$

which gives an explicit characterisation of the set of reachable states.

A simple ML program can be used to compose together the inferences just described, to implement a derived rule:

`computeReachableThm`

$$\frac{\vdash \text{B}(v_1, \dots, v_n) = \text{QBF}_1 \quad \vdash \text{R}((v_1, \dots, v_n), (v'_1, \dots, v'_n)) = \text{QBF}_2}{\vdash \text{Reachable RB } (v_1, \dots, v_n) = t_{\text{ReachRB}}}$$

This is a derived rule that infers a HOL conclusion from HOL hypotheses using internal BDD calculations. For an application, see Section 4.

3.2. Computing traces

Suppose that one wants to check that all reachable states have property P ; then one can compute the BDD of $\text{Reachable RB } (v_1, \dots, v_n) \Rightarrow P(v_1, \dots, v_n)$ and see whether it is true (using `TermBddOracle`). If it turns out that P does not hold for all reachable states, then it is useful to find a trace to a counterexample. Such a trace can be computed in three steps as follows.

1. Find the first i such that $\text{ReachBy RB } i (v_1, \dots, v_n) \wedge \neg P(v_1, \dots, v_n)$ is satisfiable.
2. Find a particular state, s_i say, such that $\text{ReachBy RB } i s_i \wedge \neg(P s_i)$.
3. Work backwards from s_i to get a trace s_0, \dots, s_i .

In more detail, this involves the following actions.

1. Iteratively compute the BDD, say b_i , of $\text{ReachBy RB } i (v_1, \dots, v_n)$ for $i = 0, 1, 2, \dots$, and on each iteration check if the conjunction of b_i and the BDD of $\neg P(v_1, \dots, v_n)$ is not `FALSE` (using the rules `BddNot` and `BddAnd`). Stop at the first i for which the check succeeds.
2. Compute the BDD of $\text{ReachBy RB } i (v_1, \dots, v_n) \wedge \neg(P(v_1, \dots, v_n))$ (where i is the fixed point found in 1), and then use `BddFindModel` to deduce a judgement:

$$\{v_1 = c_1, \dots, v_n = c_n\} \rho (\text{ReachBy RB } i (v_1, \dots, v_n) \wedge \neg(P(v_1, \dots, v_n))) \mapsto \text{TRUE}.$$

From this it follows that if $s_i = (c_1, \dots, c_n)$, then $\text{ReachBy RB } i s_i \wedge \neg(P s_i)$.

3. Generate a sequence of vectors representing states starting from an initial state and ending in s_i by tracing backwards, as follows. First, define two auxiliary HOL constants `Prev` and `Eq` by:

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

and then iteratively construct a sequence s_i, \dots, s_0 , where, if a vector of boolean constants s_p has been constructed, then `BddFindModel` is used to obtain s_{p-1} satisfying $\text{ReachBy } R \ B \ (p-1) \ s_{p-1} \ \wedge \ \text{Prev } R \ (\text{Eq } s_p) \ s_{p-1}$. Note that the BDD b_p of $\text{ReachBy } R \ B \ p \ (v_1, \dots, v_n)$ (for $p = 0, 1, \dots, i$) was computed during step 1, so is available without further work. The BDD of $\text{Prev } R \ (\text{Eq } s_p) \ (v_1, \dots, v_n)$ is computed from the BDDs of $R((v_1, \dots, v_n), (v'_1, \dots, v'_n))$ and $\text{Eq}(s_p, (v_1, \dots, v_n))$.

The sequence s_0, \dots, s_i is a trace from an initial state to a counter-example state. The following properties can be deduced:

$$\begin{aligned} &\vdash B \ s_0 \\ &\vdash \text{ReachBy } R \ B \ p \ s_p \ \wedge \ R(s_p, s_{p+1}) \quad (0 \leq p < i) \\ &\vdash \text{ReachBy } R \ B \ n \ s_i \ \wedge \ \neg(P \ s_i) \end{aligned}$$

For debugging, it seems an overkill to prove theorems verifying 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 synthesise 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 in Section 4 is suggestive: a schedule of boat trips is a kind of program.

3.3. *Disjunctive partitioning by proof*

Disjunctive partitioning is a well-known method of transforming the formulae for the next state (forward image) or previous state (backward image, that is, $\text{Prev } R \ P \ s$) so that it is not necessary to compute the BDD of the whole transition relation $R((v_1, \dots, v_n), (v'_1, \dots, v'_n))$ [4, p. 80]. It is suitable for large asynchronous circuits. The technique is also called *early quantification* [14, p. 45].

Consider, for example, the state transition relation \mathcal{R} defined by:

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

where $E_x(x, y, z)$, $E_y(x, y, z)$ and $E_z(x, y, z)$ are boolean formulae. Such a transition relation represents the asynchronous composition of E_x , E_y and E_z .

Logical simplification gives:

$$\begin{aligned} \exists x' \ y' \ z'. \ \text{ReachBy } \mathcal{R} \ \mathcal{B} \ n \ (x', y', z') \ \wedge \ \mathcal{R}((x', y', z'), (x, y, z)) \\ &= (\exists x'. \ \text{ReachBy } \mathcal{R} \ \mathcal{B} \ n \ (x', y, z) \ \wedge \ x = E_x(x', y, z)) \\ &\quad \vee (\exists y'. \ \text{ReachBy } \mathcal{R} \ \mathcal{B} \ n \ (x, y', z) \ \wedge \ y = E_y(x, y', z)) \\ &\quad \vee (\exists z'. \ \text{ReachBy } \mathcal{R} \ \mathcal{B} \ n \ (x, y, z') \ \wedge \ z = E_z(x, y, z')). \end{aligned}$$

The BDD of $\exists x' \ y' \ z'. \ \text{ReachBy } \mathcal{R} \ \mathcal{B} \ n \ (x', y', z') \ \wedge \ \mathcal{R}((x', y', z'), (x, y, z))$ can thus be computed without ever computing the BDD of $\mathcal{R}((x', y', z'), (x, y, z))$. This simplification is done automatically by the HOL simplifier.

The usual implementation of disjunctive partitioning is achieved by writing programs that directly construct the BDD of the simplified term. The logical transformations are thus encoded in BDD building code. The approach here is to deductively simplify the next-state relation prior to creating the BDD. The advantages are that the implementation is very straightforward, and the simplification is guaranteed sound.

3.4. *Encoding values as boolean vectors*

BDD representation judgements $a \rho t \mapsto b$ only represent terms t whose free variables are boolean. To use BDDs to calculate with terms containing free variables of other types, one must encode these types in terms of booleans. Such an encoding of a type σ inside higher-order logic consists of a predicate $D : \sigma \rightarrow \text{bool}$ (where bool is the higher-order logic type of booleans) defining the subset of σ that will be encoded, a function $\text{rep} : \sigma \rightarrow \tau$ that maps elements of type σ to a type τ consisting of vectors of booleans (so τ has the form $\text{bool} \times \dots \times \text{bool}$) and a function $\text{abs} : \tau \rightarrow \sigma$ that abstracts vectors of booleans (that is, elements of the type τ) to values of type σ . An example, which will be used in Section 4, is the encoding of a subset of numbers, which are values of the higher-order logic type num , in binary. The subset of numbers that are less than 16 will be encoded.

Let $\text{Rep4} : \text{num} \rightarrow \text{bool} \times \text{bool} \times \text{bool} \times \text{bool}$ map a number to the 4-bit binary representation of its value mod 16, let $\text{Abs4} : \text{bool} \times \text{bool} \times \text{bool} \times \text{bool} \rightarrow \text{num}$ be the number denoted by a 4-bit word, and let $\text{Dom4} : \text{num} \rightarrow \text{bool}$ be defined by $\text{Dom4 } n = (n < 16)$. Note that:

$$\vdash (\forall n. \text{Dom4 } n = (\text{Abs4}(\text{Rep4 } n) = n)) \wedge (\forall w. \text{Rep4}(\text{Abs4 } w) = w).$$

A transition system (R, B) is encoded by $(D, \text{abs}, \text{rep})$ if and only if it is the case that $\text{Encode}(R, B, D, \text{abs}, \text{rep})$, where:

$$\begin{aligned} \vdash \text{Encode}(R, B, D, \text{abs}, \text{rep}) = & (\forall u. D u = (\text{abs}(\text{rep } u) = u)) \\ & \wedge (\forall v. \text{rep}(\text{abs } v) = v) \\ & \wedge (\forall u. B u \Rightarrow D u) \\ & \wedge (\forall u. D u \Rightarrow \forall u'. R(u, u') \Rightarrow D u'). \end{aligned}$$

Here, and in the theorem below, u is of type σ (abstract state) and v is of type τ (vector of booleans). Let the function composition (\circ) and product ($\#$) operators be defined by:

$$\begin{aligned} (f \circ g) x &= f(g x) \\ (f \# g) (x, y) &= (f x, g y). \end{aligned}$$

The following theorem, which we name ReachByAbs , is proved by a simple induction:

$$\begin{aligned} \vdash \text{Encode}(R, B, D, \text{abs}, \text{rep}) \\ \Rightarrow \\ \forall n v. \text{ReachBy } R B n (\text{abs } v) = \text{ReachBy } (R \circ (\text{abs} \# \text{abs})) (B \circ \text{abs}) n v, \end{aligned}$$

hence the following theorem, which we name ReachableAbs :

$$\begin{aligned} \vdash \text{Encode}(R, B, D, \text{abs}, \text{rep}) \\ \Rightarrow \\ \forall v. \text{Reachable } R B (\text{abs } v) = \text{Reachable } (R \circ (\text{abs} \# \text{abs})) (B \circ \text{abs}) v. \end{aligned}$$

To calculate `Reachable (R ◦ (abs # abs)) (B ◦ abs) (v1, . . . , vn)`, one first derives theorems of the form:

$$\begin{aligned} \vdash (R \circ (abs \# abs))((v_1, \dots, v_n), (v'_1, \dots, v'_n)) &= \text{QBF}_1, \\ \vdash (B \circ abs)(v_1, \dots, v_n) &= \text{QBF}_2, \end{aligned}$$

where v_1, \dots, v_n are boolean variables, n depends on the type τ and QBF_1 and QBF_2 are quantified boolean formulae that can be directly represented as BDDs. These theorems can be derived from the definitions of R , B and abs . The rule `computeReachableThm` can then be applied to them.

To see if some property, P say, holds for all reachable states, use proof to derive an equation:

$$\vdash P(abs(v_1, \dots, v_n)) = \text{QBF}_3,$$

and then use `termToTermBdd` to get the BDD of $P(abs(v_1, \dots, v_n))$. Then use `BddImp` followed by `TermBddOracle`, and replace quantification over n boolean variables by a single quantification over a variable of type $bool \times \dots \times bool$ to derive:

$$\vdash \text{Reachable } (R \circ (abs \# abs)) (B \circ abs) (v_1, \dots, v_n) \Rightarrow P(abs(v_1, \dots, v_n))$$

and hence:

$$\vdash \forall v. \text{Reachable } (R \circ (abs \# abs)) (B \circ abs) v \Rightarrow P(abs v).$$

Using `ReachableAbs` gives:

$$\vdash \forall v. \text{Reachable } R B (abs v) \Rightarrow P(abs v).$$

Specialising v to $rep\ u$:

$$\vdash \text{Reachable } R B (abs(rep\ u)) \Rightarrow P(abs(rep\ u)).$$

Now, if $\vdash \text{Encode}(R, B, D, abs, rep)$, then $\forall u. \text{Reachable } R B u \Rightarrow D\ u$ (the proof is easy) and so $abs(rep\ u) = u$. Hence:

$$\vdash \forall u. \text{Reachable } R B u \Rightarrow P\ u.$$

Suppose, as in Section 3.2, that we have calculated a trace v_0, \dots, v_i :

$$\begin{aligned} \vdash (B \circ abs) v_0, \\ \vdash \text{ReachBy } (R \circ (abs \# abs)) (B \circ abs) p\ v_p \wedge (R \circ (abs \# abs))(v_p, v_{p+1}), \\ \vdash \text{ReachBy } (R \circ (abs \# abs)) B\ n\ v_i \wedge \neg((P \circ abs) v_i), \end{aligned}$$

and hence by `ReachByAbs` and the definitions of \circ and $\#$:

$$\begin{aligned} \vdash B(abs\ v_0), \\ \vdash \text{ReachBy } R B p (abs\ v_p) \wedge R(abs\ v_p, abs\ v_{p+1}), \\ \vdash \text{ReachBy } R B n (abs\ v_i) \wedge \neg(P(abs\ v_i)). \end{aligned}$$

Then the sequence $(abs\ v_0), \dots, (abs\ v_i)$ is a trace from an initial state to a reachable state refuting P .

4. Example: Missionaries and Cannibals problem

The Missionaries and Cannibals Problem is a classic of Artificial Intelligence [2]. We show how to combine theorem proving and BDD calculation to solve all cases of a generalised version.

The original Missionaries and Cannibals Problem (MCP) is [13]:

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?

This problem can be generalised to have n missionaries and n cannibals and a boat of capacity k . Call this problem $\text{MCP}(n, k)$. The problem stated above is then $\text{MCP}(3, 2)$. 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 the boat are on the left bank. Let a state (m, c, b) represent 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 $(0, 0, \text{F})$. A predicate B_{MCP} characterising the initial state is thus defined by

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

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:

$$\begin{aligned} \vdash \text{MoveRight } n k (u, v) ((m, c, b), (m', c', b')) \\ = b \wedge u \leq m \wedge v \leq c \wedge \neg(0 < u \wedge u < v) \wedge 0 < u+v \wedge u+v \leq k \\ \wedge m' = m-u \wedge c' = c-v \wedge \neg b'. \end{aligned}$$

Similarly, MoveLeft is defined by:

$$\begin{aligned} \vdash \text{MoveLeft } n k (u, v) ((m, c, b), (m', c', b')) \\ = \neg b \wedge u \leq n-m \wedge v \leq n-c \wedge \neg(0 < u \wedge u < v) \wedge 0 < u+v \wedge u+v \leq k \\ \wedge m' = m+u \wedge c' = c+v \wedge b'. \end{aligned}$$

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

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

Thus the transition relation is

$$\begin{aligned} \vdash \text{R}_{\text{MCP}} n k (s, s') \\ = \neg \text{Eat } n s' \wedge (\exists u v. u \leq n \wedge v \leq n \wedge \text{MoveRight } n k (u, v) (s, s')) \\ \wedge (\exists u v. u \leq n \wedge v \leq n \wedge \text{MoveLeft } n k (u, v) (s, s')). \end{aligned}$$

The generalised missionaries and cannibals problem $\text{MCP}(n, k)$ is solvable if the goal

state $(0, 0, F)$ is reachable from the initial state, thus:

$$\vdash \text{MCP}(n, k) = \text{Reachable} (\text{R}_{\text{MCP}} n k) (\text{B}_{\text{MCP}} n) (0, 0, F).$$

We want to determine the truth of $\text{MCP}(n, k)$ for all n and k , and in the cases where it is true, compute explicit solutions (that is, traces from the initial state to the goal state).

As a first step, we consider the finite subset of the problem when n and k are less than 16, that is, when they can be encoded using four bits.

A suitable encoding is defined by:

$$\begin{aligned} \vdash \text{Abs} &= \text{Abs4}\#\text{Abs4}\#\text{I}, \\ \vdash \text{Rep} &= \text{Rep4}\#\text{Rep4}\#\text{I}, \\ \vdash \text{Dom}(m, c, b) &= m < 16 \wedge c < 16, \end{aligned}$$

where Abs4 and Rep4 are as defined in Section 3.4, and I is the identity function.

It follows easily that:

$$\vdash n < 16 \wedge k < 16 \Rightarrow \text{Encode}(\text{R}_{\text{MCP}} n k, \text{B}_{\text{MCP}} n, \text{Dom}, \text{Abs}, \text{Rep});$$

hence by Modus Ponens and `ReachableAbs`,

$$\begin{aligned} \vdash n < 16 \wedge k < 16 \\ \Rightarrow \\ \forall v. \text{Reachable} (\text{R}_{\text{MCP}} n k) (\text{B}_{\text{MCP}} n) (\text{Abs } v) \\ = \text{Reachable} (\text{R}_{\text{MCP}} n k \circ (\text{Abs} \# \text{Abs})) (\text{B}_{\text{MCP}} n \circ \text{Abs}) v. \end{aligned}$$

Now

$$\begin{aligned} &\text{Reachable} \\ &(\text{R}_{\text{MCP}} (\text{Abs4}(n_3, n_2, n_1, n_0)) (\text{Abs4}(k_3, k_2, k_1, k_0)) \circ (\text{Abs} \# \text{Abs})) \\ &(\text{B}_{\text{MCP}} (\text{Abs4}(n_3, n_2, n_1, n_0)) \circ \text{Abs}) \\ &((F, F, F, F), (F, F, F, F), F) \end{aligned}$$

can be computed using `computeReachableThm`.

The theorems that `computeReachableThm` takes as parameters are derived by symbolically executing the terms:

$$\begin{aligned} &(\text{R}_{\text{MCP}} (\text{Abs4}(n_3, n_2, n_1, n_0)) (\text{Abs4}(k_3, k_2, k_1, k_0)) \circ (\text{Abs} \# \text{Abs})) \\ &(((m_3, m_2, m_1, m_0), (c_3, c_2, c_1, c_0), b), ((m'_3, m'_2, m'_1, m'_0), (c'_3, c'_2, c'_1, c'_0), b')), \\ &(\text{B}_{\text{MCP}} (\text{Abs4}(n_3, n_2, n_1, n_0)) \circ \text{Abs}) ((m_3, m_2, m_1, m_0), (c_3, c_2, c_1, c_0), b). \end{aligned}$$

The process of symbolic execution needs to use the definitions of `MoveRight`, `MoveLeft` and `Eat`. It also needs to use properties that relate arithmetic on words to arithmetic on numbers, that is, properties like:

$$\begin{aligned} \vdash \text{Abs4}(m_3, m_2, m_1, m_0) + \text{Abs4}(n_3, n_2, n_1, n_0) = \\ \text{Abs4}((m_3, m_2, m_1, m_0) \oplus (n_3, n_2, n_1, n_0)), \end{aligned}$$

where \oplus is a suitable operation implementing addition (for example, a binary adder implementation).

Using such properties giving binary implementations of the arithmetical operations $+$, $-$, $<$ and \leq , it is straightforward to program a derived rule that automatically performs the symbolic execution needed to derive the theorems needed by `computeReachableThm` (we omit the details here).

Invoking `computeReachableThm` results in an inscrutable theorem:

⊢ Reachable

$(R_{MCP} (Abs4(n_3, n_2, n_1, n_0)) (Abs4(k_3, k_2, k_1, k_0))) \circ (Abs \# Abs)$

$(B_{MCP} (Abs4(n_3, n_2, n_1, n_0)) \circ Abs)$

$((F, F, F, F), (F, F, F, F), F)$

$$\begin{aligned}
 &= k_0 \wedge k_1 \wedge k_2 \wedge n_0 \\
 &\quad \vee k_0 \wedge k_1 \wedge k_2 \wedge n_1 \\
 &\quad \vee k_0 \wedge k_1 \wedge k_2 \wedge n_2 \\
 &\quad \vee k_0 \wedge k_1 \wedge k_2 \wedge n_3 \\
 &\quad \vee k_0 \wedge k_1 \wedge \neg k_2 \wedge k_3 \wedge n_0 \\
 &\quad \vee k_0 \wedge k_1 \wedge \neg k_2 \wedge k_3 \wedge n_1 \\
 &\quad \vee k_0 \wedge k_1 \wedge \neg k_2 \wedge k_3 \wedge n_2 \\
 &\quad \vee k_0 \wedge k_1 \wedge \neg k_2 \wedge k_3 \wedge n_3 \\
 &\quad \vee k_0 \wedge k_1 \wedge \neg k_2 \wedge \neg k_3 \wedge n_0 \wedge n_1 \wedge \neg n_2 \wedge \neg n_3 \\
 &\quad \vee k_0 \wedge k_1 \wedge \neg k_2 \wedge \neg k_3 \wedge n_0 \wedge \neg n_1 \wedge \neg n_3 \\
 &\quad \vee k_0 \wedge k_1 \wedge \neg k_2 \wedge \neg k_3 \wedge \neg n_0 \wedge n_1 \wedge \neg n_2 \wedge \neg n_3 \\
 &\quad \vee k_0 \wedge k_1 \wedge \neg k_2 \wedge \neg k_3 \wedge \neg n_0 \wedge \neg n_1 \wedge n_2 \wedge \neg n_3 \\
 &\quad \vee k_0 \wedge \neg k_1 \wedge k_2 \wedge n_0 \\
 &\quad \vee k_0 \wedge \neg k_1 \wedge k_2 \wedge n_1 \\
 &\quad \vee k_0 \wedge \neg k_1 \wedge k_2 \wedge n_2 \\
 &\quad \vee k_0 \wedge \neg k_1 \wedge k_2 \wedge n_3 \\
 &\quad \vee k_0 \wedge \neg k_1 \wedge \neg k_2 \wedge k_3 \wedge n_0 \\
 &\quad \vee k_0 \wedge \neg k_1 \wedge \neg k_2 \wedge k_3 \wedge n_1 \\
 &\quad \vee k_0 \wedge \neg k_1 \wedge \neg k_2 \wedge k_3 \wedge n_2 \\
 &\quad \vee k_0 \wedge \neg k_1 \wedge \neg k_2 \wedge k_3 \wedge n_3 \\
 &\quad \vee \neg k_0 \wedge k_1 \wedge k_2 \wedge n_0 \\
 &\quad \vee \neg k_0 \wedge k_1 \wedge k_2 \wedge n_1 \\
 &\quad \vee \neg k_0 \wedge k_1 \wedge k_2 \wedge n_2 \\
 &\quad \vee \neg k_0 \wedge k_1 \wedge k_2 \wedge n_3 \\
 &\quad \vee \neg k_0 \wedge k_1 \wedge \neg k_2 \wedge k_3 \wedge n_0 \\
 &\quad \vee \neg k_0 \wedge k_1 \wedge \neg k_2 \wedge k_3 \wedge n_1 \\
 &\quad \vee \neg k_0 \wedge k_1 \wedge \neg k_2 \wedge k_3 \wedge n_2 \\
 &\quad \vee \neg k_0 \wedge k_1 \wedge \neg k_2 \wedge k_3 \wedge n_3 \\
 &\quad \vee \neg k_0 \wedge k_1 \wedge \neg k_2 \wedge \neg k_3 \wedge n_0 \wedge \neg n_2 \wedge \neg n_3 \\
 &\quad \vee \neg k_0 \wedge k_1 \wedge \neg k_2 \wedge \neg k_3 \wedge \neg n_0 \wedge n_1 \wedge \neg n_2 \wedge \neg n_3 \\
 &\quad \vee \neg k_0 \wedge \neg k_1 \wedge k_2 \wedge n_0 \\
 &\quad \vee \neg k_0 \wedge \neg k_1 \wedge k_2 \wedge n_1 \\
 &\quad \vee \neg k_0 \wedge \neg k_1 \wedge k_2 \wedge n_2 \\
 &\quad \vee \neg k_0 \wedge \neg k_1 \wedge k_2 \wedge n_3 \\
 &\quad \vee \neg k_0 \wedge \neg k_1 \wedge \neg k_2 \wedge k_3 \wedge n_0 \\
 &\quad \vee \neg k_0 \wedge \neg k_1 \wedge \neg k_2 \wedge k_3 \wedge n_1 \\
 &\quad \vee \neg k_0 \wedge \neg k_1 \wedge \neg k_2 \wedge k_3 \wedge n_2 \\
 &\quad \vee \neg k_0 \wedge \neg k_1 \wedge \neg k_2 \wedge k_3 \wedge n_3.
 \end{aligned}$$

From this, using the result derived from `ReachableAbs` above, and some rewriting with equations about sets, we obtain an explicit characterisation of $\text{MCP}(n, k)$ for n and k less than 16:

$$\begin{aligned}
 &\vdash n < 16 \wedge k < 16 \\
 &\Rightarrow \\
 &(\text{Reachable } (\mathbf{R}_{\text{MCP}} n k) (\mathbf{B}_{\text{MCP}} n) (0, 0, \mathbf{F}) \\
 &= (n, k) \in \{(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (1, 10), \\
 &\quad (1, 11), (1, 12), (1, 13), (1, 14), (1, 15), \\
 &\quad (2, 2), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7), (2, 8), (2, 9), (2, 10), \\
 &\quad (2, 11), (2, 12), (2, 13), (2, 14), (2, 15), \\
 &\quad (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9), (3, 10), \\
 &\quad (3, 11), (3, 12), (3, 13), (3, 14), (3, 15), \\
 &\quad (4, 3), (4, 4), (4, 5), (4, 6), (4, 7), (4, 8), (4, 9), (4, 10), \\
 &\quad (4, 11), (4, 12), (4, 13), (4, 14), (4, 15), \\
 &\quad (5, 3), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9), (5, 10), \\
 &\quad (5, 11), (5, 12), (5, 13), (5, 14), (5, 15), \\
 &\quad (6, 4), (6, 5), (6, 6), (6, 7), (6, 8), (6, 9), (6, 10), \\
 &\quad (6, 11), (6, 12), (6, 13), (6, 14), (6, 15), \\
 &\quad (7, 4), (7, 5), (7, 6), (7, 7), (7, 8), (7, 9), (7, 10), \\
 &\quad (7, 11), (7, 12), (7, 13), (7, 14), (7, 15), \\
 &\quad (8, 4), (8, 5), (8, 6), (8, 7), (8, 8), (8, 9), (8, 10), \\
 &\quad (8, 11), (8, 12), (8, 13), (8, 14), (8, 15), \\
 &\quad (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9), (9, 10), \\
 &\quad (9, 11), (9, 12), (9, 13), (9, 14), (9, 15), \\
 &\quad (10, 4), (10, 5), (10, 6), (10, 7), (10, 8), (10, 9), (10, 10), \\
 &\quad (10, 11), (10, 12), (10, 13), (10, 14), (10, 15), \\
 &\quad (11, 4), (11, 5), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), \\
 &\quad (11, 11), (11, 12), (11, 13), (11, 14), (11, 15), \\
 &\quad (12, 4), (12, 5), (12, 6), (12, 7), (12, 8), (12, 9), (12, 10), \\
 &\quad (12, 11), (12, 12), (12, 13), (12, 14), (12, 15), \\
 &\quad (13, 4), (13, 5), (13, 6), (13, 7), (13, 8), (13, 9), (13, 10), \\
 &\quad (13, 11), (13, 12), (13, 13), (13, 14), (13, 15), \\
 &\quad (14, 4), (14, 5), (14, 6), (14, 7), (14, 8), (14, 9), (14, 10), \\
 &\quad (14, 11), (14, 12), (14, 13), (14, 14), (14, 15), (15, 4), \\
 &\quad (15, 5), (15, 6), (15, 7), (15, 8), (15, 9), (15, 10), \\
 &\quad (15, 11), (15, 12), (15, 13), (15, 14), (15, 15)\}.
 \end{aligned}$$

A few things here are immediately obvious: $\text{MCP}(n, k)$ is solvable if $k \geq 2$ and $n = 1$, as the single missionary and the single cannibal can take the boat together to the right bank. If $k \geq 4$, then $\text{MCP}(n, k)$ is solvable, since one missionary and one cannibal can act as boatmen and ferry pairs of missionaries and cannibals from left to right. Thus interactive proof can be used to show that

$$\vdash ((k \geq 2 \wedge (n = 1)) \vee (k \geq 4 \wedge n > 0)) \Rightarrow \text{MCP}(n, k).$$

Also, $MCP(n, 0)$ is not solvable, because there is no one to operate the boat, and if $k < 2$ then $MCP(n, k)$ is also not solvable because one can never get more than one person to the right bank (since the first person to go to the right bank will have to return, as the boat needs at least one occupant). Thus

$$\vdash (n < 1 \vee k < 2) \Rightarrow \neg MCP(n, k).$$

Inspecting the members of the set of pairs, we see that when $k < 4$ there are no solutions if $n \geq 2k$. The following argument that this is the case for all n and k (that is, not just those less than 16) is due to Anuj Dawar, as outlined in a private communication to the author. Suppose that M_L and C_L are the numbers of missionaries and cannibals, respectively, on the left bank, and M_R and C_R are the numbers of missionaries and cannibals, respectively, on the right bank. Consider the last time that 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, and 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 also travel, so that the remaining ones are not outnumbered, and if any cannibals travel, at least that number of missionaries must also travel, so that missionaries on the left bank are not outnumbered. This means that exactly one missionary and one cannibal can travel (since $k < 4$).

This leaves $M_R < k$, and therefore $M_L > n - k$; that is, $M_L > k$ (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 that exactly one missionary and one cannibal travel, but this just reverses the last move.

Thus:

$$\vdash (1 < k \wedge k < 4 \wedge n \geq 2k) \Rightarrow \neg MCP(n, k).$$

These arguments about MCP can be formalised and the theorems proved interactively. The proofs are quite messy as there are many cases, but are routine. The arguments above established the value of $MCP(n, k)$ for the cases:

$$\begin{aligned} &(k \geq 2 \wedge (n = 1)) \vee (k \geq 4 \wedge n > 0) \\ &\quad \vee (n < 1 \vee k < 2) \\ &\quad \vee (k < 4 \wedge n \geq 2k). \end{aligned}$$

The following theorem is easily proved automatically with a decision procedure:

$$\begin{aligned} \vdash &(k \geq 2 \wedge (n = 1)) \vee (k \geq 4 \wedge n > 0) \\ &\quad \vee (n < 1 \vee k < 2) \\ &\quad \vee (k < 4 \wedge n \geq 2k) \\ &\quad \vee ((k = 2) \wedge ((n = 2) \vee (n = 3))) \\ &\quad \vee ((k = 3) \wedge ((n = 2) \vee (n = 3) \vee (n = 4) \vee (n = 5))). \end{aligned}$$

Thus, to determine the truth of $MCP(n, k)$ for all n and k , we need only to solve $MCP(2, 2)$, $MCP(3, 2)$, $MCP(2, 3)$, $MCP(3, 3)$, $MCP(4, 3)$ and $MCP(5, 3)$, but the explicit set of pairs derived above shows that these are all true.

Putting everything together, one can derive a complete characterisation of $\text{MCP}(n, k)$ in the form of a theorem:

$$\vdash \text{MCP}(n, k) = k > 1 \wedge n > 0 \wedge (k < 4 \Rightarrow n < 2k).$$

One can also use the trace-finding method of Section 3.2 to compute explicit solutions to the solvable finite cases, by finding counterexamples, for specific values of n and k , to

$$\text{Reachable}(\text{R}_{\text{MCP } n \ k}) (\text{B}_{\text{MCP } n}) (m, c, b) \Rightarrow \neg((m, c, b) = (0, 0, F))$$

For example, for the case $n = 2$ and $k = 3$ the following theorems representing a solution can be automatically generated:

$$\begin{aligned} &\vdash \text{R}_{\text{MCP } 2 \ 3} ((2, 2, T), (0, 1, F)), \\ &\vdash \text{R}_{\text{MCP } 2 \ 3} ((0, 1, F), (0, 2, T)), \\ &\vdash \text{R}_{\text{MCP } 2 \ 3} ((0, 2, T), (0, 0, F)). \end{aligned}$$

These three theorems can be compactly abbreviated by:

$$\text{MCP}(2, 3) : 22T \rightarrow 01F \rightarrow 02T \rightarrow 00F,$$

where the notation mcb abbreviates (m, c, b) . The other solutions are easily computed:

$$\text{MCP}(2, 2) : 22T \rightarrow 20F \rightarrow 21T \rightarrow 01F \rightarrow 02T \rightarrow 00F,$$

$$\text{MCP}(3, 2) : 33T \rightarrow 22F \rightarrow 32T \rightarrow 30F \rightarrow 31T \rightarrow 11F \rightarrow 22T \rightarrow 02F \rightarrow 03T \rightarrow 01F \rightarrow 02T \rightarrow 00F,$$

$$\text{MCP}(3, 3) : 33T \rightarrow 30F \rightarrow 31T \rightarrow 01F \rightarrow 02T \rightarrow 00F,$$

$$\text{MCP}(4, 3) : 44T \rightarrow 42F \rightarrow 43T \rightarrow 22F \rightarrow 33T \rightarrow 03F \rightarrow 04T \rightarrow 01F \rightarrow 02T \rightarrow 00F,$$

$$\text{MCP}(5, 3) : 55T \rightarrow 44F \rightarrow 54T \rightarrow 51F \rightarrow 52T \rightarrow 22F \rightarrow 33T \rightarrow 03F \rightarrow 04T \rightarrow 01F \rightarrow 02T \rightarrow 00F.$$

5. Related work

In the paper ‘An integration of model-checking with automated proof checking’ by Rajan, Shankar and Srivas [19], PVS [21] is extended to support symbolic model checking of properties stated in the μ -calculus via a link to an external ROBDD-based μ -calculus checker due to G. L. J. M. Janssen of the Department of Electrical Engineering at the Eindhoven University of Technology. Model checking is invoked from PVS via a command that translates higher-order logic goals into boolean formulae that can be printed into a format suitable for input to the external tool. The goal is solved if its translation model checks successfully. Recent developments have improved on the original file-based connection between PVS and the checker, and support for counter-examples is to be provided. PVS is not an LCF-style fully expansive prover, so the model checking command is implemented as a new atomic facility. This differs from the approach here, in which there is a much finer-grained connection, mediated by a calculus of representation judgements. In PVS, the checking of a property against a model is atomic; whereas in our approach this is user-programmed as a fixed-point iteration to compute a sequence of judgements resulting in one that relates the HOL term expressing the goal to its BDD representation. The PVS approach delivers a powerful shrink-wrapped checker; our approach is complementary: it provides a scripting framework for the user to implement his own bespoke tools.

PVS adds model checking to a theorem-proving platform. The dual is to add theorem proving to a model checker. This has been done by McMillan in Cadence SMV [16], which provides problem decomposition commands that split verification goals into components

small enough for model checking. The decomposition is based on deductive rules, for example compositional refinement [15], and implemented by light-weight theorem proving. SMV does not provide a user-programmable scripting facility; rather, each new deduction method is hard-wired into the system. It may be possible for users of HOL to program the kind of algorithm that SMV builds-in as derived rules for calculating representation judgements, but more work is needed to investigate this.

Carl Seger's Voss system [20] and its descendants have 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 data-type. Quantified boolean formulae can be input, and are parsed to BDDs. The normal boolean operations \neg , \wedge , \vee , \equiv , \forall and \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) [11]. 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 [12] 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 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 [10]. 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 [18, 1], but a key idea is that FL is used both as a specification language and as an LCF-style meta-language. The connection between symbolic trajectory evaluation and proof is obtained via a tactic, called `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 with representation judgements provides a somewhat similar programming environment to Voss's FL (though rather more low-level, and with eager rather than lazy evaluation). MuDDy provides BDD operations corresponding to \neg , \wedge , \vee , \equiv , \forall and \exists , and `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, and in addition it allows FL-like BDD programming in ML to be intimately mixed with theorem proving, so that, for example, simplifiers can be directly applied to terms to optimise them for BDD purposes (see the description of 'disjunctive partitioning' in Section 3.3). This is in line with future developments discussed by Joyce and Seger [11], 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.

An approach of Harrison [9] that is in some ways similar to the one here, in that it is

fully expansive, is to implement BDDs inside the HOL system without making use of an external oracle. Harrison has shown that the BDD algorithms provide a way of implementing tautology-checking that is significantly better than the methods previously used in HOL. He found, however, that performance was about a thousand times slower than with a BDD engine implemented in C. By reimplementing some of HOL's primitive rules, performance could be improved by around ten times. Harrison only provided data for boolean equivalence checking. The approach in this paper aims to get near the performance of C-based model checking (by using a BDD package implemented in C), whilst remaining fully expansive, though with a radically extended set of inference rules (namely the rules for deducing BDD representation judgements given in Section 2). Harrison's work is 'logically purer' than ours, but less efficient. The trade-off between purity and efficiency depends on the application (that is, requirements for logical security, ease of maintenance, and so forth), but Harrison's experiments on an internal implementation of BDDs provides a very interesting point between standard theorem proving and the use of an external oracle.

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 outside the critical performance loops.

The approach in this paper provides a secure platform for experimenting with intimate mixtures of deduction and BDD-based symbolic calculation. It could be especially appropriate for experimenting with tricky or complicated algorithms, since the fully expansive approach helps to ensure soundness.

Acknowledgements. The initial implementation of MuDDy, the interface between Moscow ML and BuDDy, is due to Ken Larsen and was supported by EPSRC grant GR/K57343, entitled 'Checking equivalence between synthesised logic and non-synthesisable behavioural prototypes'. He also implemented a prototype HOL oracle that was the starting point for the work reported here. MuDDy is now jointly maintained by Ken Larsen and Jakob Lichtenberg at The IT University, Copenhagen. Special thanks go to Jørn Lind-Nielsen for making his BuDDy code freely available to us.

Further support has been provided by 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', ESPRIT Framework IV LTR 26241 project Prosper ('Proof and specification assisted design environments') and EPSRC grant GR/R27105/01, entitled 'Fully expansive proof and algorithmic verification'.

Mark Aagaard provided some of the information on Voss and its successors described in Section 5.

In addition to the above, the author would like to thank Hasan Amjad, Anuj Dawar, David Greaves, Mark Greenstreet, Joe Hurd, Paul Jackson, John Matthews, Tom Melham, Jesper Møller, Michael Norrish, Atanas Parashkevov, Larry Paulson, Bill Roscoe, Konrad Slind, Mark Staples and Daryl Stewart for various kinds of input to this work, including comments on drafts of this and related papers.

References

1. MARK D. AAGAARD, ROBERT B. JONES and CARL-JOHAN H. SEGER, ‘Lifted-FL: a pragmatic implementation of combined model checking and theorem proving’, *Theorem proving in higher order logics (TPHOLs99)*, Lecture Notes in Comput. Sci. 1690 (Springer, 1999) 323–340. 73
2. SAUL AMAREL, ‘On representation of problems of reasoning about action’, *Machine intelligence 3* (ed. Donald Michie, Edinburgh University Press, 1971) 131–171. 66
3. RANDALL E. BRYANT, *Symbolic boolean manipulation with ordered binary-decision diagrams*, ACM Computing Surveys 24 (1992) 293–318. 56
4. EDMUND M. CLARKE, ORNA GRUMBERG and DORON A. PELED, *Model checking* (The MIT Press, 1999). 64
5. OLIVIER COUDERT, CHRISTIAN BERTHET and JEAN CHRISTOPHE MADRE, ‘Verification of synchronous sequential machines based on symbolic execution’, *Automatic verification methods for finite state systems*, Lecture Notes in Comput. Sci. 407 (ed. J. Sifakis, Springer, 1989) 365–373. 59
6. MIKE GORDON, ‘Reachability programming in HOL98 using BDDs’, *Proc. 13th International Conference on Theorem Proving and Higher Order Logics* (Springer, 2000) 179–196. 62
7. MIKE GORDON, ‘HolBddLib’,
<http://www.cl.cam.ac.uk/~mjc/HolBddLib/>. 57
8. M. J. C. GORDON, R. MILNER and C. P. WADSWORTH, *Edinburgh LCF: a mechanised logic of computation*, Lecture Notes in Comput. Sci. 78 (Springer, 1979). 56, 56
9. JOHN HARRISON, ‘Binary decision diagrams as a HOL derived rule’, *The Computer Journal* 38 (1995) 162–170. 73
10. SCOTT HAZELHURST and CARL-JOHAN H. SEGER, ‘Symbolic trajectory evaluation’, *Formal hardware verification* (ed. Thomas Kropf, Springer, 1997) 3–78. 73
11. J. JOYCE and C. SEGER, ‘The HOL-Voss System: model-checking inside a general-purpose theorem-prover’, *Higher order logic theorem proving and its applications*, 6th International Workshop, HUG’93, Vancouver, B.C., August 11-13 1993, Lecture Notes in Comput. Sci. 780 (ed. J. J. Joyce and C.-J. H. Seger, Springer, 1994), 185–198. 73, 73
12. TREVOR W. S. LEE, MARK R. GREENSTREET and CARL-JOHAN SEGER, ‘Automatic verification of asynchronous circuits’, Tech. Rep. UBC TR 93-40, The University of British Columbia (November, 1993). 73
13. JOHN MCCARTHY, ‘Elaboration tolerance’,
<http://www-formal.stanford.edu/jmc/elaboration/node2.html>. 67
14. KENNETH L. McMILLAN, *Symbolic model checking* (Kluwer Academic Publishers, 1993). 64
15. K. L. McMILLAN, ‘A compositional rule for hardware design refinement’, *Computer-aided verification, CAV ’97*, Lecture Notes in Comput. Sci. (ed. Orna Grumberg, Springer, Haifa, Israel, 1997) 24–35. 73
16. KEN McMILLAN, ‘SMV’,
<http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/>. 72

17. R. MILNER, 'A theory of type polymorphism in programming', *J. Comput. System Sci.* 17 (1978) 348–375. 56
18. JOHN O'LEARY, XUDONG ZHAO, ROBERT GERTH and CARL-JOHAN H. SEGER, 'Formally verifying IEEE compliance of floating-point hardware', *Intel Technology J.*, <http://developer.intel.com/technology/itj/>. 73
19. S. RAJAN, N. SHANKAR and M.K. SRIVAS, 'An integration of model-checking with automated proof checking', *Computer-aided verification, CAV '95*, Lecture Notes in Comput. Sci. 939 (ed. Pierre Wolper, Springer, Liege, Belgium, 1995) 84–97. 72
20. CARL-JOHAN H. SEGER, 'Voss - a formal hardware verification system: User's guide', Tech. Rep. UBC TR 93-45, The University of British Columbia, (December, 1993). 73
21. SRI INTERNATIONAL, 'PVS', <http://www.csl.sri.com/pvs.html>. 72

Michael J. C. Gordon mjcg@cl.cam.ac.uk
<http://www.cl.cam.ac.uk/~mjcg>

University of Cambridge Computer Laboratory
New Museums Site
William Gates Building
J. J. Thomson Avenue
Cambridge CB3 0FD