BDD Representation Judgements in HOL : A Performance Evaluation

Hasan Amjad

University of Cambridge Computer Laboratory

Abstract. This paper describes some preliminary results in evaluating the performance of representation judgements in the Hol98 proof assistant (HOL) [10]. Representation judgements allow "LCF-style" fully-expansive programming of BDD-based symbolic algorithms [6]. They are of the form $\rho t \mapsto b$ meaning "HOL term t is represented by BDD b with respect to variable order ρ" and were introduced in [7] which also evaluated the performance of reachability calculations using this approach. We now extend the evaluation to include a model checker for the Computation Tree Logic (CTL) [1] and provide tentative evidence that the performance is within acceptable bounds.

1 Background and Motivation

Theorem proving and model checking are two complementary approaches to formal verification. Model checking is based on exhaustive exploration of the state space of the system under consideration. Verification is fully automatic and can provide counter-examples for debugging but suffers from the state explosion problem when dealing with complex systems. Theorem proving is based on exploring the space of correctness proofs for the system. It can handle complex formalisms but requires skilled manual guidance for verification and human insight for debugging.

An increasing amount of attention has thus been focused on combining these two approaches (see [17] for a survey). The two basic strategies considered are *abstraction* and *composition* (see [12, 13] for examples). Abstraction tries to abstract the state space of the system to a smaller one – using a theorem prover to show that the abstraction preserves properties of interest – in the hope that the abstracted system can be handled by the model checker. Composition breaks up the problem into smaller subgoals using a theorem prover each of which can be verified with a model checker and the results then composed back in the theorem prover. Representation judgements allow model checking steps to be composed with deductions performed by theorem proving.

HOL is based on the LCF proof assistant ([9]) and is written in Moscow ML. Terms are values of type `term` and can be freely constructed. Theorems are represented as values of type `thm` and can be constructed using axioms and inference rules only i.e.by proof. HOL provides a number of decision procedures, proof searches and simplifiers to aid this task. The goal of the work described here is

to generalise the "LCF-style" fully-expansive approach to support combinations of theorem proving and model checking

This requires generalising the idea of theorem to include BDD representation judgements as described in [7]. A central concern is to ensure that this approach does not create an unacceptable penalty in terms of the performance of the model checker. In this paper we evaluate the performance of a CTL model checker integrated into HOL using this approach. In the next section the CTL model checking algorithm is described briefly. In section 3 we introduce representation judgements, and empirical results are reported in section 4. We conclude with an overview of related work and ideas for the future.

2 CTL Model Checking

Let $\mathcal{M}$ be a finite state machine whose state is a vector $(v_1, \ldots, v_n)$ of boolean variables $v_1, \ldots, v_n$. Let P be some property of interest of states of $\mathcal{M}$ and let S be defined so that $\mathsf{S}i(v_1, \ldots, v_2)$ is true if $(v_1, \ldots, v_n)$ is a state reachable in i or fewer steps from some initial state of $\mathcal{M}$. Then the term $\forall i.\mathsf{S}i(v_1, \ldots, v_n) \Rightarrow \mathsf{P}(v_1, \ldots, v_n)$ is true if all reachable states of $\mathcal{M}$ (represented as a BDD denoted by $[\![\mathcal{M}]\!]^\rho$, where ρ is the variable ordering used for the BDD) satisfy P.

Any quantified boolean formula with boolean free variables can be represented by a binary decision diagram (BDD) ([2]). The term above is quantified over natural numbers, so standard BDD algorithms cannot be used to construct its BDD. It has already been demonstrated in [7] that standard reachability algorithms programmed in HOL can compute the BDD of such natural number quantifications. In this paper we show how standard model checking algorithms can compute the BDDs of HOL formulae representing the sets of states for which a given CTL property holds in a given model.

CTL allows us to describe properties of the states and paths of a *computation tree*. A computation tree is formed by unwinding (infinitely) the transitions of $\mathcal{M}$, starting with the initial states. CTL consists of propositional logic augmented with path quantifiers and temporal operators. The two path quantifiers are **A** (" for all computation paths") and **E** (" for some computation path"). The five basic temporal operators are:

- **X** (unary,"next state") is satisfied by a state if there is a transition to a state satisfying the required property.
- **F** (unary,"future state") is satisfied by a state if there is a sequence of transitions (i.e. a path) to a state satisfying the required property.
- **G** (unary,"globally") is satisfied by a state if all states along the path satisfy the property.
- **U** (binary,"until") is satisfied by a state if the first property holds along the path until a state where the second property holds.
- **R** (binary,"release") is satisfied by a state if the second property holds up to and including the first state where the first property holds. However, the first property need not ever hold. Thus this is the logical dual of **U**.

The syntax of CTL is made out of *state formulae* which are true of states, and *path formulae* which are true of paths (where by path we just mean a sequence of states connected via transitions of the system). A well-formed CTL formula is always a state formula, which it must be because the properties of a system are semantically represented by sets of states of that system. However, we need path formulae because the temporal operators talk about a state with respect to the path of the computation tree the state is on. Formally, formulae of CTL are constructed as follows:

Definition 1 *Let AP be the set of atomic boolean propositions. Then CTL is the smallest set of all state formulae such that*

- T *and* F *are state formulae.*
- $a \in AP$ *is a state formula.*
- *If* f *and* g *are state formulae then* $\neg f$, $f \vee g$ *and* $f \wedge g$ *are state formulae.*
- *If* f *and* g *are state formulae, then* $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, $f\mathbf{U}g$ *and* $f\mathbf{R}g$ *are path formulae.*
- *If* f *is a path formula, then* $\mathbf{A}f$ *and* $\mathbf{E}f$ *are state formulae.*

The ten compound operators thus formed can all be expressed in terms of the three operators **EX**, **EG** and **EU**. We state the following without proof (see [6]) :

Proposition 2

- $\mathbf{AX}f = \neg\mathbf{EX}(\neg f)$
- $\mathbf{EF}f = \mathbf{E}[True\mathbf{U}f]$
- $\mathbf{AG}f = \neg\mathbf{EF}(\neg f)$
- $\mathbf{AF}f = \neg\mathbf{EG}(\neg f)$
- $\mathbf{A}[f\mathbf{U}g] \equiv \neg\mathbf{E}[\neg g\mathbf{U}(\neg f \wedge \neg g)] \wedge \neg\mathbf{EG}(\neg g)$
- $\mathbf{A}[f\mathbf{R}g] \equiv \neg\mathbf{E}[\neg f\mathbf{U}\neg g]$
- $\mathbf{E}[f\mathbf{R}g] \equiv \neg\mathbf{A}[\neg f\mathbf{U}\neg g]$

The CTL symbolic model checking algorithm is simply a procedure that, given $\mathcal{M}$ and a CTL formula P, will return the set (as a BDD) of those states of $\mathcal{M}$ that satisfy P. The notation $R(\bar{v}, \bar{v}')$ denotes the transition relation for $\mathcal{M}$, where $\bar{v}$ is shorthand for the vector $(v_1, \ldots, v_n)$ and $\bar{v}'$ denotes the next state vector. $R(\bar{v}, \bar{v}')$ can easily be expressed as a boolean term (and hence a BDD). Due to Proposition 2 it suffices to consider only **EX**, **EG** and **EU** from the set of operators. We state the following results without proof (see [6] for details).

Proposition 3

- $\mathbf{EG}f$ *is the greatest fixpoint (under subset inclusion over state sets) of the function* $\tau(Z) = f \wedge \mathbf{EX}Z$.
- $\mathbf{E}[f\mathbf{U}g]$ *is the least fixpoint (under subset inclusion over state sets) of the function* $\tau(Z) = f \vee (g \wedge \mathbf{EX}Z)$.

These fixpoints are computed by iteratively computing approximations, each step involving a computation of all states that still satisfy the required property that are reachable in one more step (also called the *relational product* computation). Since the system is finite state and the approximations are increasing sets, we are guaranteed to reach a fixpoint. The model checking algorithm follows.

Definition 4 *The* CTL *model checking procedure* $[\![-]\!]^{\rho}_{\mathcal{M}}$ *is defined recursively over the structure of* CTL *formulae as follows*

- $[\![\mathsf{T}]\!]^{\rho}_{\mathcal{M}} =$ `TRUE` *and* $[\![\mathsf{F}]\!]^{\rho}_{\mathcal{M}} =$ `FALSE`*, where* `TRUE` *and* `FALSE` *are the BDDs of the boolean terms* True *and* False *respectively.*
- $[\![a \in AP]\!]^{\rho}_{\mathcal{M}} =$ *the BDD of the set of states of* $\mathcal{M}$ *in which* a *is true.*
- $[\![\neg f]\!]^{\rho}_{\mathcal{M}} = \neg_b [\![f]\!]^{\rho}_{\mathcal{M}}$ *and* $[\![f \wedge g]\!]^{\rho}_{\mathcal{M}} = [\![f]\!]^{\rho}_{\mathcal{M}} \wedge_b [\![g]\!]^{\rho}_{\mathcal{M}}$ *where* $\neg_b$ *and* $\wedge_b$ *are standard BDD operations.*
- $[\![\exists x f]\!]^{\rho}_{\mathcal{M}} = [\![f]\!]^{\rho}_{\mathcal{M}} \restriction_{x \leftarrow \mathrm{True}} \vee_b [\![f]\!]^{\rho}_{\mathcal{M}} \restriction_{x \leftarrow \mathrm{False}}$ *where* $t \restriction_{x \leftarrow C}$ *denotes the simultaneous substitution of the constant* C *for all free occurrences of the variable* x *in the term* t*.*
- $[\![\mathbf{EX} f(\bar{v})]\!]^{\rho}_{\mathcal{M}} = [\![\exists \bar{v}[f(\bar{v}') \wedge R(\bar{v}, \bar{v}')]]\!]^{\rho}_{\mathcal{M}}$ *i.e. the relational product.*
- $[\![\mathbf{E}[f \mathbf{U} g]]\!]^{\rho}_{\mathcal{M}} = [\![\mu Z.g \vee (f \wedge \mathbf{EX}\, Z)]\!]^{\rho}_{\mathcal{M}}$ *by Proposition 3.*
- $[\![\mathbf{EG} f]\!]^{\rho}_{\mathcal{M}} = [\![\nu Z.f \wedge \mathbf{EX}\, Z]\!]^{\rho}_{\mathcal{M}}$ *by Proposition 3.*

It is clear that BDD operations, in particular the relational product computation, are the workhorses of the algorithm. An efficient model checker therefore requires (among other things) efficient BDD operations. This is why most BDD packages are written in highly optimized C, which immediately makes their soundness suspect. The next section describes an approach to representing BDD operations which we hope will restrict the ability of the programmer to introduce soundness bugs, without slowing down the model checker too much.

3 Representation Judgements

In order to provide a platform for programming model checking procedures from within HOL, the BuDDy ([3]) package has been interfaced to Moscow ML so that BDDs can be manipulated as ML values of type `bdd`. A representation judgement is a type `term_bdd` that represents a 'judgement' $\rho t \mapsto b$. An LCF-style approach to 'proving' such a judgement is implemented much like `thm` implements theorems. Representation judgements are implemented in the `BddRules` structure of the `HolBddLib` library.

Table 1 presents a subset of the 'axioms' and 'rules' that form the primitive operations for `term_bdd`, along with the names of ML functions implementing them (in brackets). As the BuDDy package uses numbers to denote variables, the function ρ maps variables to numbers. The BuDDy function `ithvar` (as interfaced to Moscow ML) simply returns the BDD of the boolean variable v. `TRUE` and `FALSE` denote the corresponding BDDs, and `AND`, `OR`, `NOT`, `IMP`, `BIIMP`, `forall` and `exists` denote the eponymous BDD operations.

Table 1. Primitive Operations for Representation Judgements

$$(\mathsf{BddT})\quad True \mapsto \mathtt{TRUE} \qquad (\mathsf{BddF})\quad False \mapsto \mathtt{FALSE}$$

$$(\mathsf{BddVar})\quad \frac{\rho(v) = n}{\rho v \mapsto \mathtt{ithvar}} \qquad (\mathsf{BddNot})\quad \frac{\rho t \mapsto b}{\rho \neg t \mapsto \mathtt{NOT}\, b}$$

$$(\mathsf{BddAnd})\quad \frac{\rho\, t_1 \mapsto b_1 \quad \rho\, t_2 \mapsto b_2}{\rho\, t_1 \wedge t_2 \mapsto b_1\, \mathtt{AND}\, b_2} \qquad (\mathsf{BddOr})\quad \frac{\rho\, t_1 \mapsto b_1 \quad \rho\, t_2 \mapsto b_2}{\rho\, t_1 \vee t_2 \mapsto b_1\, \mathtt{OR}\, b_2}$$

$$(\mathsf{BddImp})\quad \frac{\rho\, t_1 \mapsto b_1 \quad \rho\, t_2 \mapsto b_2}{\rho\, t_1 \Rightarrow t_2 \mapsto b_1\, \mathtt{IMP}\, b_2} \qquad (\mathsf{BddEq})\quad \frac{\rho\, t_1 \mapsto b_1 \quad \rho\, t_2 \mapsto b_2}{\rho\, t_1 = t_2 \mapsto b_1\, \mathtt{BIIMP}\, b_2}$$

$$(\mathsf{BddForall})\quad \frac{\rho\, t \mapsto b \quad \rho(v_1) = n_1 \ \ldots\ \rho(v_p) = n_p}{\rho \forall v_1 \ldots v_p.t \mapsto \mathtt{forall}(n_1, \ldots, n_p)\, b}$$

$$(\mathsf{BddExists})\quad \frac{\rho\, t \mapsto b \quad \rho(v_1) = n_1 \ \ldots\ \rho(v_p) = n_p}{\rho \exists v_1 \ldots v_p.t \mapsto \mathtt{exists}(n_1, \ldots, n_p)\, b}$$

In practice, quantification of conjunction (i.e.the relational product) occurs frequently and is an expensive operation. BuDDy provides special operations for performing these two steps in one pass, and `BddRules` provides `term_bdd` analogues for this.

Theorem proving support is provided by two rules. The first expresses the fact that logically equivalent terms should have the same BDD (up to variable orderings).

$$(\mathsf{BddEqMp})\quad \frac{\vdash t_1 = t_2 \quad \rho\, t_1 \mapsto b}{\rho\, t_2 \mapsto b} \tag{1}$$

The second rule is the only way to make theorems in `BddRules`. It simply checks to see if the BDD part of the judgement is `TRUE` and if so, returns the term part as a theorem.

$$(\mathsf{TermBddOracle})\quad \frac{\rho\, t \mapsto \mathtt{TRUE}}{\vdash t} \tag{2}$$

This oracle is only as good as the BDD that was produced. Thus it depends on the soundness of BuDDy. By treating BDD operations as inference applications, we restrict the scope of soundness bugs to single operations which are easier to get right. This is why this approach was chosen in favour of a single powerful rule which, given a term, would return its BDD. A utility function `termToTermBdd` : `term` → `term_bdd` is provided, which behaves as expected. `HolBddLib` provides several other utility functions which are explained in detail in [8].

From Table 1, we can derive a modified version of the model checker in Definition 4, this time using representation judgements:

Definition 5 *The* CTL *model checking procedure using representation judgements,* $\mathcal{T}[\![-]\!]^{\rho}_{\mathcal{M}}$*, is defined recursively over the structure of* CTL *formulae as follows*

- $\mathcal{T}[\![\mathsf{T}]\!]^{\rho}_{\mathcal{M}} = \mathsf{BddT}$ *and* $\mathcal{T}[\![\mathsf{F}]\!]^{\rho}_{\mathcal{M}} = \mathsf{BddF}$

- $\mathcal{T}[\![a \in AP]\!]^{\rho}_{\mathcal{M}} = \mathsf{BddVar}(a)$
- $\mathcal{T}[\![\neg f]\!]^{\rho}_{\mathcal{M}} = \mathsf{BddNot}(\mathcal{T}[\![f]\!]^{\rho}_{\mathcal{M}})$ *and* $\mathcal{T}[\![f \wedge g]\!]^{\rho}_{\mathcal{M}} = \mathsf{BddAnd}(\mathcal{T}[\![f]\!]^{\rho}_{\mathcal{M}}, \mathcal{T}[\![g]\!]^{\rho}_{\mathcal{M}})$
- $\mathcal{T}[\![\exists \bar{v} f]\!]^{\rho}_{\mathcal{M}} = \mathsf{BddExists}(\bar{v}, \mathcal{T}[\![f]\!]^{\rho}_{\mathcal{M}})$
- $\mathcal{T}[\![\mathbf{EX} f(\bar{v})]\!]^{\rho}_{\mathcal{M}} = \mathcal{T}[\![\exists \bar{v}[f(\bar{v}') \wedge R(\bar{v}, \bar{v}')]]\!]^{\rho}_{\mathcal{M}}$ *i.e. the relational product.*[1]
- $\mathcal{T}[\![\mathbf{E}[f \mathbf{U} g]]\!]^{\rho}_{\mathcal{M}} = \mathcal{T}[\![\mu Z.g \vee (f \wedge \mathbf{EX}\, Z)]\!]^{\rho}_{\mathcal{M}}$ *by Proposition 3.*
- $\mathcal{T}[\![\mathbf{EG} f]\!]^{\rho}_{\mathcal{M}} = \mathcal{T}[\![\nu Z.f \wedge \mathbf{EX}\, Z]\!]^{\rho}_{\mathcal{M}}$ *by Proposition 3.*

Given a model $\mathcal{M}$ and a CTL formula P that expresses some property of $\mathcal{M}$, we are interested in deriving a theorem that P is satisfied in $\mathcal{M}$. In model checking terms, this means that the set of reachable states of $\mathcal{M}$ that satisfy P is exactly the set of reachable states of $\mathcal{M}$. Using representation judgements, this can be achieved by evaluating

$$\mathsf{TermBddOracle}(\mathsf{BddImp}(\mathcal{T}[\![\mathcal{M}]\!]^{\rho}, \mathcal{T}[\![\mathsf{P}]\!]^{\rho}_{\mathcal{M}})) \tag{3}$$

where $\mathcal{T}[\![\mathcal{M}]\!]^{\rho}$ is just the `term_bdd` analogue of $[\![\mathcal{M}]\!]^{\rho}$. The implication operation is needed because the BDD returned by $\mathcal{T}[\![-]\!]^{\rho}_{\mathcal{M}}$ may contain unreachable states that we are not interested in. TermBddOracle will then either return the theorem or raise an exception.

It is clear from Table 1 that using representation judgements to manipulate BDDs is less efficient than pure BDD manipulation because each BDD operation requires a corresponding operation on the term part of the judgement. It is therefore reasonable to investigate how bad this performance hit might be. We do so in the next section.

4 Empirical Results

The BDD method for testing boolean satisfiability is only of heuristic value: the problem is NP-complete. Using BDDs to represent state sets is similarly claimed to be efficient only in a practical sense. Thus a performance evaluation needs to demonstrate empirical results. Since this is a preliminary study, we have not done extensive benchmarking. We have chosen a single scalable example, that of a synchronous pipelined ALU (Fig. 1), first introduced in [4].

The circuit performs three-address logical operations on a register file (whose registers are denoted by $reg_0, reg_1, \ldots$). The pipeline has three stages:

1. *Fetch*: The operands are read from the register file (the source registers being pointed to by the addresses *src0* and *src1*) into the operand registers *op0* and *op1*.
2. *Execute*: The ALU computes the result and writes it into the pipe register *res*.
3. *Write back*: The result is written back into the register file, at the location pointed to by *dest*.

[1] $R(\bar{v}, \bar{v}')$ can be converted to `term_bdd` form using `termToTermBdd`.

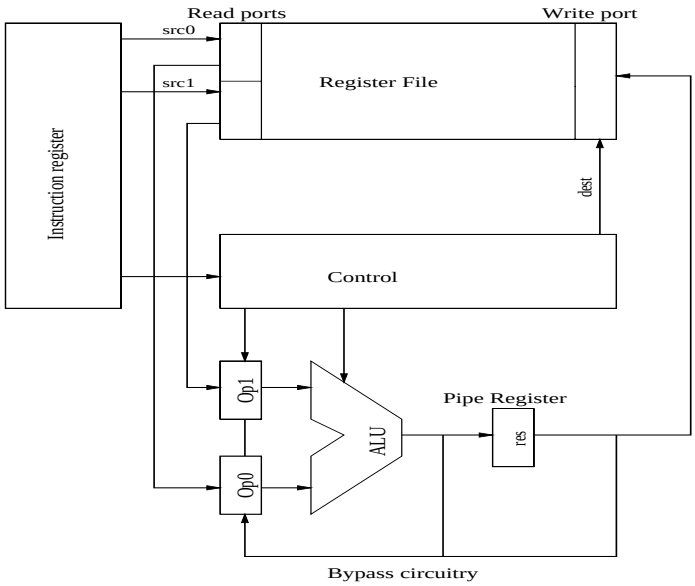

Fig. 1. Simple Pipelined ALU

There is a register bypass path, required for data forwarding. The circuit thus contains both datapath and control circuitry. Addition of extra pipe registers would result in as many new stages, each propagating the result down the pipeline. The number of registers, the number of instructions and the width of the datapath are variable. For simplicity, we fix the number of instructions to two (logical OR and NOR). For the timing measurements, we work with increasing values for the width of the datapath and the number of registers in the register file.

With these parameters, an instruction to the circuit has five components that form the inputs:

- A one-bit opcode, *ctrl*.
- One-bit addresses for the two source and one destination registers (*src0*, *src1* and *dest* respectively).
- A one-bit *stall* input. If this is true, signalling for example a cache miss, then a no-op is propagated down the pipeline.

In this simple circuit, we are concerned with verifying two properties at the RTL level. The first property is expressed by the CTL formula

$$\mathbf{AG}(\neg stall \Rightarrow ((aluop(src\_op0_i, src\_op1_i) = dest\_res_i)) \tag{4}$$

where *aluop* abbreviates a simple propositional formula to ensure that the correct operation is applied given the value of *ctrl*. The place holders *src_op0*,*src_op1* and

dest_res are abbreviations for the source registers for the operands and for the destination register respectively, with the subscript encoding the bit. Thus, this specifies that the destination register is always updated correctly.

To express *src_op0*,*src_op1* and *dest_res* in CTL, we must factor in the latency of the pipeline. For example, for a given operation, the values in the source registers at the time the operation begins are *not* the values that are input to the operation. The values that are required are from the state of the register file after the previous instruction has finished i.e. two clock cycles in the future. Similarly, the value required for the destination register is the value three cycles in the future.

The assumption here is that the an instruction begun at time t will not affect the register file until time $t + 3$, i.e. three clock cycles in the future. To check that this assumption holds, we check that,

$$\mathbf{EX}^k reg_{j,i} \Leftrightarrow \mathbf{AX}^k reg_{j,i} \qquad 1 \leq k \leq 3 \tag{5}$$

where $\mathbf{EX}^k$ abbreviates k applications of $\mathbf{EX}$, and $reg_{j,i}$ is bit i of register j. This can also be done in the model checker.

After accounting for the latency in the pipeline, these abbreviations expand out as follows (for simplicity, we assume there are only two file registers):

$$src\_op0_i = (\neg src1 \wedge \mathbf{AX}(\mathbf{AX}(reg_{0,i}))) \vee (src1 \wedge \mathbf{AX}(\mathbf{AX}(reg_{1,i}))) \tag{6}$$

and similarly for *src_op1*. Similarly, *dest_res* expands to

$$\begin{aligned} dest\_res_i = (\neg dest \wedge \mathbf{AX}(\mathbf{AX}(\mathbf{AX}(reg_{0,i})))) \\ \vee (dest \wedge \mathbf{AX}(\mathbf{AX}(\mathbf{AX}(reg_{1,i})))). \end{aligned} \tag{7}$$

The second property of interest is that for each instruction, the register not being written to (which is all registers if the pipeline stalls) is not changed. So for example for register 1:

$$\mathbf{AG}((stall \vee \neg dest) \Rightarrow (\mathbf{AX}(\mathbf{AX}(reg_{1,i})) = \mathbf{AX}(\mathbf{AX}(\mathbf{AX}(reg_{1,i}))))). \tag{8}$$

For our experiment, we implemented two CTL model checkers in Moscow ML, the first (*PureCheck*) using the BuDDy interface to Moscow ML directly, and the second (*RuleCheck*) using `BddRules` but otherwise the same as *PureCheck* line for line. These were used to verify both properties above for data path widths from one to eight bits, and for two and four registers. Neither program used any state space reduction techniques.

The same variable ordering was used for all BDDs. It is essentially the ordering given in [5] which generally yields good results: the source address registers are closest to the root, with their bits interleaved. Next we interleave the stall and destination address registers, for all three pipeline stages, starting with the fetch stage. These are followed by the opcode, followed by the interleaved bits of the operand, general and pipe registers (this time in big-Endian order).

Table 2 summarises the results of the experiment. Each entry is the ratio of the time taken by *RuleCheck* to the time taken by *PureCheck* to verify both

Table 2. Experimental results: *RuleCheck* time / *Purecheck* time

Bits/Registers	Two	Four
1	62.73	19.26
2	69.97	2.61
3	66.42	2.04
4	65.27	1.73
5	61.48	1.71
6	54.07	1.68
7	43.38	1.67
8	33.98	1.63

properties for the given values of datapath width and number of registers in the register file. We see that though there is a clear penalty for using *RuleCheck*, the difference in performance gets smaller as the system becomes more complex.[2] In fact, at any real world level of complexity, the difference should be quite small.

Intuitively the reason is easy to see: Most of the operations used in the model checking algorithm are linear in the product of the sizes of the operand BDDs. The relational product computation is an exception to this rule. Since this computation occurs frequently, the model checker spends most of its time computing relational products. But the corresponding operation on the term part of the judgement is just a quantification plus a conjunction, which is linear in the sizes of the operand terms i.e. it is relatively efficient in this case.

The same reasoning explains why increasing the number of registers has greater effect: such an increment causes an exponential increase in the level of branching in the computation tree, making the relational product much harder to compute. Increasing the number of bits does not have the same effect because in this circuit the value of each datapath bit is independent of the values of the other datapath bits.

5 Future Possibilities and Related Work

Getting good results with a single class of example is not conclusive evidence in favour of using representation judgements. However, the results can be called encouraging. Further benchmarking is required, with other kinds of examples such as asynchronous circuits, with other kinds of properties that may not require large relational products, and with other implementations of the model checker which use state space reductions and lower-level optimizations. The results also justify a more scientific runtime analysis of the `BddRules` code.

The design of `BddRules` has been strongly influenced by the Voss system [14]. Voss uses a lazy ML-like functional language with BDDs as a built-in datatype.

[2] For the first row of the table, the verification occurs within a second and so is very sensitive to the environment even when averaged over several runs. Hence the odd values.

In [11], Voss is integrated with Hol88 enabling HOL to make external calls to the Voss system, passing subgoals via a translation between HOL and Voss term representations. Later work on Voss has largely been outside the public domain, but the basic approach is that theorem proving is used to split goals into smaller subgoals that are tractable for model checking, and to transform formulae so that they can be checked more efficiently.

On a more general level, work on combining state-space exploration algorithms with deduction has received considerable attention. The interested reader is referred to [17, 13, 15, 16].

References

1. M. Ben-Ari, Z. Manna, A. Pnueli, : The temporal logic of branching time. Acta Informatica 20(1983):207-226.
2. R. E. Bryant : Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. ACM Computing Surveys 24(3):293-318. September 1992.
3. The BuDDy ROBDD Package : `http://www.itu.dk/research/buddy`
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill : Sequential Circuit Verification Using Symbolic Model Checking. In proceedings of the ACM Design Automation Conference 1990.
5. J. R. Burch, E. M. Clarke, D. E. Long : Representing Circuits More Efficiently in Symbolic Model Checking. In proceedings of the ACM Design Automation Conference 1991.
6. E. M. Clarke, O. Grumberg, D. Peled: Model Checking, The MIT Press, 1999.
7. M.J.C. Gordon : Reachability Programming in Hol98 using Binary Decision Diagrams. In the 13th International Conference on Theorem Proving and Higher Order Logics. Springer-Verlag. 2000.
8. M.J.C. Gordon : `HolBddLib` Documentation. Hol98 (*Kananaskis* release) documentation. 2001.
9. M.J.C. Gordon, R. Milner and C. P. Wadsworth: Edinburgh LCF: A mechanised logic of computation. LNCS 78, Springer-Verlag 1979.
10. The HOL Proof Tool : `http://www.cl.cam.ac.uk/Research/HVG/HOL/`
11. J. Joyce, C. Seger : The HOL-Voss System : Model Checking inside a General-Purpose Theorem Prover. In LNCS 780, pages 185-198. Springer-Verlag, 1994.
12. K. L. McMillan : Symbolic Model Checking. Kluwer Academic Pub.,1993.
13. S. Rajan, N. Shankar and M. K. Srivas : An integration of model checking and automated proof checking. In Pierre Wolper ed., CAV '95 in LNCS 939. Springer-Verlag 1995.
14. C-J. H. Seger : Voss - A formal hardware verification system: User's Guide. UBC-TR-93-45, The University of British Columbia, December 1993.
15. The Stanford Temporal Prover: `http://www-step.stanford.edu`
16. The Symbolic Model Prover: `http://www.cs.cmu.edu/~modelcheck/symp.html`
17. T. E. Uribe : Combinations of Model Checking and Theorem Proving. In Third Intl. Workshop on Frontiers of Combining Systems, vol. 1794 of Lecture Notes in Computer Science, pp. 151-170. Springer-Verlag, March 2000