

Programming a Symbolic Model Checker in a Fully Expansive Theorem Prover

Hasan Amjad

University of Cambridge Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK (e-mail: Hasan.Amjad@cl.cam.ac.uk)

Abstract. Model checking and theorem proving are two complementary approaches to formal verification. In this paper we show how binary decision diagram (BDD) based symbolic model checking algorithms may be embedded in a theorem prover to take advantage of the comparatively secure environment without incurring an unacceptable performance penalty.

1 Introduction

Model checking and theorem proving are two complementary approaches to formal verification. Model checking models the system as a state machine and desired properties of the system are expressed as temporal logic formulae that are true in the desired states of the system. 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 models the system as a collection of definitions and desired properties are proved by formal derivations based on these definitions. It can handle complex systems 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 [30] for a survey). In this paper we demonstrate an approach to embedding a model checker in a theorem prover. The expectation is that this will ease combination of state-based and definitional models and the respective property checking techniques. Model checkers are typically written in tightly optimised C with an emphasis on performance. Theorem provers typically are not. Preliminary benchmarking shows that the loss in performance using our approach is within acceptable bounds.

Since our emphasis is on security (in the sense of soundness not being compromised), we have chosen the HOL theorem prover [16] for our task. HOL is based on the HOL logic [15] which is an extension of Church’s simple theory of types [5], and is written in Moscow ML. Terms (of ML type `term`) in the logic can be freely constructed. Theorems (of ML type `thm`) can be constructed using the core axioms and inference rules only, i.e. by proof. This reliance on a very small trusted core is often named the “fully-expansive” approach and gives a high assurance of security.

Symbolic model checking [21] is a popular model checking technique. Sets of states are represented by the BDDs [3] of their characteristic functions. This representation is compact and provides an efficient¹ way to test set equality and do image computations. This is useful because evaluating temporal logic formulae almost always requires a fixed point computation that relies on image computations to compute the next approximation to the fixed point and a set equality test to determine termination. Most of the work is done by the underlying BDD engine.

By representing primitive BDD operations as inference rules added to the core of the theorem prover, we can model the execution of a model checker for a given property as a formal derivation tree rooted at the required property. These inference rules are hooked to a high performance BDD engine [?] external to the theorem prover. Thus the loss of performance is low, and the security of the theorem prover is compromised only to the extent that the BDD engine or the BDD inference rules may be unsound. Since we do almost everything within HOL and use only the most primitive BDD operations, we expect a higher assurance of security than from an implementation that is entirely in C.

2 Representing BDDs in the Theorem Prover

In order to provide a platform for programming model checking procedures from within HOL, the BuDDy [?] BDD engine has been interfaced to ML so that BDDs can be manipulated as ML values of type `bdd`. To represent BDD operations as inference rules, we use judgements of ML type `term_bdd` of the form

$$\rho t \mapsto b$$

where t is a HOL term (which we shall call the *term part* of the judgement) and b is a BDD (called the *BDD part* of the judgement). The only free variables of t are the boolean variables used in b (also called the *support* of b). Intuitively, if we collect these boolean variables together in a tuple s , the judgement is saying that for all assignments to the variables in s , an assignment will satisfy t if and only if it is also a satisfying assignment for b . The variable map ρ (of ML type `vm`) maps HOL variables to numbers. The map is required because BuDDy uses numbers to represent variables and for the moment its presence is only a technical requirement.

Our approach to ‘proving’ such a judgement is implemented analogously to the manner in which we prove theorems i.e. BDD representation judgements cannot be freely constructed but may be derived using primitive inference steps.

Table 1 presents a subset (that is relevant to our work) of the rules that form the primitive operations for BDD judgements, along with the names of ML functions implementing them (in brackets). The BuDDy function `ithvar` (as interfaced to ML) simply returns the BDD of the boolean variable v . `TRUE` and `FALSE` denote the corresponding BDDs, `T` and `F` are HOL terms for truth and

¹ The problem is NP-complete. So this efficiency is of heuristic value only.

Table 1. Primitive Operations for Representation Judgements

$$\begin{array}{l}
(\text{BddT} : \text{vm} \rightarrow \text{term_bdd}) \quad \rho \text{T} \mapsto \text{TRUE} \quad (\text{BddF} : \text{vm} \rightarrow \text{term_bdd}) \quad \rho \text{F} \mapsto \text{FALSE} \\
\\
(\text{BddVar} : \text{vm} \rightarrow \text{term} \rightarrow \text{term_bdd}) \quad \frac{\rho(v) = n}{\rho v \mapsto \text{ithvarn}} \\
(\text{BddNot} : \text{term_bdd} \rightarrow \text{term_bdd}) \quad \frac{\rho t \mapsto b}{\rho \neg t \mapsto \text{NOT } b} \\
(\text{BddAnd} : \text{term_bdd} * \text{term_bdd} \rightarrow \text{term_bdd}) \quad \frac{\rho t_1 \mapsto b_1 \quad \rho t_2 \mapsto b_2}{\rho t_1 \wedge t_2 \mapsto b_1 \text{ AND } b_2} \\
(\text{BddOr} : \text{term_bdd} * \text{term_bdd} \rightarrow \text{term_bdd}) \quad \frac{\rho t_1 \mapsto b_1 \quad \rho t_2 \mapsto b_2}{\rho t_1 \vee t_2 \mapsto b_1 \text{ OR } b_2} \\
\\
(\text{BddAppEx} : \text{termlist} \rightarrow \text{term_bdd} * \text{term_bdd} \rightarrow \text{term_bdd}) \\
\frac{\rho(v_1) = n_1 \dots \rho(v_p) = n_p \quad \rho t_1 \mapsto b_1 \quad \rho t_2 \mapsto b_2}{\rho \exists v_1 \dots v_p. t_1 \text{ op } t_2 \mapsto \text{appex } b_1 b_2 (n_1, \dots, n_p)}
\end{array}$$

falsity, and NOT, AND and OR denote the eponymous BDD operations (see [14] for details).

In practice, existential quantification of conjunction (often called the relational product or image computation) occurs frequently and is an expensive operation. BuDDy provides a special operation `appex` for performing quantification of a boolean operation in one pass, and we have a BDD inference rule `BddAppEx` corresponding to it.

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).

$$(\text{BddEqMp}) \quad \frac{\rho t_1 \mapsto b}{\rho t_2 \mapsto b} \vdash t_1 \Leftrightarrow t_2 \tag{1}$$

This rule enables us to use higher-order predicates in the term part of judgements to succinctly express the propositional content of the BDD part of the judgement.

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

$$(\text{BddOracleThm}) \quad \frac{\rho t \mapsto \text{TRUE}}{\vdash t} \tag{2}$$

This theorem is only as good as the BDD that was produced: the soundness of it depends on the soundness of the BDD engine and of our representation judgement inference rules. By treating BDD operations as inference applications, we restrict the scope of soundness bugs to single operations which are easy 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 formal proof of the soundness of this calculus is underway.

3 Model Checking

Our general approach is independent of the choice of temporal logic. We shall apply it to the model checking procedure for the propositional μ -calculus L_μ from [19]. L_μ is very expressive and a model checker for it gives us model checkers for the popular temporal logics CTL and LTL.²

Formulae of L_μ describe properties of a system that can be represented as a state machine. In particular, the semantics of a formula is the set of states of the system satisfying the formula. The model checking algorithm computes this set given a formula and a system.

We need to make the notion of “system” precise. For our purposes, the system is represented by a Kripke structure. If AP is the set of atomic propositions relevant to the system we wish to model, then a Kripke structure over AP is defined as follows:

Definition 1. *A Kripke structure M over AP is a tuple (S, S_0, T, L) where*

- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- T is the set of actions (or transitions or program letters) such that for any action $a \in T$, $a \subseteq S \times S$.
- $L : S \rightarrow 2^{AP}$ labels each state with the set of atomic propositions true in that state.

We now present the syntax of L_μ , essentially as given in [6].

Definition 2. *Let VAR be the set of relational variables, $p \in AP$ be an atomic proposition and $a \in T$ be an action. Then if f and g are L_μ formulas, so are: True, False, p , $\neg f$, $f \wedge g$, $f \vee g$ (the propositional fragment); $[a]f$ and $\langle a \rangle f$ (the modal fragment); P , $\mu Q.f$ and $\nu Q.f$ (the relational or recursive fragment where $\{P, Q\} \subseteq VAR$ and all occurrences of Q in the negated normal form³ (NNF) of f are not negated).*

We often use the term “variable” instead of “relational variable” or “propositional variable”; the meaning should be clear from the context. We use p, p_0, \dots to denote propositional atoms and Q, Q_0, Q_1, \dots for relational variables. In the formulas $\mu Q.f$ and $\nu Q.f$, μ and ν are considered binders on Q , and thus there is the standard notion of bound and free variables. We use $f(Q_1, Q_2, \dots)$ to denote that Q_1, Q_2, \dots occur free in f .

² Though from a practical viewpoint, model checkers for L_μ are not as efficient as say SMV [21] or SPIN [17]. Note also that for logics that do not admit a direct syntactic embedding into L_μ , e.g. LTL, the translation into L_μ is non-trivial and a fully-expansive translation provides much needed assurance of soundness.

³ This is a syntactic transformation that pushes all negations inwards to the atoms using the De Morgan style dualities $\neg(f \wedge g) = \neg f \vee \neg g$, $\neg(f \vee g) = \neg f \wedge \neg g$, $\neg[a]f = \langle a \rangle \neg f$, $\neg \langle a \rangle f = [a] \neg f$, $\neg \mu Q.f(Q) = \nu Q. \neg f(\neg Q)$ and $\neg \nu Q.f(Q) = \mu Q. \neg f(\neg Q)$ and the involutiveness of \neg .

The semantics of a formula f is written $\llbracket f \rrbracket_{Me}$, where M is a Kripke structure and the environment $e : VAR \rightarrow 2^S$ holds the state sets corresponding to the free relational variables of f . By $e[Q \leftarrow W]$ we mean the environment that has $e[Q \leftarrow W]Q = W$ but is the same as e otherwise. We now define $\llbracket f \rrbracket_{Me}$.

Definition 3. *The semantics of L_μ are defined recursively as follows*

- $\llbracket True \rrbracket_{Me} = S$ and $\llbracket False \rrbracket_{Me} = \emptyset$
- $\llbracket p \rrbracket_{Me} = \{s \mid p \in L(s)\}$
- $\llbracket Q \rrbracket_{Me} = e(Q)$
- $\llbracket \neg f \rrbracket_{Me} = S \setminus \llbracket f \rrbracket_{Me}$
- $\llbracket f \wedge g \rrbracket_{Me} = \llbracket f \rrbracket_{Me} \cap \llbracket g \rrbracket_{Me}$
- $\llbracket f \vee g \rrbracket_{Me} = \llbracket f \rrbracket_{Me} \cup \llbracket g \rrbracket_{Me}$
- $\llbracket \langle a \rangle f \rrbracket_{Me} = \{s \mid \exists t. s \xrightarrow{a} t \wedge t \in \llbracket f \rrbracket_{Me}\}$
- $\llbracket [a] f \rrbracket_{Me} = \{s \mid \forall t. s \xrightarrow{a} t \Rightarrow t \in \llbracket f \rrbracket_{Me}\}$
- $\llbracket \mu Q.f \rrbracket_{Me}$ is the least fix-point of the predicate transformer $\tau : 2^S \rightarrow 2^S$ given by $\tau(W) = \llbracket f \rrbracket_{Me}[Q \leftarrow W]$
- $\llbracket \nu Q.f \rrbracket_{Me}$ is the greatest fix-point of the predicate transformer $\tau : 2^S \rightarrow 2^S$ given by $\tau(W) = \llbracket f \rrbracket_{Me}[Q \leftarrow W]$

Environments can be given a partial ordering \subseteq under component-wise subset inclusion. By Tarski's fix-point theorem [29], if

$$e[Q \leftarrow W] \subseteq e'[Q \leftarrow W'] \Rightarrow \llbracket f(Q) \rrbracket_{Me}[Q \leftarrow W] \subseteq \llbracket f(Q) \rrbracket_{Me'}[Q \leftarrow W']$$

i.e. the semantics evaluate monotonically over environments, then the existence of fix-points is guaranteed. In fact, since S is finite, monotonicity implies continuity [29], which gives

Proposition 4. $\llbracket \mu Q.f \rrbracket_{Me} = \bigcup_i \tau^i(\emptyset)$ and $\llbracket \nu Q.f \rrbracket_{Me} = \bigcap_i \tau^i(S)$

where $\tau^i(Q)$ is defined by $\tau^0(Q) = Q$ and $\tau^{i+1} = \tau(\tau^i(Q))$. So we can compute the fix-points by repeatedly applying τ to the result of the previous iteration, starting with $\llbracket False \rrbracket_{Me}$ for least fix-points and $\llbracket True \rrbracket_{Me}$ for greatest fix-points. Since S is finite, the computation stops at some $k \leq |S|$, so that the least fix-point is given by $\tau^k(\llbracket False \rrbracket_{Me})$ and the greatest fix-point by $\tau^k(\llbracket True \rrbracket_{Me})$. We then have that

Proposition 5. *If $\tau^i(Q) = \tau^{i+1}(Q)$ then $k = i$.*

Essentially, the semantics describe the model checking algorithm itself. An executable version of Proposition 5 would rely on being able to efficiently test state sets for equality. Since states are boolean tuples, we can represent state sets by the BDDs of their characteristic functions. Since the semantics are constructed using set operations, every step of the algorithm can be represented by an operation on BDDs. Hence every step can be represented by the application of a BDD representation judgement inference rule.

From Table 1, we can give a more concrete semantics for μ -formulae, this time using representation judgements.

Definition 6. The L_μ model checking procedure $\mathcal{T}[-]_M^\rho e$ is defined recursively over the structure of μ -formulae as follows

- $\mathcal{T}[\text{True}]_M^\rho e = \text{BddT } \rho$ and $\mathcal{T}[\text{False}]_M^\rho e = \text{BddF } \rho$
- $\mathcal{T}[p]_M^\rho e = \text{BddVar}(\rho p)$
- $\mathcal{T}[\neg f]_M^\rho e = \text{BddNot}(\mathcal{T}[f]_M^\rho e)$
- $\mathcal{T}[f \wedge g]_M^\rho e = \text{BddAnd}(\mathcal{T}[f]_M^\rho e, \mathcal{T}[g]_M^\rho e)$
- $\mathcal{T}[f \vee g]_M^\rho e = \text{BddOr}(\mathcal{T}[f]_M^\rho e, \mathcal{T}[g]_M^\rho e)$
- $\mathcal{T}[\langle a \rangle f]_M^\rho e = \text{BddAppEx}(\wedge, \llbracket M.T(a) \rrbracket, \mathcal{T}[f]_M^\rho e)$
 where $\llbracket M.T(a) \rrbracket$ is the BDD judgement for the action a
- $\mathcal{T}[\llbracket a \rrbracket f]_M^\rho e = \mathcal{T}[\neg \langle a \rangle \neg f]_M^\rho e$
- $\mathcal{T}[\mu Q.f]_M^\rho e = \bigcup_{i=0}^k \tau^i(\emptyset)$
 where $\tau = \lambda W. \mathcal{T}[f]_M^\rho e[Q \leftarrow W]$ and $\tau^k(\emptyset) = \tau^{k+1}(\emptyset)$
- $\mathcal{T}[\nu Q.f]_M^\rho e = \bigcap_{i=0}^k \tau^i(S)$
 where $\tau = \lambda W. \mathcal{T}[f]_M^\rho e[Q \leftarrow W]$ and $\tau^k(S) = \tau^{k+1}(S)$

Executing the procedure in Definition 6 for some L_μ formula f with respect to a Kripke structure M and environment e will yield a judgement $\rho f' \mapsto b$ where ρ is the variable map, f' is the *boolean* semantic equivalent of f and b is the BDD of f' . So f' is likely to be large, unreadable and unsuitable for further manipulation by the theorem prover. Ideally, we would like f' to be some term expressing satisfiability of f in M and e i.e. we would like to obtain a judgement

$$\rho (s \models_M^e f) \mapsto b,$$

where the state s is a tuple of free boolean variables corresponding to the support of b . Deriving a judgement in the form above is what we shall now attempt. To do this, we must provide a definition and semantics of L_μ to the theorem prover.

4 Model Checking in the Theorem Prover

This section presents a mechanical formalisation of the theory and algorithms described above. To save space, the lengthy and theorem prover specific formal proofs are not given. Proof sketches are provided where they aid intuition.

4.1 Formalising the theory

The formalisation goes along the lines of section 3. The propositional atoms $p \in AP$ have type β . We use α to denote the type of a state. During a model checking run α would be specialised to $(\beta \times \beta \times \dots \times \beta)$ where the size of the product would be $|AP|$. Currently β is always specialised to the HOL boolean type `bool`. Kripke structures are represented by a simple record type KS , with fields $S : \alpha \text{ set}$, $S0 : \alpha \text{ set}$, $T : \text{string} \rightarrow (\alpha \times \alpha) \rightarrow \text{bool}$ and $L : \alpha \rightarrow \beta \rightarrow \text{bool}$ representing components so named in Definition 1.⁴ Note that action names are modelled as strings.

⁴ Sets in HOL are not ZF sets [31]. A set $S : \alpha \text{ set}$ in HOL is a predicate of type $\alpha \rightarrow \text{bool}$. Thus set membership $x \in S$ is equivalent to the application Sx . We use both notations as appropriate.

At the time of writing HOL did not support predicate subtypes, so a well-formedness predicate on Kripke structures had to be defined separately.

Definition 7. A Kripke structure M is well-formed if $S = \mathcal{U} : (\alpha \text{ set})$ where \mathcal{U} is the universal set of all things of type α .

The identification of S with all states is not a strict requirement. It is a technical convenience and does not result in loss of generality in the current context.

Formulas of L_μ are represented by a simple recursive datatype. The syntactic constraint on bound variables (see Definition 2) is enforced by a well-formedness predicate on μ -formulas.

Definition 8. A well-typed L_μ formula f is well-formed, wff f , if and only if all subformulas of f are well-formed. However, if $f \equiv \mu Q.g$ or $f \equiv \nu Q.g$ then we additionally require that $\neg Q \not\sqsubseteq \text{NNF}g$.

The subformula relation \sqsubseteq is defined as expected. The definition of negated normal form (sketched earlier in the footnote to Definition 2) requires a limited form of substitution (the notation $f(\neg Q)$ abbreviates $f[\neg Q/Q]$). Since HOL has no native support for variable binding in higher order abstract syntax,⁵ the definition of negated normal form has to explicitly avoid free variable capture. We shall assume that all formulas are well-formed and elide the well-formedness condition from theorem statements to avoid clutter.

The heart of the formalisation is the formal semantics, which follow Definition 3.

Definition 9. The formal semantics of the μ -calculus for a Kripke structure M and environment e are defined by the mutual recursion

$$\begin{aligned}
\mathcal{FS}[\text{True}]_M^e &= S \wedge \\
\mathcal{FS}[\text{False}]_M^e &= \emptyset \wedge \\
\mathcal{FS}[p]_M^e &= \{s \mid s \in S \wedge p \in M.Ls\} \wedge \\
\mathcal{FS}[Q]_M^e &= \{s \mid s \in S \wedge eQs\} \wedge \\
\mathcal{FS}[\neg f]_M^e &= S \setminus \mathcal{FS}[f]_M^e \wedge \\
\mathcal{FS}[f \vee g]_M^e &= \mathcal{FS}[f]_M^e \cup \mathcal{FS}[g]_M^e \wedge \\
\mathcal{FS}[f \wedge g]_M^e &= \mathcal{FS}[f]_M^e \cap \mathcal{FS}[g]_M^e \wedge \\
\mathcal{FS}[\langle a \rangle f]_M^e &= \{s \mid \exists q. q \in S \wedge s \xrightarrow{a} q \wedge q \in \mathcal{FS}[f]_M^e\} \wedge \\
\mathcal{FS}[\langle a \rangle f]_M^e &= \{s \mid \forall q. q \in S \wedge s \xrightarrow{a} q \Rightarrow q \in \mathcal{FS}[f]_M^e\} \wedge \\
\mathcal{FS}[\nu Q.f]_M^e &= \{s \mid \forall n. s \in \text{FP } f \text{ } Q \text{ } M \text{ } e[Q \leftarrow S] n\} \wedge \\
\mathcal{FS}[\mu Q.f]_M^e &= \{s \mid \exists n. s \in \text{FP } f \text{ } Q \text{ } M \text{ } e[Q \leftarrow \emptyset] n\} \wedge \\
\text{FP } f \text{ } Q \text{ } M \text{ } e 0 &= eQ \wedge \\
\text{FP } f \text{ } Q \text{ } M \text{ } e (n + 1) &= \mathcal{FS}[f]_M^{e[Q \leftarrow \text{FP } f \text{ } Q \text{ } M \text{ } e n]}
\end{aligned}$$

⁵ Providing automatic α -conversion is an active research area. Work done on this includes [12, ?, 13]

Table 2. Satisfiability theorems for model checking based on Definition 9

$\forall s M e.$	$s \models_M^e \text{True} \Leftrightarrow \mathbf{T}$	(3)
$\forall s M e.$	$s \models_M^e \text{False} \Leftrightarrow \mathbf{F}$	(4)
$\forall s M e.$	$s \models_M^e p \Leftrightarrow p \in M.Ls$	(5)
$\forall s M e f.$	$s \models_M^e \neg f \Leftrightarrow s \not\models_M^e f$	(6)
$\forall s M e f g.$	$s \models_M^e f \wedge g \Leftrightarrow s \models_M^e f \wedge s \models_M^e g$	(7)
$\forall s M e f g.$	$s \models_M^e f \vee g \Leftrightarrow s \models_M^e f \vee s \models_M^e g$	(8)
$\forall s M e Q.$	$s \models_M^e Q \Leftrightarrow e Q s$	(9)
$\forall s M e a f.$	$s \models_M^e \langle a \rangle f \Leftrightarrow \exists q.(M.Ta)(s, q) \wedge q \models_M^e f$	(10)
$\forall f M e Q s X n.$	$s \models_M^{e[Q \leftarrow FP f Q M e[Q \leftarrow X]^n]} f$	
	$\Leftrightarrow FP f Q M e[Q \leftarrow X](n+1) s$	(11)

Since we need to test semantics for boolean satisfiability (Proposition 5 requires this), we need to define satisfaction of a formula in a state.

Definition 10. A μ -calculus formula f is satisfied by a state s of a Kripke structure M under an environment e if and only if $s \in \mathcal{FS}[[f]]_M^e$. We denote this by

$$s \models_M^e f$$

For efficiency, theorems expressing satisfiability of all μ -calculus operators (see Table 2) are proved in terms of \models using Definition 9, assuming the Kripke structure is well-formed.⁶ This is trivial due to the simple connection between \models and $\mathcal{FS}[-]$.

For fixed point computations, we require theorems that tell us when a fixed point has been reached. The first step is to prove monotonicity of the semantics of a formula with respect to the free variables i.e. the environment. We show the results for the least fixed point operator only (greatest fixed points follow by duality).

Theorem 11. For a well-formed Kripke structure M and well-formed μ -calculus formula $\mu Q.f$, we have that

$$\begin{aligned} & \forall e e' X Y. \\ & \quad X \subseteq Y \\ & \wedge \quad \forall Q'. \text{ if } (\neg Q' \sqsubseteq \text{NNF } f) \text{ then } e Q' = e' Q' \text{ else } e Q' \subseteq e' Q' \\ & \Rightarrow \quad \mathcal{FS}[[f]]_M^{e[Q \leftarrow X]} \subseteq \mathcal{FS}[[f]]_M^{e'[Q \leftarrow Y]} \end{aligned}$$

⁶ The missing modal and fix-point operator theorems follow by duality. Fix point computations require several other satisfiability theorems discussed later.

Proof Sketch By the remarks preceding Proposition 4. Monotonicity can be shown by the observations that

1. Each of the operators except negation is monotonic.
2. Only relational variables occur negated in the *NNF* of a formula.
3. Bound variables occur non-negated in the *NNF* of a formula.
4. The negated normal form of f has the same semantics as f .

So we can effectively remove all negations from a formula without affecting the semantics. Monotonicity follows immediately. The second hypothesis in the antecedent expressing subset inclusion of environments is more complex than expected because it needs to account for the possibility of the environment having variables not occurring in the formula. This is a technical side-effect of HOL not supporting predicate subtypes. In the case of $e = e'$, this is trivial. \square

Using monotonicity, we are able to formally derive the equivalent of Proposition 5.

Theorem 12. *For a well-formed Kripke structure M and well-formed μ -calculus formula $\mu Q.f$, we have that*

$$\begin{aligned} & \forall e n. \\ & FP f Q M e[Q \leftarrow \emptyset]n = FP f Q M e[Q \leftarrow \emptyset](n + 1) \\ \Rightarrow & \mathcal{FS}[\mu Q.f]_M^e = FP f Q M e[Q \leftarrow \emptyset]n \end{aligned}$$

Proof Sketch It follows from Theorem 11 and [29] that $FP f Q M e[Q \leftarrow \emptyset]n$ is the least upper bound (under subset inclusion) of all $FP f Q M e[Q \leftarrow \emptyset]m$ for $m \leq n$, and that $FP f Q M e[Q \leftarrow \emptyset]n = FP f Q M e[Q \leftarrow \emptyset]m$ for $m \geq n$. Then

$$\begin{aligned} & FP f Q M e[Q \leftarrow \emptyset]n \\ &= \bigcup_{i \in \mathbb{N}} FP f Q M e[Q \leftarrow \emptyset]i \\ &= \mathcal{FS}[\mu Q.f]_M^e \end{aligned}$$

using Definition 9. \square

4.2 Formalising the model checker

We wish to be able to pass a well-formed Kripke structure M , a well-formed formula f , an environment e and a variable map ρ to the model checking procedure \mathcal{T} which returns a judgement of the form

$$\rho (s \models_M^e f) \mapsto \mathbf{b} \tag{12}$$

where the state s is a boolean tuple comprising the atomic propositions M is defined over.

Preliminaries The first step in implementing the model checker is to prove the well-formedness of M and f . This is trivial but does require that both be HOL terms. Representing M as a HOL term throughout the model checking would be inefficient as the T component can be quite large. So we simply use M as an abbreviation for the entire structure. Additionally, evaluation of $T(a)$ directly from the term representation is $O(|T|)$. So we construct an ML binary search tree (BST) T_m which maps each action $a \in T$ to its transition relation. With T_m we can evaluate $T(a)$ in time $O(\log_2 |T|)$. Since M does not change during the model checking, we do not require write access to it. Once we have proved well-formedness, the theorems of Table 2 are specialised to f , M and s .

The environment occurs in the term part of the result and therefore also needs to be represented as a HOL term. However, environments change during every iteration of a fixed point computation. Thus they cannot be abbreviated as above without creating HOL definitions on the fly, which is messy and slow. Fortunately the term representation of environments is not large so they can be represented directly. The changing environments mean that all the satisfiability theorems we proved in Table 2 change with each iteration. The solution is to specialise them with the updated environment for each iteration. Finally, we construct an ML BST e_m which is an efficient version of e , analogous to T_m .

The kernel The core algorithm is based on Definition 6, i.e. by recursion over the L_μ formula. Each step in the recursion consists of a one application of a BDD inference rule from Table 1 followed by an application of **BddEqMp** (from 1) together with the appropriate theorem (now specialised) from Table 2. Thus each step results in a judgement of the form of 12. In the end we use **BddOracleThm** to derive the final theorem.

As a trivial example, suppose we have a Kripke structure M over $\{p_0\}$ and an environment e that maps Q to the set $\llbracket \neg p_0 \rrbracket_M^e$. Then $p_0 \vee Q$ should hold in all states. This theorem is derived (with mild notational abuse) by

$$\frac{\text{BddEqMp} \frac{\text{BddVar} \frac{\rho(p_0) \mapsto \text{ithvar } 0 \quad \rho(p_0) = 0}{\rho \models p_0 \mapsto \text{ithvar } 0}}{\rho \models p_0 \mapsto \text{ithvar } 0} \vdash p_0 \leftrightarrow s \models_M^e p_0 \quad \text{BddNot} \frac{\text{BddVar} \frac{\rho(p_0) \mapsto \text{ithvar } 0 \quad \rho(p_0) = 0}{\rho \neg p_0 \mapsto \llbracket \neg p_0 \rrbracket_M^e}}{\rho \models_M^e Q \mapsto \llbracket \neg p_0 \rrbracket_M^e} \vdash \neg p_0 \leftrightarrow s \models_M^e Q}{\text{BddOr} \frac{\text{BddEqMp} \frac{\rho \models p_0 \vee s \models_M^e Q \mapsto \text{TRUE}}{\rho \models p_0 \vee s \models_M^e Q} \vdash s \models_M^e p_0 \vee s \models_M^e Q \leftrightarrow s \models_M^e p_0 \vee Q \quad \text{BddOracleThm} \frac{\rho \models p_0 \vee s \models_M^e Q \mapsto \text{TRUE}}{\vdash s \models_M^e p_0 \vee Q}}{\rho \models p_0 \vee s \models_M^e Q \leftrightarrow s \models_M^e p_0 \vee Q}}$$

where the side-conditions to **BddEqMp** are derived from Table 2 and Defn. 3.

In the case of propositional atoms, relational variables and the modal operators, we do not have one theorem but sets of theorems AP_t , VAR_t and T_t indexed by the name of the atom, relational variable or action respectively; the appropriate theorem is picked out for passing to **BddEqMp**. The theorems in AP_t , VAR_t and T_t are obtained by specialising the theorems 5, 9 and 10 of Table 2. The HOL terms for the maps L and T are constructed so that a “lookup” i.e. a rewrite evaluating the application of an atom or action name to L or T respectively, has the same asymptotic cost as a lookup in a Patricia tree. All theorem sets are cached in ML BSTs for efficiency. The case for fixed point operators is more involved and is discussed in the next section.

It should be noted that the model checker derives every step from theorems proved in HOL and thus the result is provably correct (this does not apply to the BDD engine, but so far we have not *used* the BDD part of the representation judgements though of course it is updated by the inference rules). As we shall show, the performance penalty for this proof has been acceptable in regression tests. This is the justification for using representation judgements.

Computing fixed points We limit our discussion to computing least fixed points and elide well-formedness predicates (for the Kripke structure and formula). If in a recursive descent \mathcal{T} encounters a subformula $\mu Q.f$, we would like to derive the judgement

$$\rho (s \models_M^e \mu Q.f) \mapsto b$$

where, if we consider the BDD b as a set, we would like $b = \bigcup_{i=0}^k \tau^i(\emptyset)$ where $\tau = \lambda W. \mathcal{T} \llbracket f \rrbracket_M^e [Q \leftarrow W]$ and $\tau^k(\emptyset) = \tau^{k+1}(\emptyset)$ (see Definitions 3 and 6).

Thus, by Theorem 12, if we can show that in the $(i+1)^{th}$ iteration

$$FP f Q M e [Q \leftarrow \emptyset] i = FP f Q M e [Q \leftarrow \emptyset] (i+1) \quad (13)$$

we have the required result (using Definition 10).

To start, we require a “bootstrap” theorem.

Theorem 13.

$$\forall f M e Q s. FP f Q M e [Q \leftarrow \emptyset] 0 s = F$$

Proof Sketch Immediate from Definition 9 and the HOL definition of \emptyset . \square

Now we update the environment e with the mapping

$$[Q \leftarrow \rho (FP f Q M e [Q \leftarrow \emptyset] 0) \mapsto \text{FALSE}]$$

justified by Theorem 13 and the **BddF** rule from Table 1. Intuitively we can say that $\rho (FP f Q M e [Q \leftarrow \emptyset] 0) \mapsto \text{FALSE}$ is the mechanised version of $\tau^0(\emptyset)$.

For the iteration, we require a “substitution” theorem.

Theorem 14.

$$\forall f Q M e n s. s \models_M^{e[Q \leftarrow FP f Q M e (n+1)]} Q \Leftrightarrow s \models_M^{e[Q \leftarrow FP f Q M e n]} f$$

Proof Sketch Simplification with Definitions 9 and 10. \square

Then, for the i^{th} iteration, a single call to the model checker returns the judgement⁷

$$\rho (s \models_M^{e[Q \leftarrow FP f Q M e [Q \leftarrow \emptyset] i]} f[f/Q]) \mapsto b$$

We use this and **BddEqMp** together with equation 11 of Table 2 and Theorem 14 to derive

$$\rho (FP f Q M e [Q \leftarrow \emptyset] (i+1)) \mapsto b$$

⁷ The substitution notation denotes that syntactic recursion occurs in the term part.

which is the mechanised version of $\tau^{i+1}(\emptyset)$. This is then made the new value of Q in e before calling the model checker again. At each iteration, all satisfiability theorems are recreated by specialising the theorems (or theorem sets) from Table 2 with the updated environment e . However, the theorem in VAR_t corresponding to Q has to be proved from scratch each time since the value of Q changes with every iteration.

After each call to the model checker, we check if the BDD parts of the two most recent iterations are equal. If so, we are able to derive the condition in equation 13 using `BddOracleThm` and stop. This is the only point where BDDs are actually used. Thus the result is guaranteed to be correct assuming the BDD representation judgement inference rules are sound (modulo the soundness of the BDD engine and HOL itself). We have already commented on the relatively high likelihood of this.

5 Empirical Results

This section presents the results of the first round of benchmarks. We compare the speed of our model checker with one we wrote in plain ML that bypasses HOL and works directly with the BDD engine. The numbers given in Table 3 are ratios normalised to the plain model checker’s time. The lower the numbers, the lower the performance hit caused by the theorem proving overhead.

Our “fully-expansive” model checker’s CPU time can be split into two phases: setting up the model (overhead time) and checking it. The overhead work needs to be done only once per model. The benchmarks show the amortised checking time by spreading overhead time over the various properties checked for the same model.

Table 3. Performance: fully-expansive to non-fully-expansive execution time ratio

Datapath/Address space	1	2	3
1	5.58	9.7	8.04
2	5.52	9.21	4.39
3	6.37	9.34	3.46
4	8.33	9.07	3.1
5	10.14	8.08	1.69
6	11.94	9.04	1.45
7	15.29	9.16	1.32
8	18.93	8.97	1.29

The model is a simple 3-stage pipelined ALU, described in [6] and earlier work. The BDD variable ordering is that recommended in [4]. We do not have space to describe the model in detail. The properties we checked use fixed point computations and thus thoroughly exercise both implementations. All benchmarks were conducted in the same hardware and software environment.

Both programs were run for datapaths of one to eight bits, and address spaces of one to three bits (i.e. two to eight registers). An increase in either increases the branching complexity of the model, with increases in address space having a stronger effect. Although several properties can be checked for this ALU, the results shown are for the one which represents the worst (greatest) performance difference. Overhead costs can still be amortised over this because it is a template that needs to be separately checked for each bit on the datapath.

Due to the absence of any optimisations, the system began to thrash with an address space of four bits. Nevertheless, the table shows that the difference in performance closes as the branching complexity of the system increases and becomes acceptably small for the larger examples. This is because the most expensive BDD operation, the relational product, is hit particularly hard by any increase in branching complexity, whereas the corresponding operation on the term part of the judgement is trivial. This allows the theorem proving component of the program to “catch up” with the BDD component.

These numbers should be considered preliminary because we have as yet not implemented any of the standard optimisations such as partitioning the transition relation, iterative squaring, caching or exploiting non-alternating quantifiers [11]. All these would speed up the BDD component and increase the performance difference. On the other hand, our test bed is a toy example and the expectation is that even with all these optimisations, the theorem proving component, which does not scale as badly as BDDs for larger examples, will catch up when the program is run for harder examples.

6 Related Work

The system closest in spirit to our own is the HOL-Voss system [26]. Voss has a lazy functional language FL with BDDs as a built-in datatype. In [18] Voss was interfaced to HOL and verification using a combination of deduction and symbolic trajectory evaluation (STE) was demonstrated. Later work can be found in [1]. Recent developments have been outside the public domain after the developers moved to Intel.

Local model checkers have been implemented in a purely deductive fashion. This is possible because local model checking [?, 28, 32] does not require external oracles like BDD engines for efficiency. Thus it is difficult to directly compare this work with our own global model checker. In [2] a local model checking algorithm is given for L_μ . However monotonicity conditions for assertions are proved on-line rather than as a general theorem (e.g. Theorem 11) that can later be specialised. A deeper treatment for the less expressive logic CTL* can be found in [27]. This work also proves the proof system sound and complete using game-theoretic analysis.

An early example of combination of theorem provers and model checkers can be seen in [20]. Here the prover is used to split the proof into various sub-goals which are small enough to be verified by a separate model checker. There is

no actual integration so the translation between the languages of the theorem prover and the model checker is done manually.

Improved integrations of theorem provers with global model checkers typically enable the theorem prover to call upon the model checker as a black-box decision procedure given as an atomic proof rule [24, 22]. The prover translates expressions involving values over finite domains into purely propositional expressions that can be represented by BDDs. This allows use of the result as a theorem (as in our framework) but this method does not extend readily to the fully expansive approach. It thus achieves better efficiency at the expense of higher assurance of soundness.

Theorem provers have also been used to help with abstraction refinement [?, 10, 25] for model checking. Decision procedures in the theorem prover are used to discharge assumptions added to refine an abstraction that turns out to be too coarse and adds too much non-determinism to the system resulting in spurious counter-examples. Decision procedures for some subsets of first order logic have been used in automatic discovery of abstraction predicates [9] and invariant generation [23]. There is no technical obstacle to implementing these frameworks in our setting.

Well-known tools that implement some of the research sketched here include [16, ?, ?, ?].

7 Concluding Remarks

The implementation presented here does not contain non-trivial pieces of code whose soundness might be suspect: the core model checker is straight-forward. We expect this approach to pay off when we add optimisations and enhancements, particularly those that combine deductive and algorithmic verification.

The implementation itself is about 2000 lines of definitions and theorems about L_μ and sets, and about 500 lines of executable ML code. The proof of Theorem 11 is non-trivial, on account of having to do explicit α -conversion for the higher-order binders. The executable part is much easier to code, mainly because no soundness checks of the code are required: if the procedure terminates, the result is correct by construction.

Future work will focus on: providing error traces that can be manipulated in the theorem prover; leveraging the model checker for logics other than L_μ , justified by a semantics-based translation of the language into L_μ using the theorem prover; implementing more powerful versions of standard abstraction frameworks (e.g. [7, 8]) using the decision procedures and simplifiers provided by HOL.

A different aspect that we have not touched on is that this approach enables us to simultaneously get a handle on both the syntax and semantics of the derivation/evaluation⁸ at every step, via the representation judgement. We strongly expect that this will enable deeper investigation of methods combining proof-space search with state-space search.

⁸ But not in the sense of the Curry-Howard isomorphism.

References

1. M. D. Aagaard, R. B. Jones, and C-J. H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In Basant R. Chawla, Randal E. Bryant, Jan M. Rabaey, and M. J. Irwin, editors, *Design Automation Conference (DAC)*, pages 538–541. ACM/IEEE, ACM Press, 1998.
2. S. Agerholm and H. Skjodt. Automating a model checker for recursive modal assertions in HOL. Technical Report 92, Aarhus University, January 1990.
3. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
4. J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In A. Richard Newton, editor, *Proceedings of the ACM Design Automation Conference*. ACM, June 1991.
5. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
7. E. M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
8. E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. of Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 265–279. Springer, July 2002.
9. Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In Mark Aagaard and John W. O’Leary, editors, *Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*. Springer, November 2002.
10. J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Proceedings of the 1995 Workshop on Computer Aided Verification*, volume 939 of *LNCS*, pages 54–69. Springer, 1995.
11. E. A. Emerson and C-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *1st Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, 1986.
12. M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In G. Longo, editor, *Proceedings of the 14th Logic in Computer Science Conference*, pages 193–202. IEEE, IEEE Computer Society Press, 1999.
13. A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, volume 1125 of *LNCS*, pages 173–190. Springer, 1996.
14. M. J. C. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, August 2002.
15. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL : A theorem-proving environment for higher order logic*. Cambridge University Press, 1993.
16. The HOL-4 Proof Tool. Tool URL <http://hol.sf.net>, 2003.
17. G. J. Holzmann and D. Peled. The state of SPIN. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV'96: 8th International Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 385–389. Springer, 1996.

18. Jeffrey J. Joyce and Carl-Johan H. Seger. The HOL-Voss system : Model checking inside a general-purpose theorem prover. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*, pages 185–198. Springer, 1993.
19. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
20. R. P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In C. Courcoubetis, editor, *Proceedings of the 5th Workshop on Computer Aided Verification*, volume 697 of *LNCS*, pages 166–180. Springer, June 1993.
21. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
22. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Thomas A. Henzinger Rajeev Alur, editor, *CAV'96: 8th International Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 411–414. Springer, July 1996.
23. A. Pnueli, S. Rua, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, 2001.
24. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking and automated proof checking. In Pierre Wolper, editor, *Proceedings of Computer Aided Verification*, volume 939 of *LNCS*, pages 84–97. Springer-Verlag, 1995.
25. H. Saïdi. Model checking guided abstraction and analysis. In Jens Palsberg, editor, *Proceedings of the 7th International Static Analysis Symposium*, volume 1824 of *LNCS*, pages 377–396. Springer, July 2000.
26. C-J. H. Seger. Voss - a formal hardware verification system: User's guide. Technical Report UBC-TR-93-45, The University of British Columbia, December 1993.
27. C. Sprenger. *Deductive Local Model Checking*. PhD thesis, Computer Networking Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2000.
28. C. Stirling and D. J. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.
29. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
30. T. E. Uribe. Combinations of model checking and theorem proving. In Hélène Kirchner and Christophe Ringeissen, editors, *Proceedings of the Third Intl. Workshop on Frontiers of Combining Systems*, volume 1794 of *LNCS*, pages 151–170. Springer-Verlag, March 2000.
31. J. van Heijenoort, editor. *From Frege to Godel: A Source Book in Mathematical Logic, 1879-1931*. Harvard University Press, 1967.
32. G. Winskel. A note on model checking in the modal ν -calculus. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *Proceedings of the International Colloquium on Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 761 – 772. Springer, 1989.