Checking ACL2 Theorems via SAT Checking

Rob Sumners
robert.sumners@amd.com

Computer Engineering Research Center
The University of Texas at Austin
Austin, Texas, USA

Abstract

We present a procedure for checking a suitably-bounded ACL2 theorem using a SAT checker. The check is carried out by first translating the ACL2 theorem into an equivalent theorem defined with functions which only use the primitives `if`, `cons`, `car`, `cdr`, and `nil`. The translated theorem on simple cons-trees is then translated to a propositional formula by a form of evaluation lifted to a certain representations of functions on these simple cons-trees. The resulting propositional formula can be checked using a SAT checker. We present one such SAT checker also written in ACL2, but presumably any SAT checker could be used. In the case when the propositional formula fails, the propositional witness can easily be translated back to a witness of the failure for the original theorem.

1 Introduction

The process of proving theorems in ACL2 commonly involves iteratively breaking the verification task into smaller, more manageable pieces until the pieces are small enough that they are proven automatically. A successful proof that an ACL2 theorem is valid ensures that for every possible assignment of the free variables in the theorem, the expression defined by the theorem would evaluate to a non-`nil` value. When ACL2 is unable to prove a submitted theorem, then either the theorem is invalid or the proof requires further manual decomposition or guidance (assuming an ACL2 proof of the theorem exists). Unfortunately, it is not always clear which of these cases holds when ACL2 is unsuccessful and thus, a tool which could generate counterexamples to posed theorems would be useful.

One approach to searching for counterexamples is to simply construct assignments randomly (presumably with some intelligent bias) to the free variables in the theorem and evaluate the body of the theorem with these assignments until an assignment is found for which the theorem evaluates to `nil`. Beyond the obvious incompleteness of such a procedure, the main problem is the determination of the biasing for the random selection. An “intelligent” bias may require considerable deliberation on the part of the user and still may not cover some relevant cases. In addition, any particular bias scheme which is effective for testing a given theorem, is likely to be ineffective for other theorems which are encountered. To this end, we wish to define a tool for automatically checking the validity of theorems under suitable constraints of the free variables.

In the last decade or so, most of the active research in formal verification outside of the theorem proving community has centered around improving the efficacy of model checkers. Model checking commonly involves the checking of a temporal logic formula – intuitively, a formula which defines a set of legal behaviors – on a finite state model. The downside of model checking is the requirement that the model is reasonable in size (most problems do not afford reasonably-sized state models), but the upside of model checking is that it is essentially automatic and can find subtle errors quickly. In addition, when a property fails, often a counterexample can be generated, and sometimes a "minimal" counterexample can be generated. In order to extend the range of application for model checking, symbolic techniques are used whereby representations of sets of states are manipulated symbolically as opposed to the explicit enumeration of each state in the state model. Symbolic model checking[3, 4] was initially developed with Reduced Ordered Binary Decision Diagrams (BDDs)[2, 7] which are canonical representations of boolean functions. While the use of BDDs greatly extended the range of application for model checking, in many cases, the BDDs themselves would explode and render the model checking process ineffective due to resource exhaustion. In recent years, other symbolic approaches have been developed and one which has shown promise is the use of SAT checking procedures[1]. In this context, the model checking problem is bounded to allow the translation of the model check to a finite propositional formula, such that the original model check would pass if and only if the generated formula was propositionally valid. Thus, a SAT checker could subsequently be used to determine if the state model satisfied the property of interest. We take a roughly analogous path to the problem of checking ACL2 theorems.

Specifically, where the ACL2 user is attempting to prove the theorem `(thm <expr>)`, the user can "test" the theorem using the symbolic checker presented in this paper by submitting `(check-thm (implies <constraint> <expr>))`, where `<constraint>` is an expression which has the same free variables as `<expr>`, but suitably constrains the free variables so that the theorem can be translated into a propositional formula. It is sufficient if there are only a finite number of assignments to the free variables such that `<constraint>` evaluated to a non-`nil` value, but it is not necessary.

Assuming that we have a suitable `<constraint>` and that the symbolic checker has sufficient resources, the checker will then translate the problem into a propositional formula such that the resulting formula is valid if and only if there exists an assignment to the free variables such that `<constraint>` is non-`nil` while `<expr>` evaluates to `nil`. We compute this translation in two steps. We first translate the functions defining `<constraint>` and `<expr>` into functions defined on a reduced set of primitives. We then perform a symbolic evaluation of the translated definitions which results in the generation of the desired propositional formula. The resulting formula is then checked for validity and the theorem either passes, fails (with a counterexample translated to an assignment of the free variables in the theorem), or exceeds available resources.

In the remainder of this paper, we present the details of this theorem checker. In section 2 we present the translation from the function definitions on ACL2 objects to definitions on so-called simple-trees built from `cons` and `nil`. We then present in section 3 the symbolic evaluation of the simple-tree functions. In section 4 we detail the satisfiability checking procedure we have designed for use in the lifted evaluation process presented in section 3. We present some examples in section 5 and we conclude the paper with an outline for a proposed proof of correctness for the checker in section 6.

2 Translation of Theorems to Simple-Tree Definitions

The theorem checker we define in this paper takes a term defining a theorem and either returns `:qed`, a witness (i.e. an assignment to the free variables) demonstrating that the theorem is not valid, or a message declaring that the check was incomplete and some information to help the user diagnose why the search failed to complete. The theorem checker consists of three steps. The first step translates the ACL2 theorem and function definitions into definitions using only so-called simple-tree operations. The second step computes the representation of the function defined by the body of the theorem. A final check is performed to determine if the function maps any free variable assignments to `nil`. We first consider the first step of translating the definitions and theorems to simple-tree functions.

The ACL2 universe consists of a variety of objects including numbers (integers, rational, complex), characters, strings, symbols, conses, and everything else (the so-called "bad atoms"). ACL2 has numerous built-in operations and primitives for constructing and manipulating these objects. In order to factor the complexity of the translation from an ACL2 theorem to propositional formula, we define a simple intermediate sublanguage of ACL2 and translate the theorem to this sublanguage.

Before we define this intermediate language and this translation, we first need to clarify some terminology. For our purposes, a *history* $H(P)$ is well-formed sequence of `defun` and `mutual-recursion` forms where the bodies of the functions are built from functions defined (previously or immediately) in the history and the set of primitive ACL2 functions P. A sublanguage L of ACL2 is specified by a set of primitive functions P and a domain-of-interest D (defined by a recognizer predicate and it must include the constant `nil`) and is understood to consist of all functions defined in histories built from these primitives. Note that we do not support *encapsulate* or *defchoose* forms. In addition it is required that every primitive maps elements in the domain-of-interest to elements in the domain-of-interest. A *theorem* in a given sublanguage is simply a function (i.e. the "body" of the theorem) in this sublanguage, and a theorem is *valid* if the function never evaluates to `nil` for any binding of the parameters of the function to elements in the domain-of-interest.

A translation from one sublanguage L_1 to another L_2 is *generated* by a mapping M of the primitives P_1 to functions in L_2, where any history in L_1 is translated to a history in L_2 by replacing the primitives p in the L_1 function definitions with their mapped counterpart $M(p)$, and prepending the definitions of these $M(p)$ counterparts to the history. Such a translation is *validated* by the definition of a mapping F from D_1 to D_2 which maps `nil` (and only `nil`) to `nil`, and the proof that for every primitive p in P_1, for all x in D_1, $F(p(x)) = M(p)(F(x))$ – i.e. $M(p)$ is a homomorphism of p. It is then the case that if a valid translation of a theorem in $L1$ to $L2$ is valid in $L2$, then it is valid in $L1$ (i.e. the translation is sound). While it would be useful to require that F were onto to ensure completeness of the translation, we have found this to be impractical. Nonetheless, we have found that with the definition of a suitable "inverse" of F, failures of translated theorems are effectively translated back to failures of the original theorem.

For the first step of the theorem checker, we translate the original ACL2 theorem into the target intermediate sublanguage, ST is generated by the primitives {`cons`, `car`, `cdr`, `if`, `nil`} and has a domain-of-interest defined by the set of simple-trees which are recognized by the following predicate:

```
(defun simple-tree-p (x)
  (cond ((null x)
         t)
        ((consp x)
         (and (simple-tree-p (car x))
              (simple-tree-p (cdr x))))))
```

Preferably, we would like the source sublanguage of our translation to have a rich set of ACL2 primitives and have the domain-of-interest defined by the predicate `(lambda (x) t)`. Unfortunately, such a goal would (at the least) significantly complicate the translation to our target language ST and this may impact the efficiency of subsequent steps in the theorem checker. Fortunately, most ACL2 proof efforts (especially those which are likely targets for our theorem checker) do not require the full ACL2 universe. Indeed, most ACL2 proofs could be limited to a fixed number of symbols (including the booleans), the integers, and conses of these objects. For the purpose of this initial work, we assume our source sublanguage (i.e. the language in which the original theorem is written) is a simple "modeling" language *MDL* generated by the set of primitives {`if`, `car`, `cdr`, `cons`, `binary-+`, `n-`, `<`, `naturalp`, `symbolp`, `consp`, `equal`, `not`, `implies`, `iff`, `quote`}, where the only quoted objects are symbols from a fixed list of symbols `aux` – `aux` can be understood as a parameter to MDL. The functions `n-` and `naturalp` are the only non-standard ACL2 functions and are defined as follows:

```
(defun naturalp (x)
  (and (integerp x) (>= x 0)))

(defun n- (x y)
  (nfix (- (nfix x) (nfix y))))
```

The domain-of-interest is defined by the following recognizer predicate `good-model-object-p`:

```
(defun good-model-object-p (x aux)
  (cond ((symbolp x)
         (or (booleanp x)
             (member x aux)))
        ((integerp x)
         (>= x 0))
        ((consp x)
         (and (good-model-object-p (car x) aux)
              (good-model-object-p (cdr x) aux)))))
```

The parameter `aux` for the function `good-model-object-p` is a list of symbols defining a finite set of (non-boolean) symbols used in the set of MDL function definitions. This will be a fixed set of symbols because we do not include the `intern` operation (or its variants) in the modeling language. This means that the set of symbols encountered in any object created by the set of function definitions will be restricted to the set of symbols in the quoted constants in the (macro-expanded) function definitions and the booleans `t` and `nil`. This is important because instead of defining a mapping of each symbol to a simple-tree which must uniquely identify the symbol no matter what symbols might be constructed, we instead can map each symbol to a tree which distinguishes it only from the other symbols in the quoted constants and the booleans. The theorem checker actually computes this set of symbols automatically from the function definitions used to define the theorem to be checked.

We now need to define the mapping from the good-model-objects to simple-trees which validates our translation. In the language MDL, there are three "types" of objects, while in ST, there is only `cons` and `nil`. We therefore translate the MDL objects into a `cons` pair, where the `car` is a tag identifying the "type" of the object and the `cdr` is the simple-tree encoding of the value of the object. This leads to the definition of the function `mdl-to-tree`:

```
(defun nat-to-list (x)
  (if (zp x) nil (cons nil (nat-to-list (1- x)))))

(defun location (e x)
  (if (or (endp x) (equal e (first x))) 0 (1+ (location e (rest x)))))

(defun st-make-symb (x) (cons             nil           x))
(defun st-make-cons (x) (cons       (cons nil nil)      x))
(defun st-make-nat  (x) (cons (cons (cons nil nil) nil) x))

(defun mdl-to-tree (x aux)
  (cond ((null x) nil)
        ((consp x)
         (st-make-cons (mdl-to-tree (car x) aux)
                       (mdl-to-tree (cdr x) aux)))
        ((naturalp x)
         (st-make-nat (nat-to-list x)))
        ((and (symbolp x)
              (member x (cons t aux)))
         (st-make-symb (nat-to-list (location x (cons t aux)))))
        (t
         (er hard 'mdl-to-tree
             "illegal object in translation to simple-trees: ~x0"
             x))))
```

We also define an inverse mapping `tree-to-mdl` which is straightforward since `mdl-to-tree` is one-to-one. We use this inverse mapping to translate failure witnesses of the translated ST theorems back to failure witnesses of the original MDL theorem.

We note that the use of `nat-to-list` could lead to prohibitively expensive translations if the natural numbers encountered in the theorem checks were allowed to be "large". We could define a mapping of naturals which used the common binary encoding of numbers as sequences of booleans to achieve a logarithmic reduction in number representation, but for simplicity we decided to use the (potentially expensive) function `nat-to-list`.

We now must define the ST functions which implement the primitives in MDL. As an example, we provide the definition of the function `st-binary-+` which is the translation of the ACL2 operator `binary-+` – the operator `binary-+` commonly arises from the macro-expansion of terms involving `+` and `-`.

```
(defmacro if-cons (x y z) `(if (consp ,x) ,y ,z))

(defun st-coerce-to-nat (x)
  (if (st-naturalp x) x (st-make-nat nil)))

(defun st-binary-+valus (x y)
  (if-cons y (cons nil (st-binary-+valus x (cdr y))) x))

(defun st-binary-+ (x y)
  (let ((x (st-coerce-to-nat x))
        (y (st-coerce-to-nat y)))
    (st-make-nat (st-binary-+valus (cdr x) (cdr y)))))
```

The operator `if-cons` is an unfortunate hack. It is only allowed to appear in the ST functions implementing the MDL primitives on simple-trees. Since the operator `if-cons` can only appear in contexts where simple-trees are considered, it is equivalent to `if` in ST. But because ACL2 requires a termination measure to be shown on "all" ACL2 objects and not just some subset, the use of `if-cons` simplifies (and may be required for) the admission of ST functions into ACL2. We now wish to prove the theorem which demonstrates that the function `st-binary-+` is an accurate implementation of `binary-+` in ST:

```
(defthm st-binary-+-is-legal-implementation-of-binary-+
  (implies (and (good-model-object-p x aux)
                (good-model-object-p y aux))
           (equal (mdl-to-tree (binary-+ x y) aux)
                  (st-binary-+ (mdl-to-tree x aux)
                               (mdl-to-tree y aux)))))
```

In addition we require the proof that `binary-+` maps `good-model-objects` to `good-model-objects` in order to ensure that any composition of the MDL primitives will also have this property (which is required in order to relieve the `good-model-objects` hypothesis in the `legal-implementation` theorems).

We have defined a macro which takes a set of primitives defining an MDL language, a mapping of these primitives to ST operations, and the definitions of `mdl-to-tree`, `good-model-object-p`, `tree-to-mdl` and generates the the required lemmas. Assuming the required lemmas pass, a theorem checker is installed for the language built from these MDL primitives.

3 Symbolic Evaluation of Simple-Tree Theorems

We now present the second step of the theorem checker which (if it terminates) translates the theorem defined with ST functions into a propositional formula which preserves the validity of the ST theorem. This translation is the end-result of a procedure which computes a representation of the function computed by each subterm encountered in the expansion of the theorem body. This procedure is a special lifted form of evaluation which operates on representations of functions as opposed to the normal ACL2 evaluation process (the function `ev` in the ACL2 source file "translate.lisp") which operates on ACL2 objects. After computing the function representation for a theorem using this lifted evaluation, we can then easily extract the propositional formula which is satisfiable if and only if the original ST theorem can evaluate to `nil`.

This evaluation or computation of the function representation is carried out by first computing the function representations for the subterms and then composing these representations using the lifted versions of the ST primitives. We term these function representations *TFR*s which stands for "Tree Function Representation". These TFRs are essentially terms built from various ST primitives and variable references with some restrictions on how the terms can be constructed. Formally, the TFRs are recognized by the predicate `tfr-p` defined below.

```
(defun n-tuple (x n)
  (if (zp n) (null x) (and (consp x) (n-tuple (rest x) (1- n)))))

(defun tvp-p (x)  ;; TVP -- a tree variable position
  (or (var-symbolp x)       ;; a non-boolean symbol
      (and (n-tuple x 2)
           (member (first x) '(car cdr))
           (tvp-p (second x)))))

(defun ite-p (x)  ;; ITE -- an if-then-else propositional formula
  (or (booleanp x)
      (tvp-p x)
      (and (n-tuple x 4)
           (equal (first x) 'if)
           (ite-p (second x))
           (ite-p (third x))
           (ite-p (fourth x)))))

(defun tfr-p (x)  ;; TFR -- tree function representation
  (or (null x)
      (tvp-p x)
      (case (first x)
            (if (and (n-tuple x 4)
                     (ite-p (second x))
                     (tfr-p (third x))
                     (tfr-p (fourth x))))
            (cons (and (n-tuple x 3)
                       (tfr-p (second x))
                       (tfr-p (third x)))))))
```

A TFR is built recursively using other TFRs, propositional if-then-else formula (*ITE*s), and tree variable positions (*TVP*s). Intuitively, a TVP denotes a particular subtree of a free variable, an ITE defines a function which maps free variable assignments to booleans, and a TFR defines a function which maps free variable assignments to simple-trees. This intuition leads to the following definition of the semantics of a TFR; we define a function `interp-tfr` which takes a TFR and a variable assignment (defined by an `alist`) and returns a simple-tree.

```
(defun interp-tvp (tvp alist)
  (cond ((atom tvp)                ;; tvp is a variable symbol
         (cdr (assoc tvp alist)))
        ((equal (first tvp) 'car)
         (car (interp-tvp (second tvp) alist)))
        (t
         (cdr (interp-tvp (third tvp) alist)))))
```

```
(defun interp-ite (ite alist)
  (cond ((tvp-p ite)
         (if (interp-tvp ite alist) t nil))
        ((atom ite) ite)           ;; ite is either T or NIL
        (t
         (if (interp-ite (second ite) alist)
             (interp-ite (third ite) alist)
           (interp-ite (fourth ite) alist)))))

(defun interp-tfr (tfr alist)
  (cond ((tvp-p tfr) (interp-tvp tfr alist))
        ((atom tfr)  nil)            ;; tfr is NIL
        ((equal (first tfr) 'cons)
         (cons (interp-tfr (second tfr) alist)
               (interp-tfr (third tfr) alist)))
        (t
         (if (interp-ite (second tfr) alist)
             (interp-tfr (third tfr) alist)
           (interp-tfr (fourth tfr) alist)))))
```

In the actual code for the theorem checker, we optimize the structures for storing and manipulating TFRs, but functionally, they coincide with the structures recognized and interpreted by `tfr-p` and `interp-tfr` respectively. Our goal is to now define an "evaluator" which takes a term and computes a TFR equivalent to that term. The actual code for the evaluator in the theorem checker has been optimized to use indexes and arrays (in STOBJs) instead of lists in contexts where constant-time access and update were needed. For the purpose of presentation, we define a simplified (but representative) version of this evaluator in the function `tfr-eval` below.

```
(defun tfr-destruct (operator tfr)
  (cond ((tvp-p tfr)  (list operator tfr))
        ((atom tfr)   nil)         ;; tfr is NIL
        ((equal (first tfr) 'cons)
         (if (equal operator 'car) (second tfr) (third tfr)))
        (t (list 'if (second tfr)
                 (tfr-destruct operator (third tfr))
                 (tfr-destruct operator (fourth tfr))))))

(defun ite-extract (tfr)
  (cond ((tvp-p tfr)                tfr)
        ((atom tfr)                 nil)   ;; tfr is NIL
        ((equal (first tfr) 'cons)  t)
        (t (list 'if (second tfr)
                 (ite-extract (third tfr))
                 (ite-extract (fourth tfr))))))
```

```
(defun ite-not (x) (list 'if x nil t))

(program)
(mutual-recursion
(defun tfr-eval (term alist ctx fns)
  (if (variablep term)
      (let ((bound (assoc term alist)))
        (if bound (cdr bound) term)))
    (let ((operator (first term)))
      (case operator
            (quote nil)    ;; NIL is the only quoted object in ST terms.
            (cons (list 'cons (tfr-eval (second term) alist ctx fns)
                              (tfr-eval (third term) alist ctx fns)))
            ((car cdr) (tfr-destruct operator
                                     (tfr-eval (second term) alist ctx fns)))
            (if (let* ((tst (ite-extract (tfr-eval (second term) alist ctx fns)))
                       (t-ctx (ctx-and ctx tst))
                       (f-ctx (ctx-and ctx (ite-not tst))))
                  (cond ((ctx-empty f-ctx)
                         (tfr-eval (third term) alist t-ctx fns))
                        ((ctx-empty t-ctx)
                         (tfr-eval (fourth term) alist f-ctx fns))
                        (t
                         (list 'if tst
                               (tfr-eval (third term) alist t-ctx fns)
                               (tfr-eval (fourth term) alist f-ctx fns))))))
            (otherwise
             (mv-let (formals body)
                 (if (flambdap operator)
                     (mv (lambda-formals operator) (lambda-body operator))
                   (lookup-function operator fns))
               (tfr-eval body (tfr-eval-bind formals (rest term) alist ctx fns)
                         ctx fns)))))))

(defun tfr-eval-bind (formals actuals alist ctx fns)
  (if (endp formals) ()
    (cons (cons (first formals)
                (tfr-eval (first actuals) alist ctx fns))
          (tfr-eval-bind (rest formals) (rest actuals)
                         alist ctx fns))))
) ;; end mutual-recursion defining tfr-eval and tfr-eval-bind
```

The function `tfr-eval` takes a `term`, an `alist` binding the variables in `term` to TFRs, a `ctx` (which we will explain shortly), and a list `fns` of function definitions. Every operation encountered in any term should either be a `lambda` form, one of the ST primitives or one of the functions defined in `fns`. The translation function is named `tfr-eval` because it operates in a manner similar to a simple LISP evaluator. The main differences are that the evaluation is carried out on representations of functions instead of objects, and that the evaluation of `if` may induce an evaluation of both the true branch and the false branch. The function `tfr-destruct` distributes `car` and `cdr` operations through a TFR structure in a manner consistent with the properties of `car` and `cdr`. The function `ite-extract` takes a TFR and generates an ITE such that the result of interpreting the TFR and the ITE are `iff` equivalent. The following theorems declare the desired properties of `tfr-destruct` and `ite-extract`.

```
(defthm tfr-destruct-desired-property-for-car  ;; similar theorem for 'cdr
  (equal (interp-tfr (tfr-destruct 'car tfr) alist)
         (car (interp-tfr tfr alist))))

(defthm ite-extract-desired-property
  (iff (interp-ite (ite-extract tfr) alist)
       (interp-tfr tfr alist)))
```

The parameter `ctx` to `tfr-eval` is used to pass information about the context in which a term is being evaluated. At the very least, this information will be needed to ensure that certain `if` branches are not taken, but this information could be used for other purposes. The function `ctx-and` takes an existing `ctx` and an ITE `<x>` and creates a new context consisting of the information in `ctx` and any additional information which can be ascertained from the assertion that `<x>` is equal to `t`. The function `ctx-empty` checks if the information in a given `ctx` is inconsistent.

We purposely have not specified what exactly constitutes a "context" since there are some tradeoffs involved. On one extreme, a "context" could simply be an ITE and the function `ctx-empty` could be a full-blown SAT check to determine if a context is satisfiable. Unfortunately, this could add significant expense to the evaluation of `tfr-eval` and in our initial experiments, we have found that this is unnecessary. In our current implementation of `tfr-eval`, a context is simply an alist binding TVPs to either `t` or `nil`. The function `ctx-empty` simply checks to see if a TVP has been bound to both `t` and `nil`. The function `ctx-and` traverses the ITE parameter `<x>` and adds any TVP bindings which it can deduce must be true in the case where `<x>` is equal to `t`.

In the actual code of the theorem checker, we handle contexts in a similar, but more aggressive manner. In particular, we pass the context in as a parameter to `ite-extract` and reduce any TVPs encountered during the extraction which are bound in the context. In addition, in the actual code for `ite-extract`, we simplify the ITEs which are constructed using various simplification rules for `if` terms (e.g. `(thm (equal (if x y y) y))`).

Note, that in order to admit `tfr-eval` to the logic, we will need to add a `stack-depth` parameter which bounds the number of nested function expansions. This will also require `tfr-eval` to return a flag denoting that the TFR returned is not a complete translation of the term provided. This extra `stack-depth` parameter is also useful in practice since the user can bound the search for failures which may be found even if the TSR returned is incomplete.

We finish this section with the definition of the top-level `check-thm` form which is defined as a macro below. The function `sat-check` is the main interface to the SAT checker. This function takes an ITE and either returns `:unsatisfiable` if no valuation to the TVPs could be found for which the ITE evaluates to `t`, and otherwise it returns such a valuation to the TVPs. If a satisfying valuation or witness is returned, then we translate this witness back to an alist of MDL objects by first translating the valuation to the TVPs into a binding of the free var.s in `thm` to simple trees. We then use `tree-to-mdl` to translate each value in this alist to an MDL object.

```
(defun check-thm-fn (thm state)
  (declare (xargs :stobjs state))
  (let* ((fns (assemble-ST-functions thm state))
         (check (sat-check
                 (ite-not
                  (ite-extract
                   (tfr-eval thm       ;; the term defining the body of the theorem
                             nil       ;; initial alist, no bound var.s
                             nil       ;; initial context, no TVPs are bound
                             fns)))))) ;; list of function definitions translated to ST
   (if (eq check :unsatisfiable)
       :qed
     (tree-alist-to-mdl-alist
      (sat-witness-to-tree-alist check)))))

(defmacro check-thm (thm)
  `(check-thm-fn (quote ,thm) state))
```

In the actual code for the theorem checker, we add a final step where we evaluate the term `thm` on the failure witness to double check that the witness is indeed a failure witness for the original `thm`. This is done to both double check the SAT checker results, and in the case where `tfr-eval` was bounded by the `stack-depth` parameter, the resulting witness may still be a failure witness.

4 Satisfiability Checking of If-Then-Else Forms

We briefly outline in this section a SAT checker which we designed for efficient propositional satisfiability checks of formula built from if-then-else operations. The complete details of this SAT checker are beyond the scope of this paper since the focus of the paper is to translate a (suitably-bounded) ACL2 theorem into a propositional formula. We do present some points of emphasis in the design of the SAT checker which arose from the particular needs of this application.

In essence, our SAT checker is a version of the Davis-Putnam procedure (this is a common approach to SAT checking [8, 5, 6]). A Davis-Putnam SAT checker searches for a satisfying assignment by constantly pushing and popping assignments of propositional variables onto a so-called *decision stack*. At all times during a Davis-Putnam procedure, the decision stack defines a partial assignment of the propositional variables and the goal is to find an extension of this partial assignment to a satisfying assignment. The basic Davis-Putnam procedure is defined by the function `ite-find-sat` which attempts to find a satisfying assignment for an ITE formula. The function `push-next-decision` simply selects a currently unassigned variable and pushes it onto the decision stack. The function `search-next-decision` is the main engine of a Davis-Putnam procedure. `Search-next-decision` first calls `propagate-decisions` which simplifies the current `ite` formula under the partial assignment on the decision stack. If the formula simplifies to `nil`, then we must resolve the conflict by retracting the stack to the last assignment which has not been flipped and flipping this assignment on the decision stack and continuing with the search. In addition, `propagate-decisions` will push any additional variable assignments which are deduced from the current context.

```
(defun search-next-decision (ite stack)
  (mv-let (conflict stack) (propagate-decisions ite stack)
    (if (not conflict) (mv t stack)
      (mv-let (resolved stack) (resolve-conflict stack)
        (if (not resolved) (mv nil stack)
          (search-next-decision ite stack))))))

(defun ite-find-sat (ite stack)
  (mv-let (found stack) (push-next-decision ite stack)
    (if (not found) (extract-witness stack)
      (mv-let (found stack) (search-next-decision ite stack)
        (if (not found) :unsatisfiable
          (ite-find-sat ite stack))))))

(defun sat-check (ite) (ite-find-sat ite ()))
```

As with the evaluator in the previous section, the actual code for our implementation of the SAT checker uses arrays (in STOBJs) and natural number (fixnum) indexes when constant-time access and update are needed for reasons of efficiency. In addition, for reasons of efficiency, there is an initial simplification pass of the ITE before it is even passed to the SAT checker. This initial pass is needed to reduce any TVPs in the ITE which we can deduce must be `t` or `nil`.

In a propositional formula, every variable is independently assigned to `t` or `nil`. The variables in our ITE formula are actually TVPs and are not independent. In particular, if a certain TVP `x` is bound to `nil`, then the TVPs `(car x)` and `(cdr x)` must also be `nil` – the contrapositive of this implication must also be handled. We handle this dependence between TVPs by (a) adding any additional implied bindings to decision stack whenever a variable is pushed onto the decision stack, and (b) extending the deduction of new variable assignments in `propagate-decisions` to take into account the dependence between TVPs.

Unfortunately, external SAT checkers like SATO, GRASP, or CHAFF are designed for independent propositional variables and thus the implications between the TVPs would have to be exported as additions clauses in the exported formula.

5 Examples

We now present two example uses of the theorem checker in order to demonstrate the range of application and to provide an intuition on what constitutes a suitably-bounded theorem. We begin with the definition of a simple state model implementing mutual exclusion comprised of a collection of processes which take turns setting a shared flag variable. This state model is defined by the next-state function `next` provided below.

```
(defun step-state (s f)
  (case s
        (try       (if f 'try 'go))
        (go        'wait)
        (otherwise 'try)))
```

```
(defun step-flag (s f)
  (case s
        (try       t)
        (go        nil)
        (otherwise f)))

(defun 0p (n)
  (not (and (naturalp n) (> n 0))))

(defun get-nth (n l)
  (if (0p n) (car l)
    (get-nth (n- n 1) (cdr l))))

(defun set-nth (n x l)
  (if (0p n) (cons x (cdr l))
    (cons (car l)
          (set-nth (n- n 1) x (cdr l)))))

(defun next (l n)
  (let ((f (car l))
        (s (get-nth n (cdr l))))
    (cons (step-flag s f)
          (set-nth n (step-state s f) (cdr l)))))
```

The function `next` takes a state `l` and an input `n` and returns the appropriate next state. The `(car l)` is the shared flag variable which is set by a process which is in the state `'try` and is entering the `'go` state, to keep any other states from entering the `'go` state until it has left the `'go` state. The states of the processes are stored in `(cdr l)` and the input `n` selects which process takes the next step. The goal is to prove that the processes are mutually excluded (i.e. at most one process is in the `'go` state), and we do this by proving that the following predicate `good` is an invariant.

```
(defun no-one-go (l)
  (if (endp l) t
    (and (not (equal (car l) 'go))
         (no-one-go (cdr l)))))

(defun only-one-go (l)
  (and (consp l)
       (if (equal (car l) 'go)
           (no-one-go (cdr l))
         (only-one-go (cdr l)))))

(defun good (l)
  (if (car l)
      (only-one-go (cdr l))
    (no-one-go (cdr l))))
```

The predicate `good` is verified to be an invariant by proving the following theorem: `(thm (implies (good l) (good (next l n))))`. If we simply passed this invariant check to the theorem checker by submitting the form `(check-thm (implies (good l) (good (next l n))))`, the theorem checker will either terminate with `stack-depth` exhaustion or resource exhaustion. This is because the check is not suitably bounded since the recursive expansion of `get-nth`, `set-nth`, `no-one-go`, and `only-one-go` will not terminate. We can bound the theorem by restricting the length of `l` which in effect bounds the number of processes. The following `check-thm` will then succeed:

```
(defun bounded-listp (l n)
  (if (0p n) (not l)
    (and (consp l)
         (bounded-listp (cdr l) (n- n 1)))))

(check-thm
 (implies (bounded-listp l 4)
          (implies (good l)
                   (good (next l n)))))
```

Note that this check succeeds even though there are an infinite set of assignments to `l` which satisfy `(bounded-listp l 4)`. Additional constraints on `n` and `l` are still likely to be useful in order to restrict the selection of failure witnesses. Of course, adding constraints may also eliminate potential counterexamples to the original theorem.

While the analysis of state models is likely to be a common application of the theorem checker, the checker can also be used to check theorems about any MDL function. As an example, we can check that the function `qsort` defined below returns sorted lists (for suitably-bounded lists).

```
(defun upper-part (x s)
  (if (endp x) ()
    (let ((e (first x)))
      (if (< e s)
          (upper-part (rest x) s)
        (cons e (upper-part (rest x) s))))))

(defun lower-part (x s)
  (if (endp x) ()
    (let ((e (first x)))
      (if (< e s)
          (cons e (lower-part (rest x) s))
        (lower-part (rest x) s)))))

(defun qsort (x)
  (if (endp (rest x)) x
    (let ((f (first x))
          (r (rest x)))
      (append (qsort (lower-part r f))
              (list f)
              (qsort (upper-part r f))))))

(defun sortedp (x)
  (or (endp (rest x))
      (and (<= (first x) (second x))
           (sortedp (rest x)))))
```

```
(defun bounded-elementsp (x n)
  (or (endp x)
      (and (naturalp n)
           (< (first x) n)
           (bounded-elementsp (rest x) n))))

(check-thm
 (implies (and (bounded-listp x 3)
               (bounded-elementsp x 2))
          (sortedp (qsort x))))
```

These example `check-thm` applications complete in a few seconds. For more realistic examples, the current revision of the theorem checker is inadequate. The main problem is the inefficiency of our SAT checker. We are currently addressing this problem and also working on extensions which would allow the use of existing SAT checkers (like SATO, CHAFF, or GRASP) in its place as suitable replacements. In many of these cases, an additional translation layer will be needed in order to translate from the `if-then-else` format which is currently employed to a (for example) Conjunctive Normal Form. This translation will often introduce additional auxiliary propositional variables into the formula.

We also wish to point out that another source of inefficiency is due to the general nature of the translation. A translation defined to handle a particular class of definitions would likely be able to make use of knowledge of the domain of application to optimize the translation to propositional formula in order to reduce the number of propositional variables and ITE nodes introduced. For example, an efficient translation of the function `next` would introduced a single propositional variable for the flag and two variables to encode each of the states. In our translation, though, we introduced several additional variables for tags, building the list of `cons`es, and the encoding of the symbols. We currently don't believe that this source of inefficiency is the main roadblock to the effective use of the checker.

We also believe that the benefits of the simple elegant definition and general application of the checker will outweigh the inefficiency of the translation in the long run. Further analysis is needed to determine the validity of these beliefs.

6 Proposed Correctness Proof and Future Work

A potential benefit in defining our theorem checker completely in ACL2 is the ability to prove that the theorem checker is correct. In addition, proving the correctness of the theorem checker may allow us to install the checker as a `:meta` rule which would allow ACL2 to install the theorem checker into the simplification process. This would in turn allow theorems which passed the theorem checker to be installed into the ACL2 database of rules. For instance, let's assume for the moment that the macro `check-thm` was a function which took a term as an input and returned a non-`nil` value if and only if the theorem could be translated to a propositional check and the subsequent check passed. The checker could presumably be installed as a metafunction with the following theorem:[1]

[1] This metafunction discussion is provided for presentation. In reality, additional requirements would need to be enforced. For instance, the `good-mdl-object-p` hypothesis would need to be verified before the metafunction could be applied.

```
(defun meta-check (term) (if (check-thm term) t term))

(defthm check-thm-is-metafunction
  (iff (evl term alist)
       (evl (meta-check term) alist)))
```

Where the function `evl` was generated by the `defevaluator` form. Unfortunately, the evaluator `evl` must be restricted to terms built from a fixed set of operations where the theorem checker we have defined takes the function definitions as an extra input (extracted from the `state` parameter). So our proof goal is to demonstrate that for any set of function definitions, that the checker only passes theorems. We could in addition set out to prove that when the checker returns a failure, that the theorem is invalid. But, since the checker generates a witness and it is easy to double-check this witness by evaluating the body of the theorem using the witness, the need for such a proof is reduced considerably. Thus our proof is to derive the following theorem which states that for any `alist` binding the free variables in `term`, the result of evaluating the `term` with `alist` will result in a non-`nil` value if the result of `(check-thm term)` was the keyword `:qed`.

```
(defun good-mdl-object-p-alist (alist aux)
  (or (endp alist)
      (and (good-mdl-object-p (cdar alist) aux)
           (good-mdl-object-p-alist (cdr alist) aux))))

(defthm theorem-checker-is-correct
  (let* ((fns (assemble-MDL-functions term state))
         (aux (quoted-symbols-in-fns fns)))
    (implies (and (good-mdl-object-alist-p alist)
                  (equal (check-thm term) :qed))
             (mdl-eval term alist fns))))
```

The function `mdl-eval` is a simple evaluator with built-in support for the MDL primitives. The proof of this theorem will rely on the proof of two main lemmas corresponding to the decomposition of the theorem checker into the two main pieces presented earlier. We first need to demonstrate that the translation to ST preserves the MDL definitions. We demonstrate this by showing that `mdl-eval` can be "simulated" by the corresponding evaluator for ST:

```
(defun mdl-alist-to-st-alist (alist aux)
  (if (endp alist) ()
    (cons (cons (caar alist) (mdl-to-tree (cdar alist) aux))
          (mdl-alist-to-st-alist (cdr alist) aux))))

(defthm st-evaluation-simulates-mdl-evaluation
  (let* ((mdl-fns (assemble-MDL-functions term state))
         (st-fns (assemble-ST-functions term state))
         (aux (quoted-symbols-in-fns mdl-fns)))
    (implies (good-mdl-object-alist-p alist)
             (equal (st-eval term (mdl-alist-to-st-alist alist aux) st-fns)
                    (mdl-to-tree (mdl-eval term alist mdl-fns) aux)))))
```

This theorem (along with the one-to-one property of `mdl-to-tree`) will allow us to translate the verification task to the proof of correctness of the symbolic evaluation on ST terms. This correctness is captured by the following theorem:

```
(defun interp-tfr-alist (tfr-alist alist)
  (if (endp tfr-alist) ()
    (cons (cons (caar tfr-alist) (interp-tfr (cdar tfr-alist) alist))
          (interp-tfr-alist (cdr tfr-alist) alist))))

(defthm symbolic-evaluation-of-st-correct
  (implies (interp-ite ctx alist)
           (equal (interp-tfr (tfr-eval term tfr-alist ctx fns) alist)
                  (st-eval term (interp-tfr-alist tfr-alist alist) fns))))
```

This theorem would ensure that the TFR returned by `tfr-eval` defines a function which is consistent with the evaluation of the term which built the TFR. In accordance with our previous point about to the need of a `stack-depth` parameter, the above theorem would need to be conditioned on the effective completion of the function `tfr-eval`. When we combine the above property with some additional properties of the TFRs which are returned from `tfr-eval`, we can obtain our desired goal. The proof of this theorem will of course rely on the verification of the SAT checker, but at least the correctness proof of the SAT checker should be easily separated and encapsulated.

Of course, the fact that this proof has not been carried out at the time of writing for this paper almost ensures that the completed proof will be different in many relevant facets. Nonetheless, we included this sketch to provide a picture of the potential for verifying this checker in ACL2.

We conclude the paper by reiterating that the work presented in this paper is still a work in progress. The current version of the checker can be used to check "smaller" theorems, but needs some significant optimizations in order to scale to the theorems we wish to target. We also plan to investigate the integration of external SAT checkers using the `sys-call` function introduced in ACL2 v2-6. Another area of further investigation is an interface for allowing the user to define a default ordering on the decision variables for the subsequent SAT check. While the problem of determining a good ordering is expensive in the general case, for a particular class of problems which the user encounters, certain heuristic choices for ordering the decisions may prove very effective (e.g. splitting on program location or control variables before data variables). Assuming the desired efficiency goals are achieved, the theorem checker previewed in this paper should be an effective tool for the ACL2 user either in reducing the amount of time spent proving invalid theorems or simplifying the task of proving suitably-bounded theorems.

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. in Tools and Algorithms for Construction and Analysis of Systems, 1999.

[2] K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD package, *Proceedings of the ACM/IEEE Design Automation Conference*, 1990.

[3] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} states and beyond," in *IEEE Symposium on Logic in Computer Science*, 1988.

[4] P. Manolios. Mu-calculus Model Checking, in *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, June, 2000.

[5] J. Marques-Silva and K. Sukallah. GRASP: A Search Algorithm for Propositional Satisfiability." IEEE Transactions on Computers, vol. 48, 1999.

[6] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. in 39th Design Automation Conference, June, 2001.

[7] R. Sumners. Correctness Proof of a BDD Manager in the Context of Satisfiability Checking. in ACL2 Workshop 2000, URL – http://www.cs.utexas.edu/users/moore/acl2/workshop-2000.

[8] H. Zhang. SATO, An Efficient Propositional Prover. in International Conference on Automated Deduction, 1997.