7

Mu-Calculus Model-Checking

Panagiotis Manolios
Department of Computer Sciences, University of Texas at Austin, Texas
Email: pete@cs.utexas.edu

Abstract

Temporal logic model-checking has received substantial academic interest and has enjoyed wide industrial acceptance. Temporal logics are used to describe the behavior (over time) of systems which continuously interact with their environment. Model-checking algorithms are used to decide if a finite-state system satisfies a temporal logic formula. Many temporal logics, *e.g.*, *CTL*, *LTL*, and CTL^* can be translated into the Mu-Calculus. In addition, the algorithm that decides the Mu-Calculus is used for symbolic (BDD-based) model-checking, a technique that has greatly extended the applicability of model-checking. In this case study we define a model-checker for the Mu-Calculus in ACL2 and show how to translate *CTL* into the Mu-Calculus.

In the process of defining the Mu-Calculus, we develop (ACL2) books on set theory, fixpoint theory, and relation theory. The development of these books is given as a sequence of exercises. These exercises make use of varied ACL2 features; therefore, the first few sections may be of interest to readers who want more practice in proving theorems in ACL2.

Introduction

Machine-checked proofs are increasingly being used to cope with the complexity of current hardware and software designs: such designs are too complicated to be checked by hand and machine-checked proofs are a reliable way to ensure correctness. *Reactive systems* are systems with nonterminating or concurrent behavior. Such systems are especially difficult to design and verify. Temporal logic was proposed as a formalism for specifying the correctness of reactive systems in [87]. Algorithms that decide if a finite-state system satisfies its specification are known as *model-checking* algorithms [20, 27, 89]. Model-checking has been successfully applied to automatically verify many reactive systems and is now being used by hardware companies as part of their verification process. In this chapter, we develop a model-checker for the propositional Mu-Calculus [64, 30, 32, 31, 29, 85]—a calculus that subsumes the temporal logics CTL, LTL, and CTL^*—in ACL2.

This chapter is intended as a bridge between the companion book, *Computer-Aided Reasoning: An Approach* [58], and the other case studies. There are several self-contained sections in which the reader is presented with exercises whose solutions lead to books on set theory, fixpoint theory, and relation theory. We expect that the exercises in these sections are at the right level of difficulty for readers who have read the companion book. These exercises make use of diverse, less elementary features of ACL2 such as congruence-based reasoning, refinements, packages, the use of macros, guard verification, encapsulation, mutual recursion, and functional instantiation. We also discuss *compositional reasoning*; specifically we show how to reason about efficient implementations of functions by using rewrite rules that transform the efficient functions into other functions that are easier to reason about. Therefore, we expect—at least the first part of—this chapter to be of general interest.

If you are not interested in developing the required set theoretic results, but are interested in formalizing the Mu-Calculus in ACL2, then, instead of solving the exercises on your own, download the appropriate books from the supporting material for this chapter.

This chapter is organized as follows: the next three sections develop the set theory, fixpoint theory, and relation theory discussed above. In the three sections after that, we present the notion of a model, the syntax and semantics of the Mu-Calculus, and proofs that the fixpoint operators of the Mu-Calculus actually compute fixpoints. A section on the temporal logic CTL and its relation to the Mu-Calculus follows. We conclude with some directions for further exploration.

Conventions on Exercises

Whenever we introduce a function or ask you to define one, admit it and add and verify guards; this is an implicit exercise. Many exercises consist solely of a term or an event; interpret this as a command to prove that the term is a theorem or to admit the event. The supporting material includes a macro that you may find useful for dealing with guards. The file `solutions/defung-intro.txt` describes the macro and contains exercises.

7.1 Set Theory

In this section, we develop some set theory. We represent sets as lists and define an equivalence relation on lists that corresponds to set equality. It turns out that we do not have to develop a "general" theory of sets; a theory of *flat* sets, *i.e.*, sets whose elements are compared by `equal`, will do. For example, in our theory of sets, `'(1 2)` is set equal to `'(2 1)`, but `'((1 2))` is not set equal to `'((2 1))`.

We develop some of the set theory in the package `SETS` (see `defpkg`) and the rest in the package `FAST-SETS`, in subsections labeled by the package names. When using packages, we define constants that contain all of the symbols to be imported into the package. We start by guessing which symbols will be useful. For example, we import `len` because we need it to define the cardinality of a set and we import the symbols `x` and `x-equiv`; otherwise, when using `defcong`, `x-equiv` prints as `ACL2::x-equiv`, which strains the eye. As we develop the book, we notice that we forgot a few symbols and add them.[1]

7.1.1 SETS

Here is how we define the package `SETS`.

```
(defconst *export-symbols*
  (union-eq *acl2-exports*
       (union-eq
        '(len  ...  *export-symbols*)
        *common-lisp-symbols-from-main-lisp-package*)))

(defconst *sets-symbols* (union-eq *export-symbols*  ...  ))

(defpkg "SETS" *sets-symbols*)
```

We use the simplest definitions that we can think of so that it is easy to prove theorems. Later, we define functions that are more efficient and prove the rewrite rules that allow us to rewrite the efficient functions into the simpler ones. In this way, once rewritten, all the theorem proving is about the simple functions, but the execution uses the efficient versions.

The definitions of `in` (set membership), `=<` (subset), and `==` (set equality) follow.

```
(defun in (a X)
  (cond ((endp X) nil)
        ((equal a (car X)) t)
        (t (in a (cdr X)))))
```

[1] Due to some technical issues (see `package-reincarnation-import-restrictions`), this unfortunately means that we have to start a new ACL2 session.

```
(defun =< (X Y)
  (cond ((endp X) t)
        (t (and (in (car X) Y)
                (=< (cdr X) Y)))))

(defun == (X Y)
  (and (=< X Y)
       (=< Y X)))
```

Notice that `==` is an *equivalence relation*: it is reflexive, symmetric, and transitive. The macro `defequiv` can be used to show that a relation is an equivalence relation. Use `:trans1` to print out the translation of the `defequiv` form in the exercise below before you do it.

Exercise 7.1 `(defequiv ==)`

We make heavy use of congruence-based reasoning and will therefore discuss the topic briefly. For a full explanation consult the companion book [58] and the documentation on `equivalence`, `defequiv`, and `congruence`. Congruence-based reasoning can be seen as an extension of the substitution of equals for equals, where arbitrary equivalence relations can be used instead of equality. We motivate the need for congruence-based reasoning with an example using the equivalence relation `==`.

Consider the function `set-union` which computes the union of two sets. This function is defined below and is equivalent to `append`. We might want to prove

```
(implies (== X Z)
         (equal (set-union X Y) (set-union Z Y)))
```

so that ACL2 can replace x by z in `(set-union` x y`)`, if it can establish `(==` x z`)`. Letting x be `(1 1)` and z be `(1)`, it is easy to see that this is not a theorem. However, the following is a theorem.

```
(implies (== X Z)
         (== (set-union X Y) (set-union Z Y)))
```

If stored as a congruence rule (see `congruence` and `rule-classes`), ACL2 can use this theorem to substitute z for (a set equal) x in `(set-union` x y`)`, in a context where it is enough to preserve `==`. More generally, a theorem of the form:

```
(implies (eq1 X Z)
         (eq2 (foo ... X ...)
              (foo ... Z ...)))
```

where `eq1` and `eq2` are known equivalence relations can be made a congruence rule. Such a rule allows us to replace x by z in `(foo ...` x `...)` if x and z are `eq1`-equal and we are in a context where it is enough to preserve `eq2`. This should make it clear why congruence-based reasoning is a generalization of the substitution of equals for equals.

The macro <u>`defcong`</u> can be used to prove congruence rules. Use <u>`:trans1`</u> to print out the translation of the `defcong` forms in the exercise below before you do it.

Exercise 7.2

1. `(defcong == equal (in a X) 2)`
2. `(defcong == equal (=< X Y) 1)`
3. `(defcong == equal (=< X Y) 2)`
4. `(defcong == == (cons a X) 2)`

We now give the definition of `set-union`.

```
(defun set-union (X Y)
  (if (endp X)
      Y
    (cons (car X) (set-union (cdr X) Y))))
```

Exercise 7.3

1. `(equal (in a (set-union X Y)) (or (in a X) (in a Y)))`
2. `(=< X (set-union Y X))`
3. `(== (set-union X Y) (set-union Y X))`
4. `(equal (== (set-union X Y) Y) (=< X Y))`
5. `(defcong == == (set-union X Y) 1)`
6. `(equal (=< (set-union Y Z) X) (and (=< Y X) (=< Z X)))`

The definition of `intersect`, a function which computes the intersection of two sets, follows.

```
(defun intersect (X Y)
  (cond ((endp X) nil)
        ((in (car X) Y)
         (cons (car X) (intersect (cdr X) Y)))
        (t (intersect (cdr X) Y))))
```

Exercise 7.4

1. `(equal (in a (intersect X Y)) (and (in a X) (in a Y)))`
2. `(== (intersect X Y) (intersect Y X))`

3. `(implies (=< X Y) (== (intersect X Y) X))`

4.
```
(implies (or (=< Y X) (=< Z X))
         (=< (intersect Y Z) X))
```

The definition of `minus`, a function which computes the set difference of two sets, follows.

```
(defun minus (X Y)
  (cond ((endp X) nil)
        ((in (car X) Y)
         (minus (cdr X) Y))
        (t (cons (car X) (minus (cdr X) Y)))))
```

Exercise 7.5

1. `(implies (=< X Y) (equal (minus X Y) nil))`

2. `(implies (=< X Y) (=< (minus X Z) Y))`

The functions `set-complement`, `remove-dups`, `cardinality`, and `s<` (strict subset) are defined below.

```
(defun set-complement (X U) (minus U X))

(defun remove-dups (X)
  (cond ((endp X) nil)
        ((in (car X) (cdr X))
         (remove-dups (cdr X)))
        (t (cons (car X)
                 (remove-dups (cdr X))))))

(defun cardinality (X) (len (remove-dups X)))

(defun s< (X Y) (and (=< X Y) (not (=< Y X))))
```

Exercise 7.6 *Define* `perm`*, a function of two arguments that returns* `t` *if its arguments are permutations and* `nil` *otherwise. Prove* `(defequiv perm)` *and* `(defrefinement perm ==)`*. (*`Perm` *is defined in the companion book [58].)*

Exercise 7.7
```
(implies (s< X Y)
         (< (len (remove-dups X)) (len (remove-dups Y))))
```

7.1.2 FAST-SETS

Although the definitions of the basic set operations defined above are good for reasoning about sets, some are not appropriate for execution. For example, `set-union` is not tail-recursive[2], hence, even if compiled, we can easily get stack overflows. In this section, we define functions that are more appropriate for execution and prove rewrite rules that transform the new, efficient versions to the old, simpler versions in the appropriate context (specifically, when it is enough to preserve ==). This approach is *compositional*, *i.e.*, it allows us to decompose proof obligations of a system into proof obligations of the components of the system. Compositional reasoning is routinely used by ACL2 experts and is essential to the success of large verification efforts.

The functions we define below have the same names as their analogues, but are in the package `FAST-SETS`. `FAST-SETS` imports symbols from `SETS`, *e.g.*, == (we expect this to be clear from the context, but one can consult the supporting material for the package definition, if required). The definition of `set-union`, in the package `FAST-SETS`, follows.

```
(defun set-union (X Y)
  (cond ((endp X) Y)
        ((in (car X) Y)
         (set-union (cdr X) Y))
        (t (set-union (cdr X) (cons (car X) Y)))))
```

Exercise 7.8 `(== (set-union X Y) (sets::set-union X Y))`

Recall that the above rule allows ACL2 to replace occurrences of `set-union` by `sets::set-union` in a context where it is enough to preserve ==.

The definition of `intersect` follows. Note that its auxiliary function is tail recursive.

```
(defun intersect-aux (X Y Z)
  (cond ((endp X) Z)
        ((in (car X) Y)
         (intersect-aux (cdr X) Y (cons (car X) Z)))
        (t (intersect-aux (cdr X) Y Z))))

(defun intersect (X Y) (intersect-aux X Y nil))
```

Exercise 7.9 `(== (intersect X Y) (sets::intersect X Y))`

Exercise 7.10 *Define* `minus`*, a tail-recursive version of* `sets::minus`*, and prove* `(== (minus X Y) (sets::minus X Y))`*.*

[2]See the companion book [58] for a discussion of tail recursion and for example proofs.

Alternate definitions of `remove-dups` and `cardinality` are given below.

```
(defun remove-dups (X) (set-union X nil))
```

```
(defun cardinality (X) (len (remove-dups X)))
```

Exercise 7.11 `(equal (cardinality X) (sets::cardinality X))`

7.2 Fixpoint Theory

In this section, we develop a book in the package `SETS` on the theory of fixpoints. We do this in a very general setting, by using encapsulation to reason about a constrained function, `f`, of one argument. Later, we show that certain functions compute fixpoints by using functional instantiation. An advantage of this approach is that we can ignore irrelevant issues, *e.g.*, in a later section we show that certain functions compute fixpoints; these functions have many arguments, but `f` has only one.

We say that x is a *fixpoint* of f iff $f(x) = x$. If f is a monotonic function on the powerset of a set, then by the following version of the Tarski-Knaster theorem [105], it has a least and greatest fixpoint, denoted by μf and νf, respectively.

Theorem 7.1 *Let* $f : 2^S \rightarrow 2^S$ *such that* $a \subseteq b \quad \Rightarrow \quad f(a) \subseteq f(b)$. *Then*

1. $\mu f \;=\; \cap\{b : b \subseteq S \;\wedge\; f(b) \subseteq b\} \;=\; \cup_{\alpha \in On} f^{\alpha}(\emptyset)$, *and*
2. $\nu f \;=\; \cup\{b : b \subseteq S \;\wedge\; b \subseteq f(b)\} \;=\; \cap_{\alpha \in On} f^{\alpha}(S)$,

where 2^S *is the powerset of* S, f^{α} *is the* α*-fold composition (iteration) of* f, *and* On *is the class of ordinals.*

We say that x is a *pre-fixpoint* of f iff $x \subseteq f(x)$; x is a *post-fixpoint* iff $f(x) \subseteq x$. The Tarski-Knaster theorem tells us that μf is below all post-fixpoints and that νf is above all pre-fixpoints.

We can replace On by the set of ordinals of cardinality at most $|S|$; since we are only interested in *finite* sets, this gives us an algorithm for computing least and greatest fixpoints. Notice that by the monotonicity of f, $\alpha \leq \beta \quad \Rightarrow \quad f^{\alpha}(\emptyset) \subseteq f^{\beta}(\emptyset) \quad \wedge \quad f^{\beta}(S) \subseteq f^{\alpha}(S)$. Therefore, we can compute μf by applying f to $\emptyset$ until we reach a fixpoint; similarly, we can compute νf by applying f to S until we reach a fixpoint.

We start by constraining functions `f` and `S` so that `f` is monotonic and when `f` is applied to a subset of `S`, it returns a subset of `S`. Since functions defined in ACL2 are total, we cannot say that `f` is a function whose domain is the powerset of `S`. We could add hypotheses stating that all arguments to `f` are of the right type to the theorems that constrain `f`, but this generality is not needed and will make it slightly more cumbersome to prove theorems about `f`. The issue of what to do when a function is applied outside its intended domain is one that comes up quite a bit in ACL2. The definitions of the constrained functions follow.

```
(encapsulate
 ((f (X) t)
  (S () t))
 (local (defun f(X) (declare (ignore X)) nil))
 (local (defun S() nil))
 (defthm f-is-monotonic
   (implies (=< X Y)
            (=< (f X) (f Y))))
 (defthm S-is-top
   (=< (f X) (set-union X (S))))).
```

We now define `applyf`, a function that applies `f` a given number of times.

```
(defun applyf (X n)
  (if (zp n)
      X
    (if (== X (f X))
        X
      (applyf (f X) (1- n)))))
```

From the Tarski-Knaster theorem, we expect that `lfpf` and `gfpf`, defined below, are the least and greatest fixpoints, respectively.

```
(defabbrev lfpf () (applyf nil (cardinality (S))))

(defabbrev gfpf () (applyf (S) (cardinality (S))))
```

Now all that is left is to prove the Tarski-Knaster theorem, which is given as the following two exercises.

Exercise 7.12 *Prove that* `lfpf` *is the least fixpoint:*

1. `(== (f (lfpf)) (lfpf))`

2. `(implies (=< (f X) X) (=< (lfpf) X))`

Exercise 7.13 *Prove that* `gfpf` *is the greatest fixpoint:*

1. `(== (f (gfpf)) (gfpf))`

2. ```
   (implies (and (=< X (S)) (=< X (f X)))
            (=< X (gfpf)))
   ```
   ```

7.3 Relation Theory

In this section we develop a book, in the package `RELATIONS`, on the theory of relations. We represent relations as alists which map an element to the set of elements it is related to. A recognizer for relations is the following.

```
(defun relationp (r)
  (cond ((atom r) (eq r nil))
        (t (and (consp (car r))
                (true-listp (cdar r))
                (relationp (cdr r))))))
```

The definition of `image`, a tail-recursive function that computes the image of a set under a relation, follows.

```
(defun value-of (x alist) (cdr (assoc-equal x alist)))

(defun image-aux (X r tmp)
  (if (endp X)
      tmp
    (image-aux (cdr X) r
               (set-union (value-of (car X) r) tmp))))
(defun image (X r) (image-aux X r nil))
```

Exercise 7.14 *Define* `range`*, a function that determines the range of a relation.*

Exercise 7.15 *Define* `inverse` *so that it is tail recursive and computes the inverse of a relation.*

The following function checks if the range of its first argument (a relation) is a subset of its second argument.

```
(defun rel-range-subset (r X)
  (cond ((endp r) t)
        (t (and (=< (cdar r) X)
                (rel-range-subset (cdr r) X)))))
```

Exercise 7.16

1. `(implies (rel-range-subset r X) (=< (image Y r) X))`
2. ```
   (implies (and (rel-range-subset r X) (=< X Y))
            (rel-range-subset r Y))
   ```
   ```

7.4 Models

In this section we introduce the notion of a model. A *model*, sometimes called a Kripke structure or a transition system, is a four-tuple consisting of a set of states, a transition relation, a set of atomic propositions, and a labeling relation. The transition relation relates a pair of states if the second state can be reached from the first in a single step. The atomic propositions can be thought of as Boolean variables that are either true or false at a state. The labeling relation relates states to the atomic propositions true at those states. A program can be thought of as a model: there is a state for every combination of legal assignments to the program's variables—which can be recovered from the labeling of the state—and the transition relation relates a pair of states if, in one step, the program can transition from the first state to the second. There are some technical details to consider, *e.g.*, a program can have variables of varying types, but atomic propositions are Boolean, hence, program variables are represented by a set of atomic propositions (this set can be infinite if the domain of the variable is infinite). We restrict our attention to finite models because we want to check them algorithmically.

We define the notion of a model in ACL2. The functions defined in this section, as well as the next two sections, are in the package `MODEL-CHECK`. An ACL2 model is a seven-tuple because it is useful to precompute the inverse relations of the transition relation and the labeling relation as well as the cardinality of the set of states. The inverse transition relation relates a pair of states if, in one step, the first state can be reached from the second. The inverse labeling relation relates atomic propositions to the states at which they hold. A function that creates a model is defined below.

```
(defun make-model (s r ap l)
  (list s r ap l (inverse r) (inverse l) (cardinality s)))
```

Exercise 7.17 *Define* `modelp`*, a recognizer for models. Define the accessor functions:* `states`*,* `relation`*,* `atomic-props`*,* `s-labeling`*,* `inverse--relation`*,* `a-labeling`*, and* `size` *to access the: states, transition relation, atomic propositions, (state) labeling relation, inverse transition relation, (atomic proposition) labeling relation, and cardinality of the states, respectively.*

7.5 Mu-Calculus Syntax

We are now ready to look at the Mu-Calculus. Informally, a formula of the Mu-Calculus is either an atomic proposition, a variable, a Boolean combination of formulae, `EX`f, where f is a formula, or $\mu Y f$ or $\nu Y f$, where f is a formula and Y is a variable (as we will see when we discuss semantics,

```
(defun mu-symbolp (s)
  (and (symbolp s)
       (not (in s '(+ & MU NU true false)))))

(defun basic-mu-calc-formulap (f ap v)
  (cond ((symbolp f)
         (or (in f '(true false))
             (and (mu-symbolp f)
                  (or (in f ap) (in f v)))))
        ((equal (len f) 2)
         (and (in (first f) '(~ EX))
              (basic-mu-calc-formulap (second f) ap v)))
        ((equal (len f) 3)
         (let ((first (first f))
               (second (second f))
               (third (third f)))
           (or (and (in second '(& +))
                    (basic-mu-calc-formulap first ap v)
                    (basic-mu-calc-formulap third ap v))
               (and (or (in first '(MU NU)))
                    (mu-symbolp second)
                    (not (in second ap))
                    (basic-mu-calc-formulap
                     third ap (cons second v))))))))
```

Figure 7.1: The Syntax of the Mu-Calculus

f and Y define the function whose fixpoint is computed). Usually there is a further restriction that f be monotone in Y; we do not require this. We will return to the issue of monotonicity in the next section.

In Figure 7.1, we define the syntax of the Mu-Calculus (`ap` and `v` correspond to the set of atomic propositions and the set of variables, respectively). `Mu-symbolp` is used because we do not want to decide the meaning of formulae such as `'(mu + f)`.

Exercise 7.18 *Define* `translate-f`*, a function that allows us to write formulae in an extended language, by translating its input into the Mu-Calculus. The extended syntax contains* `AX` *(*`'(AX f)` *is an abbreviation for* `'(~ (EX (~ f)))`*) and the infix operators* `|` *(which abbreviates* `+`*),* `=>` *and* `->` *(both denote implication), and* `=`*,* `<->`*, and* `<=>` *(all of which denote equality).*

Exercise 7.19 `(Mu-calc-sentencep f ap)` *recognizes sentences (formulae with no free variables) in the extended syntax; define it.*

7.6 Mu-Calculus Semantics

The semantics of a Mu-Calculus formula is given with respect to a model and a valuation assigning a subset of the states to variables. The semantics of an atomic proposition is the set of states that satisfy the proposition. The semantics of a variable is its value under the valuation. Conjunctions, disjunctions, and negations correspond to intersections, unions, and complements, respectively. `EX`f is true at a state if the state has some successor that satisfies f. Finally, μ's and ν's correspond to least and greatest fixpoints, respectively. Note that the semantics of a sentence (a formula with no free variables) does not depend on the initial valuation. The formal definition is given in Figure 7.2; some auxiliary functions and abbreviations used in the figure follow.

```
(defabbrev semantics-EX (m f val)
  (image (mu-semantics m (second f) val)
         (inverse-relation m)))

(defabbrev semantics-NOT (m f val)
  (set-complement (mu-semantics m (second f) val)
                  (states m)))

(defabbrev semantics-AND (m f val)
  (intersect (mu-semantics m (first f) val)
             (mu-semantics m (third f) val)))

(defabbrev semantics-OR (m f val)
  (set-union (mu-semantics m (first f) val)
             (mu-semantics m (third f) val)))

(defabbrev semantics-fix (m f val s)
  (compute-fix-point
   m (third f) (put-assoc-equal (second f) s val)
   (second f) (size m)))

(defabbrev semantics-MU (m f val)
  (semantics-fix m f val nil))

(defabbrev semantics-NU (m f val)
  (semantics-fix m f val (states m)))
```

Now, we are ready to define the main function:

```
(defun semantics (m f)
  (if (mu-calc-sentencep f (atomic-props m))
      (mu-semantics m (translate-f f) nil)
    "not a valid mu-calculus formula"))
```

```
(mutual-recursion
(defun mu-semantics (m f val)
  (cond ((eq f 'true) (states m))
        ((eq f 'false) nil)
        ((mu-symbolp f)
         (cond ((in f (atomic-props m))
                (value-of f (a-labeling m)))
               (t (value-of f val))))
        ((equal (len f) 2)
         (cond ((equal (first f) 'EX)
                (semantics-EX m f val))
               ((equal (first f) '~)
                (semantics-NOT m f val))))
        ((equal (len f) 3)
         (cond ((equal (second f) '&)
                (semantics-AND m f val))
               ((equal (second f) '+)
                (semantics-OR m f val))
               ((equal (first f) 'MU)
                (semantics-MU m f val))
               ((equal (first f) 'NU)
                (semantics-NU m f val))))))

(defun compute-fix-point (m f val y n)
  (if (zp n)
      (value-of y val)
    (let ((x (value-of y val))
          (new-x (mu-semantics m f val)))
      (if (== x new-x)
          x
        (compute-fix-point
         m f (put-assoc-equal y new-x val) y (- n 1))))))
         ; note that the valuation is updated
)
```

Figure 7.2: The Semantics of the Mu-Calculus

`Semantics` returns the set of states in `m` satisfying `f`, if `f` is a valid Mu-Calculus formula, otherwise, it returns an error string.

How would you write a Mu-Calculus formula that holds exactly in those states where it is possible to reach a p-state (*i.e.*, a state labeled by the atomic proposition p)? The idea is to start with p-states, then add states that can reach a p-state in one step, two steps, and so on. When you are adding states, this corresponds to a least fixpoint computation. A solution is $\mu Y(p \lor \texttt{EX}Y)$; it may help to think about "unrolling" the fixpoint.

How would you write a Mu-Calculus formula that holds exactly in those states where every reachable state is a p-state? The idea is to start with p-states, then remove states that can reach a non p-state in one step, two steps, and so on. When you are removing states, this corresponds to a greatest fixpoint computation. A solution is $\nu Y(p \land \neg\texttt{EX}\neg Y)$; as before it may help to think about unrolling the fixpoint. Similar exercises follow so that you can gain some experience with the Mu-Calculus.

Exercise 7.20 *For each case below, define a Mu-Calculus formula that holds exactly in states that satisfy the description. A* path *is a sequence of states such that adjacent states are related by the transition relation. A* fullpath *is a maximal path,* i.e., *a path that cannot be extended.*

1. *There is a fullpath whose every state is a p-state.*
2. *Along every fullpath, it is possible to reach a p-state.*
3. *There is a fullpath with an infinite number of p-states.*

The model-checking algorithm we presented is *global*, meaning that it returns the set of states satisfying a Mu-Calculus formula. Another approach is to use a *local* model-checking algorithm. The difference is that the local algorithm is also given as input a state and checks whether that particular state satisfies the formula; in some cases this can be done without exploring the entire structure, as is required with the global approach.

The model-checking algorithm we presented is *extensional*, meaning that it represents both the model and the sets of states it computes explicitly. If any of these structures gets too big—since a model is exponential in the size of the program text, *state explosion* is common—resource constraints will make the problem practically unsolvable. Symbolic model-checking [74, 16, 86] is a technique that has greatly extended the applicability of model-checking. The idea is to use compact representations of the model and of sets of states. This is done by using BDDs[3] (binary decision diagrams), which on many examples have been shown to represent states and

[3]BDDs can be thought of as deterministic finite state automata (see any book covering Automata Theory, *e.g.*, [50]). A Boolean function, f, of n variables can be thought of as a set of n-length strings over the alphabet $\{0, 1\}$. We start by ordering the variables; in this way an n-length string over $\{0, 1\}$ corresponds to an assignment of values to the

models very compactly [14]. Symbolic model-checking algorithms, even for temporal logics such as CTL whose expressive power compared with the Mu-Calculus is quite limited, are based on the algorithm we presented (except that BDDs are used to represent sets of states and models).

Now that we have written down the semantics of the Mu-Calculus in ACL2, we can decide to stop and declare success, because we have an executable model-checker. In many cases this is an appropriate response, because deciding if you wrote what you meant is not a formal question. However, in our case, we expect that `MU` formulae are least fixpoints (if the formulae are monotonic in the variable of the `MU` and certain "type" conditions hold), and similarly `NU` formulae are greatest fixpoints. We will check this. We start by defining what it means to be a fixpoint.

```
(defun fixpointp (m f val x s)
  (== (mu-semantics m f (put-assoc-equal x s val)) s))

(defun post-fixpointp (m f val x s)
  (=< (mu-semantics m f (put-assoc-equal x s val)) s))

(defun pre-fixpointp (m f val x s)
  (=< s (mu-semantics m f (put-assoc-equal x s val))))
```

Read the rest of the exercises in this section before trying to solve any of them.

Exercise 7.21 *Use encapsulation to constrain the functions* `sem-mon-f`, `good-model`, `good-val`, *and* `good-var` *so that* `sem-mon-f` *is monotone in* `good-var`, `good-model` *is a "reasonable" model,* `good-val` *is a "reasonable" valuation, and* `good-var` *is a "reasonable" variable.*

We prove the fixpoint theorems by functionally instantiating the main theorems in the supporting book `fixpoints`. (See `lemma-instance`; an example of functional instantiation can be found in the companion book [58].)

Exercise 7.22 *Prove that* `MU` *formulae are least fixpoints and that* `NU` *formulae are greatest fixpoints. As a hint, we include the statement of one of the four required theorems.*

---

variables. We can represent f by an automaton whose language is the set of strings that make f true. We can now use the results of automata theory, *e.g.*, deterministic automata can be minimized in $O(n \log n)$ time (the reason why nondeterministic automata are not used is that minimizing them is a PSPACE-complete problem), hence, we have a canonical representation of Boolean functions. Automata that correspond to Boolean functions have a simpler structure than general automata (*e.g.*, they do not have cycles); BDDs are a data structure that takes advantage of this structure. Sets of states as well as transition relations can be thought of as Boolean functions, so they too can be represented using BDDs. Finally, note that the order of the variables can make a big (exponential) difference in the size of the BDD corresponding to a Boolean function.

```
(defmu semmu-is-a-fixpoint
  (fixpointp (good-model) (sem-mon-f) (good-val) (good-var)
             (mu-semantics
              (good-model)
              (list 'mu (good-var) (sem-mon-f))
              (good-val)))
  sets::lfix-is-a-fixpoint)
```

Exercise 7.23 *The hint in the previous example is a macro call. This saves us from having to type the appropriate functional instantiation several times. Define the macro. Our solution is of the following form.*

```
(defmacro defmu (name thm fn-inst &rest args)
  `(defthm ,name ,thm
     :hints
     (("goal"
       :use (:functional-instance
             ,fn-inst
             (sets::S (lambda() (states (good-model))))
             (sets::f (lambda(y) (mu-semantics  ...  )))
             (sets::applyf
              (lambda(y n) (compute-fix-point  ...  )))
             (sets::cardinality cardinality)))
      ,@args)))
```

You will notice that reasoning about mutually recursive functions (which is required for the exercises above) can be tricky, *e.g.*, even admitting the mutually recursive functions and verifying their guards (as mentioned in the introduction, this is an implicit exercise for every function we introduce) can be a challenge. Read the documentation for `mutual-recursion` and `package-reincarnation-import-restrictions`. There are several approaches to dealing with mutually recursive functions in ACL2. One is to remove the mutual recursion by defining a recursive function that has an extra argument which is used as a flag to indicate which of the functions in the nest to execute. Another approach is to identify a sufficiently powerful induction scheme for the functions, add it as an induction rule (see `induction`) so that this induction is suggested where appropriate, and prove theorems by simultaneous induction, *i.e.*, prove theorems that are about all the functions in the mutual recursion nest. We suggest that you try both approaches.

7.7 Temporal Logic

Temporal logics can be classified as either *linear-time* or *branching-time* (see [28]). In linear-time logics the semantics of a program is the set of its possible executions, whereas in branching-time, the semantics of a program is its computation tree; therefore, branching time logics can distinguish between programs that linear-time logics consider identical. A branching time logic of interest is CTL: many model-checkers are written for it because of algorithmic considerations. We present the syntax and semantics of CTL. It turns out that CTL, as well as the propositional linear time logic LTL, and the branching time logic CTL^* can be translated to the Mu-Calculus.

The syntax of CTL is defined inductively by the following rules:

1. p, where p is an atomic proposition, and

2. $\neg f, f \vee g$, where f is a CTL formula, and

3. $\mathtt{EX}f, \mathtt{E}(f\mathtt{U}g), \mathtt{E}\neg(f\mathtt{U}g)$, where f and g are CTL formulae.

Although we presented the syntax of CTL, it turns out to be just as easy to present the semantics of what is essentially CTL^*. The semantics are given with respect to a *fullpath*, *i.e.*, an infinite path through the model. If x is a fullpath, then by x_i we denote the i^{th} element of x and by x^i we denote the suffix $\langle x_i, \ldots \rangle$. Henceforth, we assume that the transition relation of models is *left total*, *i.e.*, every state has a successor. Note that CTL formulae are *state formulae*, *i.e.*, formulae whose semantics depends only on the first state of the fullpath. $M, x \models f$ means that fullpath x of model M satisfies formula f.

1. $M, x \models p$ iff x_0 is labeled with p;

2. $M, x \models \neg f$ iff not $M, x \models f$;
   $M, x \models f \vee g$ iff $M, x \models f$ or $M, x \models g$;

3. $M, x \models \mathtt{E}f$ iff there is a fullpath $y = \langle x_0, \ldots \rangle$ in M s.t. $M, y \models f$;
   $M, x \models \mathtt{X}f$ iff $M, x^1 \models f$; and
   $M, x \models f\mathtt{U}g$ iff there exists $i \in \mathbb{N}$ s.t. $M, x^i \models g$ and for all $j < i,\ M, x^j \models f$.

The first two items above correspond to Boolean formulae built out of atomic propositions. $\mathtt{E}f$ is true at a state if there exists a fullpath from the state that satisfies f. A fullpath satisfies $\mathtt{X}f$ if in one step (next time), the fullpath satisfies f. A fullpath satisfies $f\mathtt{U}g$ if g holds at some point on the fullpath and f holds until then.

The following abbreviations are useful:

$\mathtt{A}f \ = \ \neg\mathtt{E}\neg f, \ \ \mathtt{F}g \ = \ \mathsf{true}\,\mathtt{U}g, \ \ \mathtt{G}f \ = \ \neg\mathtt{F}\neg f$

$\mathtt{A}f$ is true at a state if every fullpath from the state satisfies f. A fullpath satisfies $\mathtt{F}g$ if eventually g holds on the path. A fullpath satisfies $\mathtt{G}f$ if f holds everywhere on the path.

Exercise 7.24 *Translate the following state formulae into Mu-Calculus formulae (the penultimate formula is not a CTL formula, but is a* CTL^* *formula which you can think of as saying "there exists a path such that infinitely often p"):* $\mathtt{EF}p, \mathtt{AF}p, \mathtt{AG}p, \mathtt{EG}p, \mathtt{EGF}p$*, and* $\mathtt{EGEF}p$*.*

Exercise 7.25 *Define a translator that translates CTL formulae (where the abbreviations above, as well as* true *and* false *are allowed) into the Mu-Calculus.*

7.8 Conclusions

We gave a formal introduction to model-checking via the Mu-Calculus, but only scratched the surface. We conclude by listing some of the many interesting directions one can explore from here. One can define a programming language so that models can be described in a more convenient way. One can make the algorithm symbolic, by using BDDs instead of our explicit representation. One can define the semantics of a temporal logic (*e.g.*, CTL^*) in ACL2 and prove the correctness of the translation from the temporal logic to the Mu-Calculus. One can use monotonicity arguments and memoization to make the model-checking algorithms faster. Finally, one can verify that the optimizations suggested above preserve the semantics of the Mu-Calculus.