Verifying TLA$^+$ Invariants with ACL2

Carlos Pacheco[⋆]

Department of Computer Sciences,
University of Texas at Austin,
Austin, Texas 78712
pacheco@cs.utexas.edu

Abstract. We describe the use of the ACL2 theorem prover to model and verify properties of TLA$^+$ specifications. We have written a translator whose input is a TLA$^+$ specification along with conjectures and structured proofs of properties of the specification. The translator's output is an ACL2 model of the specification, and a list of ACL2 conjectures corresponding to those sections of the proof outlines flagged for mechanical verification. We have used our tools to translate the Disk Synod algorithm, and to verify two invariants of the algorithm.

1 Introduction

Reasoning in TLA consists largely of reasoning about actions. By most accounts, 90% of all reasoning in TLA$^+$ specifications [1] occurs at the action level, where temporal logic has been eliminated. Action reasoning alone, for example, is involved in all but the last step of establishing an invariant of a specification. Consider a system whose starting state satisfies the formula *Init*, and whose next-state relation is described by *Next*. In order to prove an invariant *Inv* of the system, two lemmas are established:

1. $Init \Rightarrow Inv$
2. $Inv \wedge Next \Rightarrow Inv'$

Once we establish these two lemmas, one application of a TLA inference rule, along with simple temporal reasoning, lets us establish the invariant at the temporal level (the formula $\Box Inv$). In the Disk Synod algorithm [2], establishing $\Box Inv$ from formulas like (1) and (2) above takes up one page, while the creation of an invariant and its verification at the action level spans 18 pages.

This report summarizes our experiences using a mechanical theorem prover to verify properties of TLA$^+$ specifications. Our goal is to provide mechanical support for proving TLA$^+$ invariants at the action level. A system that deals

[⋆] This work was supported by an IBM Partnership Award to J Strother Moore, a UROP grant from the Educational Advancement Foundation, and by Compaq's Systems Research Center.

[1] TLA is a first-order temporal logic, while TLA$^+$ is a specification language based on TLA.

effectively with action-level formulas would take us a long way in mechanically verifying the correctness of specifications.

Our platform of choice for mechanical verification is ACL2 [3]. The ACL2 system is attractive for several reasons. It is among the most automated in the spectrum of theorem provers, it blends arithmetic decision procedures with rewriting techniques, and it is a stable and robust system, designed to tackle industrial-sized verification projects.

There are two main drawbacks to the use of ACL2 for verifying TLA$^+$ specifications. One is that, to make effective use of the prover's strengths, our TLA$^+$ constructs must be finite. Thus, infinite sets are not allowed in our specifications. The second drawback to using ACL2 is the different levels of abstraction at which TLA$^+$ and ACL2 users commonly operate. ACL2 theories are usually fairly low-level, concrete and computational. On the other hand, TLA$^+$ specifications tend to be more descriptive than constructive, and make liberal use of higher-level concepts which are difficult to handle in ACL2's first-order, essentially quantifier-free logic. For example, in TLA$^+$ we might write the maximal element of a set of integers as

$$\mathit{Max} \;\triangleq\; \text{CHOOSE } x \in S \,:\, \forall\, y \in S \,:\, x \geq y.$$

We can faithfully translate *Max* into ACL2 as follows. First, we define a function `(forall-1` S x`)` that captures the meaning of the universally quantified statement $\forall\, y \in S : x \geq y$. `Forall-1` goes through every element y in S, checking whether $x \geq y$, and returns `t` or `nil` accordingly. Next, we define a function `(filter-1` S`)` that finds every element m in S such that `(forall-1` S m`) = t`. Finally, we define *Max* as follows.

```
(defun Max (S) (choose (filter-1 S))).
```

Without the restriction of a faithful translation, we may be inclined to define the same concept—the maximal element of a set—in ACL2 with a single definition, say, as `(choose-max` S`)`, where `choose-max` is a recursive function that finds a maximal element in S by gradually working on a larger subset of S. (Ignore for the moment the representation of sets as ACL2 lists):

```
(defun choose-max (s)
    (cond ((empty s) nil)
          ((empty (cdr s)) (car s))
          ((>= (car s)
               (choose-max (cdr s)))
           (car s))
          (t (choose-max (cdr s)))))
```

Not surprisingly, properties of a maximal element are much easier to derive in ACL2 under the second representation.

A way to reconcile the conflict between a faithful translation and an effective translation is to do our proving using what comes naturally in ACL2, and then relate our "natural" constructs to their corresponding "spec-faithful" versions. (Notice that "natural" definitions in ACL2 will usually be at a lower-level of abstraction than their "spec-faithful" counterparts). In our above example, we could prove a theorem stating the relationship between `choose-max` and `Max`. Then, theorems about `choose-max` could be transferred to `Max` under appropriate hypotheses. This solution requires a greater degree of interaction with the theorem prover, but its lets us use ACL2 effectively and at the same time relates our work to to the original specification.

A first experiment at the University of Texas at Austin [8] consisted in manually translating the Disk Synod algorithm into ACL2 and verifying two invariants of the algorithm. The next step, performed at Compaq's Systems Research Center, was to automate the translation. Our tool not only translates TLA$^+$ specifications, but also *structured proofs* [4] of conjectures about the specifications. In writing a structured proof, we mark some reasoning steps as "checked by ACL2" and leave others unmarked. The translator creates a list of ACL2 `defthm` forms corresponding to the structured proof, and gives special names to those corresponding to the "proof by ACL2" steps in the structured proof. We use ACL2 to verify only such `defthm` forms. The idea is that, short of mechanically verifying every step of a proof, a user might first want to explore pieces of a proof that are not entirely clear or where he lacks confidence. Also, we want to use ACL2 only on those steps where it is appropriate to use ACL2 (low-level, quantifier-free formulas). A future proof checker for TLA$^+$ might, in addition to steps labeled "checked by ACL2," also have steps labeled "checked by X" where X is a different theorem prover.

In Section 2 we lay out the translation process from TLA$^+$ to ACL2. Section 3 discusses briefly the framework used to verify properties of translated specifications, and discusses some aspects of the Disk Synods specification where our verification effort shed light. Section 4 summarizes what we learned. Appendix A describes the mechanical translator in more detail.

2 Translation

TLA$^+$ is based in set theory, so we need an effective framework to reason about sets in ACL2. Our translation scheme builds upon the finite set theory work developed by Moore [7]; we will assume familiarity with it. (We also assume familiarity with ACL2 [3] and with TLA$^+$ [5].) Moore's theory allows us to express in a natural way basic set concepts like membership, set equality, ordered pairs, functions and sequences.

We use recursive functions on sets to define several concepts. Quantification is represented as a recursive function that tests a set's elements for the desired property. Set constructors, like $\{x \in S : p(x)\}$ and $\{f(x) : x \in S\}$, are also defined as recursive functions that transform or filter elements of a set. Through the use of ACL2 macros, we provide for a convenient notation to express quantification or set comprehension. For example, the ACL2 macro

```
(defall name (s) :forall x :in s :holds (p x))
```

creates a function (*name* `s`) that returns `t` if every element `x` in `s` satisfies `(p x)`, and returns `nil` otherwise. Examples of other frequently used macros are:

– `(defexists` *name* `(s) :exists x :in s :such-that (p x))` creates a function `(`*name* `s)` that returns `t` if some element `x` in `s` has property `(p x)`, and returns `nil` otherwise.
– `(defmap` *name* `(s) :for x :in s :such-that (p x))` creates a function `(`*name* `s)` that returns the set of elements `x` in `s` with property `(p x)`.
– `(defmap` *name* `(s) :for x :in s :map (f x))` creates a function `(`*name* `s)` that maps each element `x` in `s` into `(f x)`, and returns the set of mapped elements.
– `(defmap-fn` *name* `(s) :for x :in s :map (f x))` creates a function `(`*name* `s)` that maps each element `x` in `s` into `(f x)` and returns the set of ordered pairs $\langle$ `x` , `(f x)` $\rangle$.

Figure 1 presents our translation scheme for some TLA$^+$ expressions. We do not provide entries for concepts that translate directly into ACL2's built-in definitions (e.g. $\wedge$, $\vee$, $\Rightarrow$) or into Moore's finite set theory definitions (e.g. $\in$, $\subseteq$, $\cup$, $\cap$). Every set appearing in Figure 1 is assumed to be finite. Note that the constructs we cannot directly translate deal with quantification over the (infinite) universe of TLA$^+$ objects.

2.1 Naming Convention

As we mention above, an expression like $\{x \in S : p(x)\}$ is translated using the `defmap` macro, which defines a function that constructs the given set. As an ACL2 function, this set constructor needs a name. We name such expressions by concatenating the name of the top-level definition in which the expression appears with `forall`, `exists`, `subsetof` or `setofall`, depending on the expression. Finally, we append a number to the name to disambiguate similar expressions appearing in the same top-level definition. For example, if the TLA$^+$ expression $\{x \in S : p(x)\}$ is the first "set filtering" expression to occur as part of the definition of action A, it gets translated into something like `(A-subsetof-1 s)`, where `A-subsetof-1` is defined using the `defmap` macro:

```
(defmap A-subsetof-1 (s) :for x :in s :such-that (p x)).
```

Other TLA$^+$ constructs requiring a name in ACL2 are sets of records and quantified expressions. Their naming conventions are similar to the above example.

2.2 System Variables

We represent TLA$^+$ variables as ACL2 variables. We write the variable x as `x`, and the primed variable x' as `x-n`.

In TLA$^+$, state variables are global. TLA$^+$ definitions usually do not pass as parameters state variables involved in the definitions. Since ACL2 is applicative, we must include as arguments any variables used, including state variables. For example, the action $Next(a) \triangleq x' = x + a$ translates into the ACL2 event

```
(defun next (a x x-n) (= x-n (+ x a))).
```

Logic

BOOLEAN	`(brace t nil)`
$\forall x : p$	*no translation*
$\exists x : p$	*no translation*
$\forall x \in S : p$	`(`f S v_1 `...` v_k`)`, where f adheres to the naming convention, and is defined by `(deftla-forall` f `(dom` v_1 `...` v_k`) :forall` x `:in dom :holds` p`)`.
$\exists x \in S : p$	`(`f S v_1 `...` v_k`)`, where f adheres to the naming convention, and is defined by `(deftla-exists` f `(dom` v_1 `...` v_k`) :exists` x `:in dom :such-that` p`)`.
CHOOSE $x : p$	*no translation*
CHOOSE $x \in S : p$	`(choose (`f S v_1 `...` v_k`))`, where f adheres to the naming convention, and is defined by `(deftla-map` f `(dom` v_1 `...` v_k`) :for` x `:in dom :such-that` p`)`.

Sets

SUBSET S	`(powerset` S`)`
UNION S	`(union*` S`)`

Functions

$f[e]$	`(apply` f e`)`
DOMAIN f	`(domain` f`)`
$[x \in S \mapsto e]$	`(`f S v_1 `...` v_k`)`, where f adheres to the naming convention, and is defined by `(deftla-map-fn` f `(dom` v_1 `...` v_k`) :for` x `:in dom :map` e`)`.
$[S \rightarrow T]$	`(all-fns` S T`)`
$[f$ *EXCEPT* $![e_1] = e_2]$	`(except` f e_1 e_2`)`

Records

$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$	`(func (`h_1 e_1`) ... (`h_n e_n`))`
$[h_1 : S_1, \ldots, h_n : S_n]$	`(`*name*`)`, where *name* adheres to the naming convention and is defined by `(defrec` *name* `(`h_1 S_1`) ... (`h_n S_n`))`.

Fig. 1. TLA–ACL2 translations.

We have created a series of macros that let us define and use state functions and actions without explicit reference to the variables involved. The macro `(defaction next (a) (= x-n (+ x a)))` expands into the following events.

```
(defun _next (a x x-n) (= x-n (+ x a)))
(defmacro next (a) (_next a x x-n))
```

Now, we can write `(next a)` when referring to action $Next(a)$, without writing down the state variables involved. Similar macros used to hide variable arguments are:

- `defstate` : for defining state functions with variable hiding. The macro `(defstate` *name* `(`$x_1 \ldots x_n$`)` α`)` creates in turn two macros, `(`*name* $x_1 \ldots x_n$`)` and `(`*name*`-n` $x_1 \ldots x_n$`)`, referring to the state function in the current and next state, respectively.
- `deftla-exists` : same as `defexists`, with variable hiding.
- `deftla-forall` : same as `defall`, with variable hiding.
- `deftla-map` : same as `defmap`, with variable hiding.
- `deftla-map-fn` : same as `defmap-fn`, with variable hiding.

ACL2 expands away all macro calls in its output, so we will see the system variables as arguments to functions in ACL2's output. We have not found it a major distraction.

3 Proving Conjectures

Disk Synod is a distributed consensus algorithm in which a group of processors communicate through disks. In the paper introducing Disk Synod [1], Gafni and Lamport establish six invariance conjectures of Disk Synod. These invariants are used to prove consistency of the algorithm. Using our tools, we translated a TLA$^+$ specification of Disk Synod, as well as structured proofs of three of Gafni and Lamport's invariants. Of the three structured proofs we translated, we checked most of the proof steps in two of them with ACL2. We will not discuss the Disk Synod algorithm or its proof of correctness here; for such a discussion refer to [1]. The Disk Synod specification, the structured proofs and their translation can all be found under the `src/paxos/` directory (where `src/` is the directory where this report is located).

Figures 2 and 3 show the proof outlines for lemmas $I2a$ and $I2c$. These proof outlines (cast in a special syntax–see Appendix A) are the input to our tool, which translates them into ACL2 `defthm` events. In designing the translator, we want to ensure that someone looking at the structured proof of a conjecture can tell easily which steps are claimed to be mechanically checked. We also want to ensure that someone looking at the file of ACL2 `defthm` events corresponding to a structured proof can tell easily which events to prove in order to be consistent with any claims made in the structured proof. It is obvious from looking at the structured proofs in Figures 2 and 3 which steps are claimed to be mechanically checked. As for the list of `defthm` events corresponding to the structured proofs, our translator appends the suffix "-ACL2" to those events that must be mechanically checked. Other events are named according to their step number in the structured proof.

We write structured proofs using a special syntax (see Appendix A). We use the TLA$^+$ front end (in development at SRC) to parse the proof outlines. However, the front end has no notion of structured proofs; it only parses TLA$^+$ expressions and modules. Using TLA$^+$ syntax that the front end can handle, we encode structured proofs as TLA$^+$ expressions. After the front end parses them, we detect them as structured proofs and handle them accordingly. For more details on how structured proofs are handled and translated into ACL2 `defthm` events, see Appendix A.

We discuss lemmas $I2a$ and $I2c$ and their proofs at length in [8]. Here, we give some highlights.

In addition to spotting a number of typographic errors in Gafni and Lamport's written proofs, we discovered a nontrivial error in the statement of theorem $I2c$: an invariant ($HInv2$) was omitted as a hypothesis. Our proof effort has yielded a correction to the statement of Lemma $I2c$.

Our goal of mechanical verification also forced us to think about subtle and important details that should be mentioned in the original structured proofs of Lamport and Gafni. We now discuss two examples.

The next-state action. Disk Synod's next-state action is existentially quantified on the outside:

$$
\begin{array}{lll}
Next & \triangleq & \exists\, p \in Proc : \begin{array}[t]{l}
\lor\ StartBallot(p) \\
\lor\ \exists\, d \in Disk : \begin{array}[t]{l}
\lor\ Phase0Read(p, d) \\
\lor\ Phase1or2Write(p, d) \\
\lor\ \exists\, q \in Proc\ p : Phase1or2Read(p, d, q)
\end{array} \\
\lor\ EndPhase1or2(p) \\
\lor\ Fail(p) \\
\lor\ EndPhase0(p)
\end{array}
\end{array}
$$

The next-state action is typically a hypothesis in an invariant conjecture. In such a proof, we might want to show that every processor in the system has some property Φ which is preserved across steps. We do this by assuming a constant process p with property $\Phi(p)$, and showing that $\Phi'(p)$ holds with respect to the next-state action. Notice that we have mentioned the variable p twice—once in the definition of *Next* above, and once in mentioning a particular processor p for which Φ holds. However, these two mentions of p are not necessarily mentions of the same processor.

What does it mean to assume *Next*? It means that for some processor —call it p_2—*Next* holds, or intuitively, an action of *Next* is "executed." Does it follow that p_2 is the same as the constant p for which $\Phi(p)$ holds? Not necessarily. Yet, in their proof of lemma $I2c$, Gafni and Lamport make no distinction between p_2 and p. In our verification effort, we were required to make such distinctions, and to establish invariants for both cases.

```
$HInv1 \land HNext \Rightarrow HInv1'$
ASSUME:  1. CONSTANT $p \in Proc$
         2. CONSTANT $q \in Proc \setminus \{P\}$
         3. CONSTANT $d \in Disk$
         4. $HInv1$
         5. $\lor\ StartBallot(p)$
            $\lor\ Phase0Read(p, d)$
            $\lor\ Phase1or2Write(p, d)$
            $\lor\ Phase1or2Read(p, d, q)$
            $\lor\ EndPhase1or2(p)$
            $\lor\ Fail(p)$
            $\lor\ EndPhase0(p)$
         6. ChosenAllinputAction
PROVE:  $HInv1'$

⟨1⟩1. CASE: StartBallot(p)
  ASSUME: 1. CONSTANT $b \in Ballot(p)$
          2. $\land\ b > dblock[p].mbal$
             $\land\ dblock'[dblock$ EXCEPT $![p].mbal = b]$
  PROVE:  $HInv1'$
    PROOF: By ACL2.
⟨1⟩2. CASE: Phase1or2Write(p,d)
  PROOF: By ACL2.
⟨1⟩3. CASE: Phase1or2Read(p,d,q)
  PROOF: By ACL2.
⟨1⟩4. CASE: Phase0Read(p,d)
  PROOF: By ACL2.
⟨1⟩5. CASE: Fail(p)
  PROOF: By ACL2.
⟨1⟩6. CASE: EndPhase0(p)
  ASSUME: 1. CONSTANT $b \in Ballot(p)$
          2. $\land\ \forall r \in allBlocksRead(P) \,:\, B > r.mbal$
             $\land\ dblock' = [dblock$ EXCEPT $![P] = [r$ EXCEPT $!.mbal = B]]\rangle$
  PROVE:  $HInv1'$
    PROOF: By ACL2.
⟨1⟩7. CASE: EndPhase1or2(p)
  PROOF: By ACL2.
⟨1⟩8. Q.E.D.
  PROOF: Cases are exhaustive.
```

Fig. 2. Lemma I2a.

```
$HInv1 \wedge HInv2 \wedge HInv3 \wedge HNext \Rightarrow HInv3'$
ASSUME:  1. CONSTANTS $p, p2 \in Proc$
         2. CONSTANT $q \in Proc$
         3. CONSTANT $q2 \in Proc \setminus \{P\}$
         4. CONSTANTS $d, d2 \in Disk$
         5. $HInv1 \wedge HInv2 \wedge HInv3$
         6. $\vee\ StartBallot(p2) Phase0Read(p2, d2)$
            $\vee\ Phase1or2Write(p2, d2) Phase1or2Read(p2, d2, q2)$
            $\vee\ EndPhase1or2(p2) Fail(p2) EndPhase0(p2)$
         7. $ChosenAllinputAction$
         8. $phase'[p] \in 1, 2 \wedge phase'[q] \in 1, 2 \wedge hasRead(p, d, q)' \wedge hasRead(q, d, p)'$
PROVE:  $\vee\ [block \mapsto dblock'[q], proc \mapsto q] \in blocksRead'[p][d]$
        $\vee\ [block \mapsto dblock'[p], proc \mapsto p] \in blocksRead'[q][d]$

⟨1⟩1. CASE: StartBallot(p)
  ASSUME: 1. CONSTANT $b \in Ballot(p)$
          2. $\wedge\ b > dblock[p].mbal$
             $\wedge\ dblock'[dblock$ EXCEPT $![p].mbal = b]$
  PROVE:  $\vee\ [block \mapsto dblock'[q], proc \mapsto q] \in blocksRead'[p][d]$
          $\vee\ [block \mapsto dblock'[p], proc \mapsto p] \in blocksRead'[q][d]$
    PROOF: By ACL2.
⟨1⟩2. CASE: Phase1or2Write(p,d)
  PROOF: By ACL2.
⟨1⟩3. CASE: Phase1or2Read(p,d,q)
  ⟨2⟩1. CASE: d2 $\neq$ d
    PROOF: By ACL2.
  ⟨2⟩2. CASE: d2 = d
    ⟨3⟩1. CASE: p2 $\neq$ p $\wedge$p2 $\neq$ q
      PROOF: By ACL2.
    ⟨3⟩2. CASE: p2 = p
      ⟨4⟩1. CASE: q2$\neq$q
        PROOF: By ACL2.
      ⟨4⟩2. CASE: q2=q
        PROOF: By ACL2.
      ⟨4⟩3. Q.E.D.
        PROOF: Cases ⟨4⟩1 and ⟨4⟩2 are exhaustive.
    ⟨3⟩3. CASE: p2=q
      ⟨4⟩1. CASE: q2$\neq$q
        PROOF: By ACL2.
      ⟨4⟩2. CASE: q2=q
        PROOF: By ACL2.
      ⟨4⟩3. Q.E.D.
        PROOF: Cases ⟨4⟩1 and ⟨4⟩2 are exhaustive.
    ⟨3⟩4. Q.E.D.
      PROOF: Cases ⟨3⟩1, ⟨3⟩2 and ⟨3⟩3 are exhaustive.
  ⟨2⟩3. Q.E.D.
    PROOF: Cases ⟨2⟩1 and ⟨2⟩2 are exhaustive.
⟨1⟩4. CASE: Phase0Read(p,d)
  PROOF: By ACL2.
⟨1⟩5. CASE: Fail(p)
  PROOF: By ACL2.
⟨1⟩6. CASE: EndPhase0(p)
  PROOF: By ACL2.
⟨1⟩7. CASE: EndPhase1or2(p)
  PROOF: By ACL2.
⟨1⟩8. Q.E.D.
  PROOF: Cases are exhaustive.
```

Fig. 3. Lemma I2c.

An unmentioned invariant. For our verification to succeed, we had to establish an unmentioned invariant of Disk Synod, which we call the well-behaved invariant. In Disk Synod, if every processor has a local copy of a variable named x, the set of these local variables is modeled as one shared variable x, which is a function mapping each processor p to its corresponding value $x[p]$. The well-behaved invariant says that when a processor p changes the value of a shared variable like x, it changes only its own slot in the variable. Figure 4 shows the well-behaved invariant. This invariant is crucial in checking most steps of Lemma $I2c$.

$$
\begin{aligned}
\mathit{HNext}(p2) \land (p2 \neq p) \Rightarrow\ & \lor\ \mathit{input}'[p] = \mathit{input}[p] \\
& \lor\ \mathit{output}'[p] = \mathit{output}[p] \\
& \lor\ \mathit{disk}'[p] = \mathit{disk}[p] \\
& \lor\ \mathit{phase}'[p] = \mathit{phase}[p] \\
& \lor\ \mathit{dblock}'[p] = \mathit{dblock}[p] \\
& \lor\ \mathit{diskswritten}'[p] = \mathit{diskswritten}[p] \\
& \lor\ \mathit{blocksread}'[p] = \mathit{blocksread}[p]
\end{aligned}
$$

Fig. 4. The “well-behaved” invariant.

4 Conclusion

An important lesson we learned is perhaps an obvious one: use a tool only where its strengths will shine. ACL2 is a general-purpose theorem prover, and one can use it to verify every step of a proof in any mathematical domain, from real analysis to circuit design. In our first experiments, we used ACL2 to verify every step in the proofs of $I2a$ and $I2c$. More than half our time was spent trying to reason about simple steps in higher-level concepts like quantification. Our second approach was to use ACL2 only where it might be suitable—closer to the leaves of a proof, where quantification has been eliminated and all that remains are large but low-level formulas. Although low-level, these formulas are nontrivial and would be a challenge for any theorem prover. Moreover, it is most often in these elaborate steps where errors are uncovered. It is to ACL2's credit that it did so much work with little guidance. At the correct level of abstraction, the prover not only helped us verify statements, but it also pointed the way to omissions and errors with remarkable precision.

In further work, we would continue focusing ACL2's attention on low-level segments of TLA$^+$ proofs, refining our tools and lemma libraries to increase the prover's power in this restricted domain. For the remaining high-level steps of TLA$^+$ proofs, we might recruit a different theorem prover with a logic more expressive than ACL2's. The framework for structured proofs we have followed allows for collaboration among multiple provers—each with its own strengths—in attacking a verification project.

5 Acknowledgments

I would like to thank J Moore, Leslie Lamport and Yuan Yu for all their help.

References

1. Eli Gafni and Leslie Lamport. Disk Synod. Technical Report 163, Compaq Systems Research Center, July 2000.
2. Eli Gafni and Leslie Lamport. Disk Paxos. in Maurice Herlihy, editor, Distributed Computing: 14th International Conference, DISC 2000 *Lecture Notes in Computer Science number 1914,* pages 330-344, Springer-Verlag, 2000.
3. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, *Computer-Aided Reasoning: An Approach,* Kluwer Academic Publishers, 2000.
4. Leslie Lamport. How to Write a Proof. *American Mathematical Monthly 102, 7 (August-September 1993)* pages 600-608.
5. Leslie Lamport. Specifying Concurrent Systems with TLA+. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design.*
6. Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems,* 16(3):872-923, May 1994.
7. J Moore. Finite Set Theory in ACL2. *TPHOLS '01*, Edinburgh, September 2001.
8. Carlos Pacheco. Reasoning about TLA Actions. Undergraduate Honors Thesis. Technical Report TR01-16, Department of Computer Sciences, The University of Texas at Austin, May 2001.

A The Mechanical Translator

This appendix describes the mechanical translator in more detail. Our starting point is the TLA$^+$ Java front end developed at SRC. The front end takes as input a TLA$^+$ module, parses it, performs some semantic analysis, and returns a set of semantic trees corresponding to the TLA$^+$ definitions and declarations appearing in the given module.

Starting with the semantic trees generated by the Java front end, the translation into ACL2 proceeds in two stages.

A.1 Stage 1: Generating S-expressions

The first part of our translation tool is a program written in Java, *pass-one*, that is responsible for calling the front end on a TLA$^+$ file specified by the user. If the front end successfully processes the file, *pass-one* creates a new file, *intermediate.lisp*, which contains basically the same semantic trees generated by the front end, encoded as s-expressions. We now describe the encoding process. Assume a semantic tree S. In what follows, we identify S with the TLA$^+$ expression it represents, so instead of saying that S is a semantic node of type `StringKind` that represents the string *str*, we say that S is the string *str*.

- If S is a CONSTANT declaration of a constant C with n arguments, it is encoded as the s-expression `(`C n`)`. *Pass-one* actually obtains all constant declarations at the same time from the front end, encodes them as described above, and wraps them into the following s-expression:
  `(tla-constants (`C_1 n_1`) ... (`C_k n_k`))`
- If S is an ASSUME declaration, it is not translated or appended to the list of s-expressions. (This has nothing to do with our ability to process these declarations. A more developed translator would handle ASSUME declarations.)
- If S is the number n, it is encoded as n.
- If S is the string s, it is encoded as `"`s` "`.
- If S is a definition $f(x_1, \ldots, x_n) \triangleq exp$, it is encoded as
  `(definition (`*line col*`)` *level* f `(`$x_1 \ldots x_n$`)` exp_2`)`
  where *line* and *col* are the line and column numbers where f's definition appears in the source TLA$^+$ file, *level* equals `constant-level`, `variable-level` or `action-level`, depending on the level of *exp* (deduced by the front end), and exp_2 is the encoding of *exp*.
- If S has the form
  let $def_1 \triangleq exp_1$,
  $\ldots$
  $def_n \triangleq exp_n$
  in *exp*
  it is encoded as `(let-in (`*deflist*`)` exp_2`)`, where *deflist* is the list obtained by encoding definitions def_1 through def_n, and exp_2 is the encoding of expression *exp*.
- If S is the set of records $[h_1 : S_1, \ldots, h_n : S_n]$, it is encoded as
  `(set-of-records (`*line col*`) (`h_1 S_1`) ... (`h_n S_n`))`.

  NOTE: For successful translation into ACL2, a set of records must be a constant expression—it must not be defined in terms of any variables.
- The following constructs are encoded as `(set-comp` *type* `(`*line col*`) ((`x_1 S_1`) ... (`x_n S_n`))` α`)`, where *type* corresponds to the kind of expression as follows.

$\{x_1 \in S_1 : \alpha\}$	(*type* = `subsetof`)
$\{\alpha : x_1 \in S_1\}$	(*type* = `setofall`)
$\forall x_1 \in S_1, \ldots, x_n \in S_n : \alpha$	(*type* = `forall`)
$\exists x_1 \in S_1, \ldots, x_n \in S_n : \alpha$	(*type* = `exists`)
CHOOSE $x \in S : \alpha$	(*type* = `boundedchoose`)
$[x_1 \in S_1 \mapsto \alpha]$	(*type* = `function`)

- If S is the unprimed variable x, it is encoded as x.
- If S is the primed variable x', it is encoded as x `-n`.
- If S is $f(x_1, \ldots, x_n)'$ for some user-defined operator f, it is encoded as `(`f`-n` x_1 `...` x_n`)`. NOTE: This encoding makes sense for user-defined state functions, because for state functions, we will ultimately create two ACL2 function calls, f and f `-n`, denoting f in its current and next state. The encoding makes no sense for other primed expressions, like $(x + y)'$. We assume that only user-defined state functions or variables are primed in the specification. This limits the TLA$^+$ expressions we can translate.

– Any other operator application $f(x_1, \ldots, x_n)$ is encoded as $(f_2\ x_1 \ldots x_n)$, where f_2 equals f for user-defined operators, but may differ from f for other operator names like `in`, which is replaced by `mem`, the ACL2 set membership function name.

NOTES:

– We do not add definitions of operators already built into ACL2, such as boolean operators or set theory operators.
– There is no support for primed expressions other than the ones outlined above; other primed expressions will be translated incorrectly.
– We do not expect to encounter unbound expressions like $\forall x : p$. If we do encounter them, the translation will continue, but the final translation will fail to be admitted by ACL2 (it's easy to see why it fails because in place of the unbounded expression, an error message is inserted.)

A.2 Stage 2: Generating ACL2 Events

We now describe *pass-two*, the program that translates the file *intermediate.lisp* into a file of ACL2 events [2]. This program is written in ACL2 itself, a subset of Common LISP.

For an input TLA$^+$ file *spec.tla*, *pass-two* creates two files, *spec.lisp* and *spec-constants.lisp*. The file *spec-constants.lisp* contains a list of ACL2 constants (not to be confused with TLA$^+$ constants) helpful to the developer of the TLA-ACL2 system. For instance, the constant `*all-defs*` is a listing of the contents in *intermediate.lisp*. Since this constant will be loaded into the system along with a specification, the intermediate translation will be available, and macros can be developed that use `*all-defs*` to look for information about the specification. Other constants defined in *spec-constants.lisp* are `*final-defs*`, a list of all the final events generated, and `*variables*`, a list of the system variables.

The file *spec.lisp* is the most important file created in the translation process. It contains the ACL2 events corresponding to the translation.

Up to this point, we have just taken a list of semantic trees stored as Java data structures, and converted them into s-expressions. Now, we detail how these s-expressions are translated into ACL2 events.

For each (non-theorem) definition, *pass-two* calls the function *Translate*. For each theorem, it calls the function *CreateThms*.

How *Translate* Works *Translate*'s job is to take an s-expression generated by *pass-one*, and produce an ACL2 event, or list of events, corresponding to the expression. It works as follows. (Our description is intuitive, and not meant to be formal. For more details, read the commented code.)

[2] An ACL2 event is a form submitted at the ACL2 prompt that causes ACL2 to take some action, like define a new function or prove a theorem.

– The expression `(tla-constants (`C_1 n_1`) ... (`C_k n_k`))` becomes the list of events

  `(defstub` C_1 `(`$x_1 \ldots x_{n_1}$`) t)`
  `...`
  `(defstub` C_k `(`$x_1 \ldots x_{n_k}$`) t)`

– The expression `(tla-variables (`v_1 `...` v_n`))` becomes the event
  `(defconst *variables* '(`v_1 `...` v_n`)).`
– The expression
  `(definition (`*line col*`)` *level f* `(`$x_1 \ldots x_n$`)` *expr*`)`
  can occur in two contexts: as a top-level definition, or as a definition in a LET-IN form. Both instances are handled the same way. The following ACL2 event is created:
  `(`*event f* `(`$x_1 \ldots x_n$`)` *body*`)`
  where *event* is `deftla-fun`, `defstate`, or `defaction`, depending on *level* being `constant-level`, `variable-level`, or `action-level` (respectively), and *body* is the result of calling *Translate* on *expr*.
– The expression `(set-of-records (`*line col*`) (`h_1 S_1`) ... (`h_n S_n`))` becomes the ACL2 event
  `(defrec` *name* `(`h_1 S_1`) ... (`h_n S_n`))`
  where *name* is obtained by concatenating the identifiers $h_1 \ldots h_n$. The occurrence of the set of records within an expression is replaced by the function call `(`*name*`)`.
– The expression `(set-comp` *type* `(`*line col*`) ((`x_1 S_1`) ... (`x_n S_n`))` α`)` is translated as follows.

  1. If $n = 1$ (there is only one bound pair `(`x_1 S_1`)`), then
     - `(set-comp setofall (`*line col*`) ((`x_1 S_1`))` α`)` becomes
       `(deftla-map` *name* `(dom` $a_1 \ldots a_n$`) :for` x_1 `:in dom :map` α_2`)`
     - `(set-comp subsetof (`*line col*`) ((`x_1 S_1`))` α`)` becomes
       `(deftla-map` *name* `(dom` $a_1 \ldots a_n$`) :for` x_1 `:in dom :such-that` α_2`)`
     - `(set-comp function (`*line col*`) ((`x_1 S_1`))` α`)` becomes
       `(deftla-map-fn` *name* `(dom` $a_1 \ldots a_n$`) :for` x_1 `:in dom :map` α_2`)`
     - `(set-comp exists (`*line col*`) ((`x_1 S_1`))` α`)` becomes
       `(deftla-exists` *name* `(dom` $a_1 \ldots a_n$`) :exists` x_1 `:in dom`
       `:such-that` α_2`)`
     - `(set-comp forall (`*line col*`) ((`x_1 S_1`))` α`)` becomes
       `(deftla-forall` *name* `(dom` $a_1 \ldots a_n$`) :forall` x_1 `:in dom`
       `:holds` α_2`)`

     Where

     - $a_1 \ldots a_n$ are the context parameters appearing in α. We illustrate the meaning of *context parameters* with an example: given the TLA$^+$ definition $f(a, b, c) \ \triangleq\ a \in \{x \in b : x = c\}$, the context parameters of the expression $\{x \in b : x = c\}$ are b and c.
     - α_2 is the translation of α.
     - *name* is determined by the naming convention (see Section 2).

The occurrence of the quantified expression is replaced by the call `(`*name* S_1 $a_1 \ldots a_n$`)`.

2. if $n > 1$ (there are several bound pairs `(`x_i S_i`)`), we first translate the expression
   exp$_2$ =`(set-comp` *type* `(`*line col*`) ((`x_2 S_2`) ... (`x_n S_n`))` α`)`, and then we translate the expression `(set-comp` *type* `(`*line col*`) ((`x_1 S_1`))` α_2`)`, where α_2 is a call of the function resulting from *exp*$_2$.

– Any other expression translates into itself.

How *CreateThms* Works Ideally, the front end would be able to parse structured proofs. Since it does not, we have developed an ad-hoc encoding for proofs, using expressions (such as tuples and sets of records) that the front end handles. This way, we can write proofs inside a module, and have them parsed by the front end. To the front end (and to *pass-one*) proofs are just TLA$^+$ expressions; they aren't handled in any special way. It is during the second stage of translation that we recognize proofs and generate the appropriate ACL2 events for them.

We define **proofs** and **assertions** in a mutually recursive fashion.

Proof. The proof of a statement **P** is a sequence of steps:

[$\mathbf{step_1}$: $\mathbf{Assertion_1}$,
 ...
 $\mathbf{step_n}$: $\mathbf{Assertion_n}$,
 $\mathbf{step_{n+1}}$: **QEDStep**]

Where

– $\mathbf{step}_i$ is a record field identifier of the form `s_`i`_`j, where i and j are integers denoting level and step numbers of the proof (see [4]).
– $\mathbf{Assertion}_i$ is an assertion as defined below.
– **QEDStep** is a string containing an explanation of the proof of **P**. The string must begin with "Q.E.D.". Examples are "Q.E.D. By propositional logic" or "Q.E.D. ACL2". The latter example is important — this is how we let the translator know that a proof is expected to be mechanically checked by ACL2.

Writing a **QEDStep** where a level-i proof is expected is shorthand for

[**step_i_1** : **QEDStep**].

Assertion. An assertion can have two forms:

– [*assume* : **Assumptions**, *prove* : **Goal**, *proof* : **Proof**]
– [*case* : **Case**, *proof* : **Proof**]

Where

- **Assumptions** can be a single TLA$^+$ expression, a sequence of TLA$^+$ expressions $\langle \textbf{Assm}_1, \ldots, \textbf{Assm}_n \rangle$, or a record $[\, \textbf{a1} : \textbf{Assm}_1, \ldots, \textbf{an} : \textbf{Assm}_n \,]$. The last option is intended to let the user name assumptions and refer to them by name at a later stage in a proof, but this functionality has not been implemented.
- **Proof** is a proof as defined above.
- **Goal** is a TLA$^+$ expression, denoting the statement to be proved.
- **Case** is a TLA$^+$ expression, denoting the case to consider.

NOTE. An assertion [*case* : **Case**, *proof* : **Proof**] is equivalent to

[*assume* : **Case**, *prove* : **P**, *proof* : **Proof**]

where **P** is the statement whose proof contains the assertion. To illustrate, the following two assertions are equivalent.

```
[ assume : << >>,
  prove  : P,
  proof  :
    [ s_1_1 : [ assume : x = 1,
                prove  : P,
                proof  : "Q.E.D." ],
      s_1_2 : [ assume : x # 1,
                prove  : P,
                proof  : "Q.E.D." ]]]
```

```
[ assume : << >>,
  prove  : P,
  proof  :
    [ s_1_1 : [ case   : x = 1,
                proof  : "Q.E.D." ],
      s_1_2 : [ case   : x # 1,
                proof  : "Q.E.D." ]]]
```

Introducing new identifiers. Sometimes we need to introduce new identifiers in a proof. For example, when proving $(\exists x \in S) \Rightarrow P$, we might assume the existence of a constant c belonging to S, and use c to establish P. We introduce new names in proofs using the LET-IN construct. In our example, we would write

```
[ assume : << >>,
  prove  : P,
  proof  :
   LET c == "new"
   IN
   [ s_1_1 : [ assume : c \in S,
               prove  : P,
               Proof  : .....]]]
```

The value `"new"` assigned to `c` is only a placeholder—in order for the above structure to be parsed `c` needs a definition. In ACL2, we define `c` to be an constant with no properties.

Steps that choose a value. A proof step may involve choosing a value with certain properties from a set. Such a step is accompanied with a proof of the value's existence. Here is an example. (For a richer example, see [4] p.7)

$< 3 >$ 1. Choose $x \in S$ such that $p(x)$
PROOF: Let x be 2. Then $p(2)$.

$< 3 >$ 2....
$< 3 >$ 3....

We translate the above example as follows. Notice that we add a step corresponding to the proof obligation arising from our choice of `x`.

```
LET x == CHOOSE S
IN
[ s_3_1 : p(x),
  s_3_2 : ...
  s_3_3 : ...]
```

Generating Theorems. We now describe how a list of ACL2 `defthm` events is generated from a structured proof. We start with the s-expression representation of a structured proof (i.e. a set of records denoting a structured proof). *Second-pass* recognizes definitions whose name contains the substring `theorem`. In the source TLA$^+$ file, a theorem P looks like this.

```
Theorem$Name$ ==
  [ assume : << >>,
    prove  : $P$,
    proof  : $\alpha$
  ]
```

For this description, we assume that *CreateThms* takes three arguments: a list of assumptions, a goal, and a proof. The top-level call to *CreateThms* is $CreateThms(nil, P, \alpha)$.

Now, consider a general instance $CreateThms(assumptions, goal, proof)$ of a call to *CreateThms*.

- If *proof* is a Q.E.D. step, create the ACL2 event

  `(defthm` *name* `(implies (and` *assumptions*`)` *goal*`))`

  where *name* is the name of theorem, concatenated with the step name, concatenated with the string “ `-ACL2`” if the proof is “Q.E.D. ACL2”.
- Otherwise, *proof* is a sequence of steps. For each step s in the proof, generate a list of events as follows.
  - If s is of the form
    [*assume* : *newAssumptions*, *prove* : *newGoal*, *proof* : *newProof*],
    generate a list consisting of the ACL2 events returned by the call
    $CreateThms(assumptions + newAssumptions, newGoal, newProof)$
    and the event

    `(defthm` *name* `(implies (and` *assumptions*`)` *goal*`)),`

    where *name* is the name of the theorem, concatenated with the step name.

- If s is of the form
  $[\ case\ :\ newCase, proof\ :\ newProof\ ]$,
  generate a list consisting of the ACL2 events returned by the call
  $CreateThms(newCase + assumptions, goal, newProof)$
  and the event

  (defthm *name* **(implies (and** *assumptions***)** *goal***)).**

  where *name* is the name of the theorem, concatenated with the step name.

Concatenate all the lists of events resulting from processing the steps, and return the concatenated list.