A COMPLETENESS THEOREM FOR TLA $\boxed{1}$

Preface: A Quick Introduction to TLA

Here's a quick and dirty informal definition and semantics for TLA. A more precise one is given later. If you are already familiar with the new, improved TLA (defined a little differently than in SRC Report No. 57) you can skip to the Introduction.

Values:
I assume a set of values, big enough to contain all the constants of interest. It includes the values 1, TRUE, NAT (the set of all naturals), $\{n \in NAT : n \text{ a prime}\}$, etc.

State, Variable:
A variable is something that assigns a value to every state. I let s.x, denote the value state s assigns to variable s. Or maybe a state s is something that assigns a value s.x to a variable x. Take your pick.

State Function:
An expression made from variables and constants, such as $x^2 + 3*y$. A state function f assigns a value s.f to every state s. For example,

$$s.(x^2 + 3*y) = (s.x)^2 + 3*(s.y).$$

Predicate:
A boolean-valued state function--for example,

$$x^2 < 3*y$$

Action:
A Boolean expression involving variables, primed variables and constants--for example, $x + 1 < 2*y'$. An action maps pairs of states to Booleans. Letting s.A.t denote the value that action A assigns to the pair (s,t), I define

$$s.(x + 1 < 2*y').t = (s.x) + 1 < 2*(t.y)$$

In other words, the unprimed variables talk about the left-hand state, and the primed variables talk about the right-hand state. Think of s.A.t = TRUE as meaning that an A-step can take state s to state t. An action is valid, written $\models$ A, iff s.A.t is true for all states s and t.

Enabled(A):
For any action A, the predicate Enabled(A) is defined by

$$s.\mathrm{Enabled}(A) \triangleq \exists\, t : s.A.t$$

$f'=f$:
For any state function f, the action $f'=f$, which is sometimes written Unchanged(f), is defined by

$$s.(f'=f).t \triangleq (t.f) = (s.f)$$

$[A]_f$:
The action $[A]_f$ is defined by

$$[A]_f \triangleq A \lor (f'=f)$$

An $[A]_f$ step is either an A step or leaves f unchanged.

$\langle A\rangle_f$:
The action $\langle A\rangle_f$ is defined by

$$\langle A\rangle_f \triangleq \neg[\neg A]_f$$

It equals $A \wedge (f' \neq f)$. An $<A>_f$ step is an A step that changes f.

The Raw Logic:
A Raw Logic formula is any formula made from actions using logical operators and the unary $\Box$ operator--for example

$$A \vee \Box(B \wedge \Box\neg\Box\neg A)$$

where A and B are actions. A Raw Logic formula is a Boolean-valued function on infinite sequences of states. An infinite sequence of states is called a BEHAVIOR. An action A is interpreted as the temporal formula asserting that first step of the behavior is an A step. The formula $\Box A$ asserts that every step is an A step. In general, let $s_0, s_1, \ldots \models F$ denote the value that formula F assigns to the sequence $s_0, s_1, \ldots$. The semantics of Raw Logic formulas is defined as follows, where A is any action and F and G are any formulas:

$$s_0, s_1, s_2, \ldots \models A \ \triangleq\ s_0.A.s_1$$
$$s_0, s_1, s_2, \ldots \models \Box F$$
$$\triangleq\ \forall\, n \geq 0 : s_n, s_{n+1}, s_{n+2}, \ldots \models F$$
$$s_0, s_1, s_2, \ldots \models \neg F \ \triangleq\ \neg(s_0, s_1, s_2, \ldots \models F)$$
$$s_0, s_1, s_2, \ldots \models F \vee G \ \triangleq\ \text{etc.}$$

A formula F is valid, written $\models F$, iff it is true for all behaviors.

TLA:
The Raw Logic is wonderfully simple, but it is too expressive. It allows you to assert that something is true of the next state, which ruins any effort to heierarchically refine programs. We define TLA to be the subset of Raw Logic formulas obtained by application of $\Box$ and logical operators starting not from arbitrary actions, but from predicates and actions of the form $[A]_f$. For example:

$$P \Rightarrow \neg\Box\neg\Box[A]_f \ \vee\ \Box(Q \Rightarrow \Box[B]_g)$$

Observe that $\Box[A]_f$ asserts that every step is either an A step or else leaves f unchanged.

As is usual in temporal logic, we define $\Diamond$ and $\leadsto$ by

$$\Diamond F \ \triangleq\ \neg\Box\neg F$$
$$F \leadsto G \ \triangleq\ \Box(F \Rightarrow \Box G)$$

The Raw formula $\Diamond A$ is a TLA formulas iff A is a predicate or an action of the form $\Diamond <A>_f$.

Technical point. Since $\Box F \wedge \Box G = \Box(F \wedge G)$ holds for any F and G, it's convenient to let TLA include formulas of the form $\Box(P \wedge \Box[A]_f)$ where P is a predicate.

Introduction

This is a relative completeness proof for TLA, a la Cook. It is not a completeness result for all of TLA, just for the class of formulas that one is interested in proving. The formulas we're interested in are of the form

$$\text{Program} \Rightarrow \text{Property}$$

A Program has the form

$$P \ \land\ \Box[A]_{f} \ \land\ \text{Fairness}$$

for P a predicate. So far, all the Fairness conditions have have been conjunctions of the form $SF_{f}(B)$ or $WF_{f}(B)$, where B implies A and

$$WF_{f}(B) \triangleq \Box\Diamond\langle B\rangle_{f} \ \lor\ \Box\Diamond\neg\text{Enabled}(\langle B\rangle_{f})$$
$$SF_{f}(B) \triangleq \Box\Diamond\langle B\rangle_{f} \ \lor\ \Diamond\Box\neg\text{Enabled}(\langle B\rangle_{f})$$

The theorem allows the more general class of programs in which Fairness is the conjunction of formulas of the form

$$\Box\Diamond\neg\text{TACT} \ \lor\ \Diamond\Box\text{TACT} \quad \text{or} \quad \Box\Diamond\neg\text{TACT} \ \lor\ \Box\Diamond\text{TACT},$$

where TACT denotes any formula of the form $Q \land [B]_{g}$, so $\neg$TACT is a formula of the form $Q \lor \langle B\rangle_{g}$. The Fairness formula must satisfy the additional requirement that program is machine closed, meaning that for any safety property S:

$$\text{If } \models (P \ \land\ \Box[A]_{f} \ \land\ \text{Fairness}) \Rightarrow S$$
$$\text{then } \models (P \ \land\ \Box[A]_{f}) \Rightarrow S$$

(The theorem requires this only when S is of the form $\Box$TACT.) Machine closure, which was called "feasibility" by Apt, Francez, and Katz, is a reasonable requirement for any fairness condition. It can be argued that a condition not satisfying it is not a fairness condition, since it can't be implemented by a memory-less scheduler.

The Property can have any of the following forms:

Predicate
$\Box$Predicate
Predicate $\leadsto$ Predicate
GeneralProgram

where a GeneralProgram is like a Program, except without the machine-closure requirement on its fairness condition. The absence of this requirement is important, for the following reason. To prove that program Π_1 implements program Π_2, one proves $\Pi_1 \Rightarrow \Phi_2$, where Φ_2 is obtained from Π_2 by substituting state functions for variables. This substitution preserves the form of the formula Π_2, but can destroy machine-closure.

Proving relative completeness for safety properties in TLA is pretty much the same as proving it for the Floyd/Hoare method. The completeness results for Hoare's method assumes the expressibility of the predicate sp(S, P) for program statements S and predicates P, where sp is the strongest postcondition operator. Assuming such predicates for arbitrary statements S, which include loops or recursion, is equivalent to assuming the expressibility of sp(A, P) and sin(A, P) for atomic actions A, where sin is the strongest invariant operator.

Proving relative completeness for liveness is somewhat trickier. It requires induction over well-founded sets. Taking a simple, intuitive approach leads to a result whose practical interest is rather doubtful. For example, Mann and Pnueli ("Completing the Temporal Picture") use the axiom of choice to pull a well-founded ordering on the state space out of a hat. Such a construction requires the assumption that every semantic predicate is syntactically expressible.

Getting the precise expressibility assumptions, and avoiding mistakes, required a careful formal exposition.

The Assumptions

In relative completeness results for Hoare logic, one assumes a complete system for reasoning about predicates. In TLA, all the serious reasoning is in the domain of actions. So, we assume a complete system for reasoning about actions. More precisely, letting $\vdash$ denote provability, we assume a set ACT of expressible actions such that $(\models A) \Rightarrow (\vdash A)$ for any action A in ACT. There are various simple assumptions about ACT--such as its being closed under boolean operations. Let PRED denote the set consisting of all predicates in ACT (remember that a predicate is an action that doesn't mention unprimed variables). The least reasonable assumption is that for any P in PRED and A in ACT, sin(A, P) and sp(A, P) are in PRED. Of course, this assumption is what really puts the "relative" in "relative completeness".

The relatively complete logical system consists of the following:

1. The usual assortment of simple propositional temporal logic rules and axioms that you'd expect, since TLA includes simple temporal logic (the logic that's the same as the Raw Logic except starting with predicates, not arbitrary actions).

2. An induction principle, which is what you'd expect for any relatively complete system for proving temporal logic liveness properties.

3. The two TLA axioms:

   $$\vdash (\Box P \equiv P \ \wedge \ \Box [P \Rightarrow P']_P)$$

   $$\vdash (A \Rightarrow B) \ \Rightarrow \ \vdash (\Box A \Rightarrow \Box B)$$

   where P is a predicate, and A and B are actions of the form $P \wedge [A]_f$.

The axioms of 3 are the only ones that mention actions. The axioms of 1 only mention arbitrary formulas, and the induction principle of 2 talks only about predicates. These axioms are actually valid for the Raw logic, and in that logic the second axiom of 3 is a special case of the axiom

$$\vdash (F \Rightarrow G) \ \Rightarrow \ \vdash (\Box F \Rightarrow \Box G)$$

from 1, for arbitrary formulas F and G. However, $[A]_f \Rightarrow [B]_g$ isn't a TLA formula (it's a formula in the logic of actions, but not in TLA), so the second axiom of 3 is needed if you want to do all your reasoning completely within TLA.

The induction axiom 2 is tricky enough to be worth mentioning. To get it right, we first have to generalize everything to include logical variables. If you want to describe an n-process algorithm with a TLA formula, for an arbitrary but fixed n, then n is a logical variable of the formula. A logical variable represents an unspecified value that is the same for all states of a behavior. In the semantics of actions and TLA formulas, Booleans have to be replaced by Boolean-valued formulas involving logical variables. (Formally, Booleans become boolean-valued functions on interpretations, where an interpretation is an assignment of values to all logical variables.) Logical variables pop up all the time when you use TLA in practice. For example, if you have a distributed algorithm with a set Node of nodes, then Node is a logical variable. In fact, if you go really overboard in formalism--as you must to verify things mechanically--then everything that's not a program variable (the kind of variable I first talked about) or a logical operator is a logical variable. In the expression x + 3, the + and the 3 are logical variables. We just happen to have a lot of axioms about the logical variables + and 3, such as 1+1+1 = 3, while we have just a few axioms about the logical variable n (for example $n \in NAT$, n > 0).

But, I digress. I was talking about the induction principle. An induction principle involves induction over a well-founded ordering on a set. Intuitively, a well-founded ordering on a set S is a relation > such that there does not exist an infinite sequence $c_1 > c_2 > c_3 > \ldots$. More precisely,

$$\begin{aligned}&\text{Well-Founded}(>, S)\\ &\quad \triangleq \neg \forall i : (i \in NAT) \Rightarrow\\ &\qquad\qquad \exists c_i : (c_i \in S) \wedge (c_i > c_{i+1})\end{aligned}$$

But, what does this formula mean? For me, the most sensible way to interpret it as a logical formula is to rewrite it as

$$\begin{aligned}&\text{Well-Founded}(v > w, S)\\ &\quad \triangleq \forall c : \neg \forall i : (i \in NAT) \Rightarrow\\ &\qquad\qquad (c(i) \in S) \wedge (c(i) > c(i+1))\end{aligned}$$

where v > w is a formula with free logical variables v and w, and (c(i) > ci+1)) is the formula obtained by substituting c(i) for v and c(i+1) for w in the formula v > w. This is a higher-order formula, involving quantification over a function symbol c.

The completeness result requires, as an assumption, that if the formulas "v > w" and "$v \in S$" are expressible, then Well-Founded(v > w, S) is provable if it's true. I think that if you look closely at Manna and Pnueli's paper, you'll find that they are implicitly assuming this for any formula "v > w"--not just for an expressible one.

Anyway, the actual temporal induction principle looks as follows, where P(w) denotes a formula containing w as free logical variables, P(v) denotes the result of substituting v for w, and F is an arbitrary temporal formula.

$$\begin{aligned}&\text{If } \vdash \exists w \in S\\ &\quad w \text{ not free in } F\\ &\quad \vdash \text{Well-Founded}(>, S)\\ &\quad \vdash (F \wedge w \in S\\ &\qquad\quad \Rightarrow (P(w) \leadsto \exists v : (v \in S) \wedge (w > v) \wedge P(v)))\\ &\text{then } \vdash \neg F\end{aligned}$$

I've actually lied a bit. I assume this rule when w is a k-tuple of distinct logical variables, and I assume the provability only of Well-Founded(v > w, VAL^k), where v > w is an expressible formula and VAL^k denotes the set of k-tuples of values. I could have done it the other way by making a few more expressibility assumptions--such as assuming that "v $\in$ VAL^k" is expressible--but I think that would have been a little more complicated.