Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 892832, 9 pages
http://dx.doi.org/10.1155/2014/892832

Research Article

Formal Proof of a Machine Closed Theorem in Coq

Hai Wan,[1] Anping He,[2] Zhiyang You,[3] and Xibin Zhao[1]

[1] *School of Software, Tsinghua University, NLIST, KLISS, Beijing 100084, China*
[2] *Guangxi Key Laboratory of Hybrid Computation and IC Design Analysis, Guangxi University for Nationalities, Nanning, Guangxi 530006, China*
[3] *General office of the Guangxi Zhuang Autonomous Region People's Government, Nanning, Guangxi 530003, China*

Correspondence should be addressed to Anping He; hapetis@gmail.com

Received 7 January 2014; Accepted 11 February 2014; Published 30 March 2014

Academic Editor: X. Song

The paper presents a formal proof of a machine closed theorem of TLA$^+$ in the theorem proving system Coq. A shallow embedding scheme is employed for the proof which is independent of concrete syntax. Fundamental concepts need to state that the machine closed theorems are addressed in the proof platform. A useful proof pattern of constructing a trace with desired properties is devised. A number of Coq reusable libraries are established.

1. Introduction

TLA$^+$ [1, 2] is a formal specification language for describing and reasoning about distributed and concurrent systems. It is based on mathematical logic, set theory, and linear time temporal logic TLA [3]. TLA$^+$ is widely used to write precise and formal specifications of discrete systems. The notion of machine closed plays an important role in the system specification. Generally speaking, a specification consists of two parts: one is a safety part and the other is a liveness part. The specification is called machine closed if the liveness part does not constrain the safety part. In [1], it is said that "*we seldom want to write a specification that isn't machine closed. If we do write one, it's usually by mistake*." Hence, we need to check whether the specification is machine closed. Fortunately, there is a well-known theorem, that is, machine closed theorem [4], stating that all the TLA$^+$ specifications are machine closed. More precisely, a TLA$^+$ specification which consists of a transition system and a possibly countable infinite fairness constraints is machine closed. Hence, in TLA$^+$ there is no need to verify the specification to see whether it is machine closed. In other words, TLA$^+$ specifications are constructed to be machine closed.

We are now working on formalizing a subset of TLA$^+$ in the theorem prover Coq. As an important part, we want to have the machine closed theorem in our formalization. There are mainly two ways to embed the theorem: one is to have it as an axiom and the other is to state it as a theorem and prove it. We think the second way is better. The reason is twofold:

(i) it needs to be very careful to introduce an axiom into a proof system, since sometimes the introduction of a new axiom may result in inconsistence;

(ii) it is worthwhile to have a formal proof of the theorem, though it is well-known. The proof will help to understand and check the previous formalizations (i.e., the fundamental definitions needed to state the theorem) and make the whole formalization more solid.

In this paper, we present a formal proof of a machine closed theorem in the theorem prover Coq. In order to do so, various fundamental definitions, such as traces, properties, safety property, liveness property, safety closure, and machine closed are given. Based on these definitions, the theorem is formally stated. The proof follows [4]. The key part of the proof is to find a strategy which can generate a trace with some desired properties. We designed a reusable proof pattern to guide this process. Besides this proof, the pattern is also used in proving whether a property is a liveness property; for example, both strong and weak fairness constraints are proved to be liveness properties using this pattern. The whole proof is done in a shallow-embedded manner, which makes the formalization independent of any concrete syntaxes. In other words, this work can be reused in other settings.

The rest of the paper is organized as follows. In Section 2, some preliminaries and a short introduction of TLA$^+$ are given. Section 3 presents the formalization of the machine closed theorem in Coq. The detailed proof is described in Section 4. Related work and concluding remarks are given in Section 5.

2. Preliminaries

2.1. TLA$^+$ Specifications. The TLA$^+$ specification is a formula of the form $Init \wedge \Box[Next]_v \wedge L$, where v is a tuple usually containing all state variables in $Init$, $Next$, and L [4]. The first conjunct $Init$ describes the possible initial states of the system. The second conjunct of the specification asserts that every step (i.e., every pair of successive states in the system) either satisfies $Next$ or leaves the variables in v unchanged. Allowing such stuttering steps is a key ingredient to obtain compositionality of specifications. However, it also means that executions that stutter infinitely are allowed by the specification. The third conjunct L is a temporal formula stating the liveness constraints of the specification, and particularly it can be used to rule out infinite stuttering. The part $Init \wedge \Box[Next]_v$ is known to be the safety part of the specification while L to be the liveness part. The machine closed theorem states that $(Init \wedge \Box[Next]_v, L)$ is machine closed.

2.2. Definitions. We fix the set of states as St. St^* denotes the set of finite sequences of states, and St^ω is the set of infinite sequences of states. A sequence is also called a *trace* in this paper. The ith state in trace t is denoted by t_i. $|t|$ denotes the *length* of t, if t is finite.

A *property* is a set of infinite traces. Given a property P, we use $t \vDash P$, which means trace t satisfies property P, to denote $t \in P$. Given a finite trace $t \in St^*$, if $\exists t' \in St^\omega . t \circ t' \vDash P$ ($\circ$ is the traditional trace concatenation operator), then we say $t \vDash P$. $t_{[\cdots n]}$ and $t_{[n\cdots]}$ denote the prefix $(t_0 \cdots t_n)$ and suffix $(t_n t_{n+1} \cdots)$ of t, respectively. Following [5], a property P is a *safety property*, if $\forall t \in St^\omega . Pt \leftrightarrow \forall i . \exists t' \in St^\omega . t_{[\cdots i]} \circ t' \vDash P$. A property P is a *liveness property*, if $\forall t \in St^* . \exists t' \in St^\omega . t \circ t' \vDash P$.

An *action* a is a predicate over two states. Action a is a *subaction* of action a' if $\forall s\, s' . a\, s\, s' \rightarrow a'\, s\, s'$.

Following [6], we have the definitions of safety closure and machine closed. Given an arbitrary property P, its *safety closure* $C(P)$ is defined as $\{t \in St^\omega \mid \forall i . t_{[\cdots i]} \vDash P\}$. It can be proved that $C(P)$ is the smallest safety property containing P.

Definition. A pair of properties (S, L) is said to be *machine closed* if $C(S \cap L) = S$ and S is a safety property.

3. Formalization of Machine Closed Theorem

We work on the semantical level, in other words, we define all the concepts described in Section 2.2 in a shallow embedding manner. Throughout this section, we first describe the notions informally then give the corresponding Coq scripts.

3.1. Help Definitions. In the following scripts, we first define the set of state as St. A trace is a function of type $nat \rightarrow St$ (There are other ways to define traces. E.g., [7] uses coinductive data types to define infinite traces and [8] uses coinductive data types to define both infinite and finite traces. The reasons why we adapt the "function type" one is twofold: (i) the "function type" trace is intuitive; (ii) we only consider infinite traces, and we want to separate the finite traces and infinite traces at the data type level. In the subsequent section, finite traces are defined separately using inductive data type.). So given a trace t and a natural number i, $(t\, i)$ denotes the ith state of t (count from 0). *PredOn*1 and *PredOn*2 are the types of predicates over one state and two states, respectively. *PredOn*1 is the state predicate type while *PredOn*2 is the action type. A property is a set of traces, whose type is $Trace \rightarrow Prop$. $UptoN\, t\, t'\, n$ means traces t and t' are identical up to index n. $UptoN\_e\, t'\, t$ means traces t and t' are identical up to index $(n - 1)$. $Offsettn$ returns $t_{[n\cdots]}$. *State*1*toProperty* (*State*2*toProperty*, resp.) makes a trace predicate(i.e., a property) from a one-state (two-state, resp.) predicate. In order to make the representation more concise, we define three coercions which implicitly changes a one-state or a two-state predicate to properties, respectively.

```
Variable St: Set.

Definition Trace := nat -> St.

Definition PredOn1 := St -> Prop.

Definition PredOn2 := St -> St -> Prop.

Definition Property := Ensemble Trace.

Definition UptoN (t t': Trace) (n:nat):
Prop :=

  forall i, i <= n -> t i = t' i.

Definition UptoN_e (t t': Trace)
(n:nat): Prop :=

  forall i, i < n -> t i = t' i.

Definition OffsetN (t:Trace) (n:nat):
Trace :=

  fun i => t (n+i).

Definition State1toProperty (p:
PredOn1): Property :=

  fun t => p (t 0).

Definition State2toProperty (p:
PredOn2): Property :=

  fun t => p (t 0) (t 1).

Definition Actions := PredOn2.


Coercion State1toProperty: PredOn1 >->
Property.

Coercion State2toProperty: PredOn2 >->
Property.

Coercion Action2toProperty: Actions >->
Property.
```

In the following scripts, we have the traditional definitions of "finally," "always," "infinite often," "finally always," "enabled," "strong fairness," and "weak fairness."

```
Definition F (p: Property) (t:Trace):=
  exists n, p (OffsetN t n).
Definition G (p: Property) (t:Trace):=
  forall n, p (OffsetN t n).
Definition FG p := F (G p).
Definition GF p := G (F p).
Definition En (p2: PredOn2) (s: St) :=
exists s', p2 s s'.
Definition SF_Action (a: Actions):
Property :=
  fun t => GF (En a) t -> GF a t.
Definition WF_Action (a: Actions):
Property :=
  fun t => FG (En a) t -> GF a t.
```

3.2. Safety, Liveness, and Machine Closed. In the definitions of safety property, liveness property and safety closure, the notions of "finite trace" and "concatenation of a finite trace and an infinite trace" are used. Since we choose the function type to present traces, it is not very convenient to express finite traces. In order to avoid the notions of "finite trace" and "concatenation," we should change the definitions to their equivalent counterparts which only involve infinite traces.

Safety Property. We change "$\forall t \in S^{\omega}.Pt \leftrightarrow \forall i.\exists t' \in S^{\omega}.t_{[\cdots i]} \circ t' \vDash P$" to its equivalent formula "$\forall t \in S^{\omega}.Pt \leftrightarrow \forall i.\exists t' \in S^{\omega}.(UptoN\, t\, t'\, i) \wedge t' \vDash P$."

```
Definition IsSafetyProperty (p:
Property) :=
  forall t, p t <-> (forall i, exists
  t', UptoN t t' i /\ p t').
```

Liveness Property. We change "$\forall t \in S^{*}.\exists t' \in S^{\omega}.t \circ t' \vDash P$" to "$\forall t \in S^{\omega}.\forall i.\exists t' \in S^{\omega}.UptoN\_e\, t\, t'\, i \wedge t' \vDash P$" and have the following definition:

```
Definition IsLivenessProperty (p:
Property) :=
  forall t i, exists t', UptoN_e t t'
  i /\ p t'.
```

Safety Closure. "$\{t \in S^{\omega} \mid \forall i.t_{[\cdots i]} \vDash P\}$" is changed to "$\{t \in S^{\omega} \mid \forall i.\exists t'.UptoN\, t\, t'\, i \wedge t' \vDash P\}$".

```
Definition SafetyClosure (p: Property):
Property :=
  fun t => forall i, exists t', UptoN t
  t' i /\ p t'.
```

We use the notation mechanism of Coq to make the representations more succinct. $p\,[/\backslash]\,q$ denotes that the intersection of properties p and q. $p\,[<=]\,q$ expresses the fact that property p is a subset of property q. $p\,[=]\,q$ describes the fact that property p is equal to property q. $[C]p$ is the safety closure of p. $[S?]$ is a shorthand of predicate *IsSafetyProperty* while $[L?]$ is a shorthand of predicate *IsLivenessProperty*.

Machine Closed. It is defined as

```
Definition MachineClosed (s
p:Property): Prop :=
  [C] (s [/\] p) [=] s /\ [S?] s.
```

3.3. Transition System. As described in Section 2.1, the specification of a system is of form $Init \wedge \Box[Next]_v \wedge L$. Guided by this form, a transition system is parameterized by

(1) the set *St* of states,

```
Variable St: Set.
```

(2) predicates *Init* of type *PredOn*1*St* and *Next* of type *PredOn*2*St* characterizing the initial states and next-state relation, respectively, where we require that (`Parameter` is a synonym of `Variable` in Coq.): every state has a successor according to *Next*; in other words, *Next* is total.

```
Parameter Init: PredOn1 St.
Parameter Next: PredOn2 St.
Hypothesis next_input_enabled: forall
s, exists s', Next s s'.
```

(3) two sets of indexes: I and J. $I \cup J$ is finite or countable infinite and each $k \in I \cup J$, action a_k is a subaction of *Next*. Each $i \in I$ ($j \in J$, resp.), action a_i (a_j, resp.) is associated with a strong (weak, resp.) fairness constraint.

```
Parameter Acts_S: nat -> Actions St.
Parameter Acts_W: nat -> Actions St.
Definition Acts_SF := fun n =>
SF_Action _ (Acts_S n).
Definition Acts_WF := fun n =>
WF_Action _ (Acts_W n).
Hypothesis Acts_subaction_of_Next_SF:
  forall i s s', (Acts_S i) s s' ->
  Next s s'.
Hypothesis Acts_subaction_of_Next_WF:
  forall i s s', (Acts_W i) s s' ->
  Next s s'.
```

In the scripts, we use a function of type $nat \rightarrow Actions\ St$ to represent the set of actions. Note that the cases where the set of actions is finite or countable infinite are covered by the definitions of *Acts_S* and *Acts_W*. The essential point is that both *Acts_S* and *Acts_W* are of type $nat \rightarrow Actions\ St$. Suppose the set of I is finite, say its cardinality is N, we can build *Acts_S* as follows: for each $i < N$, map *Acts_S i* to the $(i+1)$th action in I; for each $i \geq N$, map *Acts_S i* to $(fun\, s\, s' \Rightarrow False)$. If the set of I is countable infinite, then for each $i \in \mathbb{N}$, map *Acts_S i* to the $(i+1)$th action.

Based on the above definitions, we can build the safety part *sp* and the liveness part *lp* for the transition system:

(i) *sp*: all traces allowed by *Init* and *Next*.

```
Definition sp :=
  fun (t:Trace _) => Init (t 0)/\
  forall n, Next (t n)(t (S n)).
```

(ii) *lp*: all traces allowed by the set of fairness constraints.

```
Definition lp: Property _ :=
  fun t => forall n, Acts_SF n t /\
  Acts_WF n t.
```

Finally, we get the theorem to prove

```
Theorem machine_closed:
  MachineClosed _ sp lp.
```

4. Proof of Machine Closed Theorem

The proof follows the proof of Proposition 4 in [4].

Given a specification of a transition system, $Init \wedge \Box Next \wedge L$ where $L = \forall i \in I.WF(a_i) \wedge \forall j \in J.SF(a_j)$, $\forall i \in I.a_i$ is a subaction of *Next*, $\forall j \in J.a_j$ is a subaction of *Next* and $I \cup J$ is finite or countable infinite, we need to prove that $(Init \wedge \Box Next, L)$ is machine closed. Based on the discussion in Section 3.3, here we prove a more general case where both I and J are countable infinite. Hence, we can take I and J to be $\mathbb{N}$.

The whole proof is divided into two steps. First, we prove that a stronger version of the specification, in which all the weak fairness constraints are changed to their strong fairness counterparts, is machine closed. Second, we prove the original specification is machine closed.

4.1. The Stronger Specification. To build a stronger specification, first we need a mapping $f : nat \to Actions\ St$, which has the property that $ran(Acts\_S) \cup ran(Acts\_W) = ran(f)$, where $ran\ \bullet$ is the range of function $\bullet$. Informally speaking, through f we can get all the actions used in the original specification. In the following scripts, *even_odd_dec* is deployed to test whether a natural number is even and $div2\ n$ returns $n/2$.

```
Definition f (n:nat): Actions St :=
  match (even_odd_dec n) with
  |left _ => Acts_S (div2 n)
  |right _ => Acts_W (div2 n)
  end.
```

Then we can build the stronger specification that is equivalent to the original one except for

(i) the fairness constraint *lp* is replaced by *lp_stronger*. As we can see in the following scripts, each action is associated with a strong fairness constraint. It differs from the original one in which some actions are associated with weak fairness constraints, while the other with strong fairness constraints.

```
Definition lp_stronger := fun t =>
forall n, SF_Action _ (f n) t.
```

(ii) Based on assumptions *Acts_subaction_of_Next_SF* and *Acts_subaction_of_Next_WF*, the following theorem is proved:

```
Theorem Acts_subaction_of_Next:
  forall i s s', (f i) s s' ->
  Next s s'.
```

Since all the actions are indexed by a natural number and accessible through the number using f, in the sequel, we will use a natural number to represent the action: phrase "action i" is equivalent to "action $f(i)$."

The intermediate theorem about the stronger specification is

```
Theorem sp_lp_stronger_machine_closed:
  MachineClosed _ sp lp_stronger.
```

Following the definition of machine closed, we need to prove

(1) *sp* is a safety property;

(2) $C(sp \cap lp\_stronger) = sp$. There are two directions:

(1.1) $C(sp \cap lp\_stronger) \subseteq sp$;

(2.2) $sp \subseteq C(sp \cap lp\_stronger)$.

The set of all runs of the transition system is a safety property. $Init \wedge \Box[Next]$, which is equal to *sp*, defines a transition system. Hence, condition (1) is proved. Condition (2.1) can be proved from the fact that $sp \cap lp\_stronger \subseteq sp$ and the following theorem:

```
Theorem SafetyClosureSmallest:
  forall p p', p[<=]p' -> [S?] p' ->
  ([C] p) [<=] p'.
```

In order to prove condition (2.2), based on the definition of safety closure, it is sufficient to prove that for each $t \in sp$ and $n \in \mathbb{N}$, $t_{[\cdots n]}$ can be extended to an infinite trace t' such that $t' \in sp \wedge t' \in lp\_stronger$. As described in [4], given $t_{[\cdots n]}$ we need to construct a trace-generate strategy g which can generate a trace t' and t' have properties: $t'_{[\cdots n]} = t_{[\cdots n]} \wedge t' \in sp \wedge t' \in lp\_stronger$.

4.1.1. Trace-Generate Strategy. Roughly speaking, a trace-generate strategy is of type $S^* \to S$, which takes a finite trace as input and returns a state. In other words, a trace-generate strategy defines a scheduling policy which returns a next state based on the state sequence the system already produced. We first define the finite trace data type *FiniteTrace* (*list* is a predefined type in Coq: Inductive list (A: Type): Type := nil: list A—cons: A → list A → list A. The concatenation of an element a of type A and a list l of type $list\ A$ is denoted by $a :: l$.).

```
Definition FiniteTrace := list St.
```

There are two ways to obtain a strategy function: (1) define it as a function of type $FiniteTrace \rightarrow St$ directly; (2) first define a relation of type $FiniteTrace \rightarrow St \rightarrow Prop$ and then derive a function of type $FiniteTrace \rightarrow St$ from the relation using the classical choice axiom. In our case, we choose the second way, since we only care about the relation. And furthermore defining a relation is easier than defining a function: a function is difficult to define if the computation is complex. In our case, we essentially defined a scheduling policy, which is complex.

In order to use the classic choice axiom, the relation g should be total; that is, for each finite trace t there exists a state s such that $g\,t\,s$. We only consider this kind of strategy relations. The set of all valid strategy relations is captured by *VGenStrategy*.

```
Definition GenStrategy := FiniteTrace
-> St -> Prop.

Definition GenStrategyF := FiniteTrace
-> St.

Definition VGenStrategy :=

  {g:GenStrategy | (forall f, exists
  s:St, g f s)}.
```

A trace t can be derived based on a valid strategy relation *vsr*. The main derivation steps are as follows:

(1) a strategy function f is derived from *vsr* using the choice axiom;

(2) given a strategy function f and a natural number n, prefix $t_{[\cdots n]}$ is calculated recursively by *GSF_to_FiniteTrace*;

```
Fixpoint GSF_to_FiniteTrace
(sf:GenStrategyF)(n:nat) {struct n}

  : FiniteTrace :=

  match n with

  |0 => sf nil :: nil

  |S n' => let t' := GSF_to_FiniteTrace
  sf n' in

      sf t' :: t'

  end.
```

(3) the prefix is always not nil, which means we can always get the last state t_n that is the nth state in t. (*myhead* of type $\forall l : FiniteTrace.l \neq nil \rightarrow St$ returns the head of a finite trace. It requires a proof that the input finite trace is not nil. This fact is provided by theorem *GSF_to_FiniteTrace_notnil*.)

```
Definition GSF_to_Trace (g:
GenStrategyF): Trace :=

  fun n => myhead (GSF_to_FiniteTrace
  g n)

    (GSF_to_FiniteTrace_notnil g n).
```

Finally we get theorem *GenStrategy_to_Trace*, through which we can derive a trace t based on a valid strategy g and t fulfils our constraints that (nil, t_0) and $(t_{[\cdots i]}, t_{i+1})$ (for all $i \in \mathbb{N}$) satisfies the strategy relation. *GetG* is used to get the *GenStrategy* part from a *VGenStrategy* and *GetPrefixN* is used to get the first n states from a *Trace*.

```
Theorem GenStrategy_to_Trace:

  forall g: VGenStrategy, exists
  t:Trace,

    forall n, (GetG g) (GetPrefixN t n)
    (t n).
```

4.1.2. Design a Trace-Generate Strategy. Recall that we need to prove condition (2.2) in the previous subsection: given a trace $t \in sp$ and a position n, a trace t' should be constructed such that three properties (a) $t'_{[\cdots n]} = t_{[\cdots n]}$, (b) $t' \in sp$ and (c) $t' \in lp\_stronger$ are hold. To achieve the goal, firstly, we need to design a trace-generate strategy relation (the relation is designed with the three properties under consideration); secondly, we need to prove the relation is valid; at last, we need to prove the generated trace conforms to the properties. We fix t, n, ft, and s in the sequel.

Define the Relation. The design principles of the relation are as follows:

(1) at any point, the set of schedulable actions should be finite and all actions in the set is enabled;

(2) at any point, if the schedulable action set is not empty, the action that has not been executed for the longest time is scheduled to execute. In event of ties, the action with minimal index is chosen. Principle 1 is the key to this principle, since if the schedulable action set is infinite, there may exist an action which is infinitely enabled but not infinitely executed;

(3) at any point, if the schedulable actions set is empty, then pick *Next* to execute since *Next* is total;

(4) $t_{[\cdots n]}$ should be a prefix of the generated trace.

The strategy is defined as:

```
Definition MC_strategy (t:Trace _)
(n:nat) :=
  fun (ft:FiniteTrace St)(s:St) =>
    match ft with
    |nil => s = t 0
    |cons s' tl =>
      (0<length ft<=n /\t (length ft)=s)
      \/
      (n<length ft /\ (forall i,~
      MC_Enabled ft s' i)/\Next s' s)

      \/(n<length ft /\exists i,
      MC_Enabled ft s' i /\

      MC_theMin ft s' i /\ (Acts i) s' s)
  end.
```

Intuitively "$(MC\_strategy\ t\ n)\ ft\ s$" has the following implicit meanings:

(i) ft is the finite trace that the system has already produced;

(ii) if $(MC\_strategy\ t\ n)\ \ ft\ s$ holds, then s is the next state the system will generate.

Depend on the length of ft, there are 4 cases of how the strategy generates a next state: (a) ft is nil; (b) $|ft| \leq n$; (c) $|ft| > n$ and all the actions are not enabled; (d) $|ft| > n$ and there is an enabled action.

$MC\_theMin\ ft\ s'\ i$ denotes that the action i has the properties mentioned in the second principle. In order to define MC_theMin, there are several concepts that need to be represented: (a) given a finite trace and an action, the number of steps that the action continuously has not been executed up to now (*nsteps*); (b) notion of the largest number of nonexecuted steps ($MC\_theLongestEnabled$); (c) the smallest index with the largest nonexecuted number MC_theMin. Given a finite trace ft, a nat i, and an action predicate a, $NoExeOfActionUpTo\ \ ft\ i\ a$ returns the number of executable actions of ft.

```
Fixpoint nsteps (ft:FiniteTrace St)
(i:nat)
  (a: Actions St) {struct ft}: nat :=
  if lt_le_dec i (length ft) then
    match ft with
    |nil => 0
    |hd :: nil => 0
    |hd :: (hd'':: _) as tail =>
      if action_dec a hd' hd then 0
      else S (NoExeOfActionUpTo tail
      i a)
    end
  else 0.
Definition MC_theLongestEnabled (ft:
FiniteTrace _) s idx :=
  MC_Enabled s idx /\
  (forall idx', MC_Enabled s idx' ->
    nsteps ft idx'(f idx')<=nsteps ft
    idx(f idx).
Definition MC_theMin (ft: FiniteTrace
_) s idx :=
  MC_theLongestEnabled ft s idx /\
    (forall idx', MC_theLongestEnabled
    ft s idx'->idx <= idx').
```

nsteps needs more descriptions. For its arguments, ft is the finite trace, i is the index of the action and a is the action. lt_le_dec returns *true* if and only if its first argument is less than the second argument. The first if-statement is the key to make the schedulable action set finite, but it does so in an implicit way, which will be explained in section "*Prove the Properties*." $action\_dec\ a\ hd'\ hd$ returns *true* if and only if $a\ hd\ hd'$ holds.

Useful Theorems. In the sequent proofs, we want to choose an element with a specific property from a set, for example, the minimal or the maximal element from a set according to some order. We use theorems to do so and these theorems are of a similar pattern: given a set, if the set is not empty, then there exists an element in the set with some specific properties.

(i) *Get the Minimal Element*. The following theorem states that given a set P, if P is not empty and R is a well-founded order, then there exists a minimal element in P according to order R.

```
Theorem ExistTheMin_general:
  forall (A:Set)(R: A -> A -> Prop)
  (P: Ensemble A),
    well_founded R ->(~ Empty_set _ P)
    ->
      (exists n, In _ P n /\ (forall i,
      In _ P i ->~ R i n)).
```

(ii) *Get the Maximal Element*. The following theorem expresses that given a set P of natural numbers, if there exists a natural number *max* which is larger than all the elements in P, then there exists a maximal element in P.

```
Theorem ExistTheMax_nat:
  (~ Empty_set _ P) ->
    (exists max, (forall n, In _ P
    n -> n <= max)) ->
      exists m, In _ P m /\ forall n,
      In _ P n -> n <= m.
```

These theorems can be thought as a kind of element choosers, which can be used to pick up a specific element from a set. Combining these choosers with proper initializations of the set predicates (i.e., P in the theorem), we can obtain some useful complex choosers, such as the one choosing the smallest index among the indexes whose associated actions have not been executed for the longest time.

(iii) The following theorem represents that given P, a finite set of natural numbers, and F, a relation over two natural numbers, if F is total on P (i.e., for each element in P there exists a number F-related to it), and F is transitive over the second parameter (i.e., for each element in P, if a number is F-related to the element then any number larger than the number is also F-related to the element) then there exists an m such that m is F-related to each element in P.

```
Theorem PFiniteExistsAMax:
  forall (P: Ensemble nat)
  (F: nat->nat->Prop),
```

```
Finite _ P->(forall i, In _ P i ->
exists xi, F i xi) ->
  (forall i xi xi', In _ P i->F i
  xi->xi<=xi'->F i xi') ->
    exists m, forall i, In _ P i ->
    F i m.
```

Prove the Validity. The validity of *MC_strategy* is demonstrated by the following theorem:

```
Theorem MC_strategy_Valid:
  forall t n f, exists s:St, MC_strategy
  t n f s.
```

The theorem is proved by case analysis, which corresponds to the four cases of *MC_strategy*. For the first 3 cases, they are not hard to prove. For the last case, we need to prove the following theorem:

```
Theorem MC_theMin_Valid:
  forall ft s, (exists i, MC_Enabled
  ft s i) ->
    exists idx, MC_theMin ft s idx.
```

This theorem is proved based on the theorems described in the previous subsection.

Prove the Properties. At this point, we have a valid trace-generate relation, based on which we obtain a generated trace. We need to prove that the generated trace conforms to the three properties.

Recall that t' is the generated trace. Property (a) (i.e., ${t'}_{[\cdots n]} = t_{[\cdots n]}$) is ensured by the first and second cases of *MC_strategy*: if ft is nil, then the next state is t_0; else if $|ft| \leq n$, then the next state is $t_{|ft|}$. Property (b) is proved based on the facts that: (1) $t \in sp$; (2) case 3 generates a next state based on *Next*; (3) case 4 generates a state based on an action which is a subaction of *Next*. The last property (c) expresses that for each action if it is infinitely enabled in trace t', then it is also infinitely executed. We prove this by contradiction:

(1) if there is an action that is infinitely enabled but finitely executed, then there is a set of actions that are infinitely enabled but only finitely executed;

(2) properly pick an action a from the set and a position n such that a is not executed in the suffix ${t'}_{[n\cdots]}$;

(3) pick a position $m > n$ such that at m the next action chosen to execute by the scheduler is a, which conflicts with step 2.

In order to make a concise representation, we have the following definitions:

```
Definition NotAfter t a n :=
  forall i, n<=i -> ~a t[i] t[S i].
Definition InfEn t a :=
  GF (En a) t.
```

For the second step, corresponding to *MC_strategy*, we choose action idx and n such that (n, idx) is the smallest element in $\{(n, idx) \mid idx <= n + 1 \wedge InfEn\, t'(f\, i) \wedge NotAfter\, t'(f\, i)\, n\}$ (the order between two pairs $p_1 = (a_1, b_1)$ and $p_2 = (a_2, b_2)$ is the classical lexicographic order: $p_1 < p_2 \triangleq a_1 < a_2 \vee (a_1 = a_2 \wedge b_1 < b_2)$), that is, the smallest idx with the smallest n. The first conjunct is used to ensure that we only consider the actions whose indexes are less than the length of ft. This constraint corresponds to the first if-statement in *nsteps*. The second conjunct guarantees that there always is an arbitrary large position at which the action is enabled. It can be proved that the set is not empty based on the assumption in step 1. Again, there is a chooser theorem for this operation.

Intuitively there is some point after n where action idx is the next action to execute, because it has not been executed since n and its index is the smallest. Now we need to find such a point m at which:

(1) for each action $idx' \leq n$, one of the following cases holds:

  (a) if idx' is infinitely enabled and infinitely executed, then $nsteps\, t_{[\cdots m]} idx' < nsteps\ t_{[\cdots m]} idx$;

  (b) if idx' is infinitely enabled but finitely executed, then

    (i) if action idx' is executed at least once after n, then $nsteps\, t_{[\cdots m]}$idx' $< nsteps\, t_{[\cdots m]} idx$;

    (ii) if action idx' is not executed after n, then $nsteps\, t_{[\cdots m]} idx' \leq nsteps\, t_{[\cdots m]} idx$;

  (c) if idx' is finitely enabled, then idx' is not enabled at m;

(2) for each action $idx' > n$, $nsteps\, t_{[\cdots m]} idx' \leq nsteps\, t_{[\cdots m]} idx$ and $idx < idx'$;

Suppose we find m satisfying both conditions. The theorem *PropertyOfM* (we omit the preconditions, since they are too many) states that idx is the smallest index with the longest nonexecute steps (i.e., $MC\_theMin\, t'[\cdots m] t'[m]\, idx$), hence only the fourth case of *MC_strategy* can be true. Based on the uniqueness of *MC_theMin*, we can infer that action idx is the action the scheduler chooses to execute at m.

```
Lemma PropertyOfM:
  ··· ->
    MC_theMin t' [···m] t' [m] idx.
Theorem MC_theMin_Unique:
  forall ft s i i', MC_theMin ft
  s i -> MC_theMin ft s i' ->
  i=i'.
```

In order to obtain such m, we first construct that m' s.t. condition (1) holds and then construct that m s.t. both conditions hold. We use theorem *PFiniteExistsAMax* to get m'. In the theorem P is the set of indexes less than $(n+1)$ and F is defined as

```
Definition F := fun i xi =>
  let inf_en := (GF St (En St (Acts i))
  t') in
```

```
let inf_exe := (GF St (Acts i) t') in
let tp := (GetPrefixN St t' (S xi)) in
let a := Acts idx in
let a' := Acts i in
n <= xi /\
((inf_en /\ inf_exe /\ nstep tp i a' <
nstep tp idx a) \/
(inf_en /\~ inf_exe /\
(((exists n, min_n <= n /\ a' (t' n)
(t' (S n)))/\
  nstep tp i a' < nstep tp idx a) \/
((forall n, min_n <= n ->~a' (t' n)
(t' (S n)))/\
  nstep tp i a' <= nstep tp idx a)))
  \/
(~ inf_en /\ (forall k, xi <= k -> ~
En St a' (t' k)))).
```

The second conjunct consists of four disjuncts, each of which corresponds to a sub condition in condition (1). The total and transitive properties of F are proved by case analysis. By *PFiniteExistsAMax* we obtain the m' which is F-related to all the actions whose indexes are less than $(n + 1)$. For each index $i > n$, by using theorem *greater_less* we know $nstep\, t_{[\ldots m]}\, i\, (f\, i) \leq (m + 1 - i)$, and by theorem *less_greater*, we know $m - n \leq nstep\, t_{[\ldots m]}\, idx\, (f\, idx)$; hence, we know m also holds for condition (2). Thus m is the position we want—property (c) holds on trace t'.

```
Lemma greater_less:
  forall ft i a, nstep ft i a <= length
  ft - i.
Lemma less_greater:
  forall t idx a n, idx <=S n ->
  (forall j, n<=j -> ~a (t j)
  (t (S j))) ->
    forall k, n < k -> k-n<=nstep
      (GetPrefixN St t (S k)) idx a.
```

Finally, we prove that the stronger specification is machine closed.

4.2. The Original Specification. Based on theorem *sp_lp_stronger_machine_closed*, we need to prove theorem *machine_closed*. According to the definition of machine closed, the proof is sketched as

(1) prove sp is a safety property;

(2) prove $C(sp \cap lp) = sp$. There are two directions:

(a) $C(sp \cap lp) \subseteq sp$. The proof is similar to the proof of condition (2.1) in Section 4.1;

(b) $sp \subseteq C(sp \cap lp)$. Given t and i, by theorem *sp_lp_stronger_machine_closed* we can get an extended trace t_0 of $t_{[\ldots i]}$ such that $t_0 \in sp \wedge t_0 \in lp\_stronger$. Hence, the only subgoal needed to solve is to prove that t_0 is also in lp. It is sufficient to prove that

(i) for each $i \in I$, t_0 satisfies the strong fairness constraint of action a_i—this holds, since t_0 satisfies the strong fairness constraint of action $a_{f(2*i)}$ which is equal to a_i.

(ii) for each $i \in J$, t_0 satisfies the weak fairness constraint of action a_j—this holds, since t_0 satisfies the strong fairness constraint of action $a_{f(2*j+1)}$ which is equal to a_j and the following theorem which expresses that if a trace satisfies the strong fairness constraint of an action it also satisfies the weak fairness constraint of that action:

```
Theorem SF_imp_WF:
  forall (a:Actions), SF_Action
  a [->] WF_Action a.
```

5. Related Works and Concluding Remarks

The machine closed theorem is first proved in [4]. There is already some work that embeds TLAi n a theorem prover [9], but to our best knowledge, this is the first time that the theorem is formally proved in a theorem prover. There are several other works that concern the definitions of properties. These works can be discussed in two steps. The first step is how traces are represented. In [10], a function of type $nat \rightarrow St$ is chosen, which is the same as our solution. In other works, inductive and/or coinductive types are used [7, 8, 11]. In [12], the authors propose a more general solution, in which they do not commit to a particular formalization of traces; instead, they exploit the module system of Coq and only list the interface of traces. The second step is how the safety and liveness properties are defined. In [7], the safety property is defined as a state invariant of a transition system and the liveness property is not defined formally.

In this paper, we present a formal proof of the the machine closed theorem in theorem prover Coq. Various fundamental definitions, such as traces, properties, safety property, liveness property, safety closure and machine closed, are given. Based on these definitions, the theorem is formally stated and proved. The main proof of machine closed theorem is done using the section mechanism of Coq. This mechanism makes our formalization adaptable. It can be encapsulated into a module or a record. In our case study, we used the module type. The result module is general, since it is at the semantics level (because we do the proof in a shallow embedding manner) and thus is independent of any concrete syntax. This work also results in several reusable Coq libraries. The Coq scripts can be provided upon request.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The paper was supported in part by the 973 Program of China (Grant no. 2010CB328000), the National Natural Science Foundation of China (Grants nos. U1201251 and 61133016), the National 863 Plan of China (Grant no. 2012AA040906), National Natural Science Foundation of Guangxi (Grant no. 2013GXNSFAA019342), GUN Project no. 2012Q017 and the Bagui scholarship Project of Guangxi.

References

[1] L. Lamport, "Specifying Systems," Addison-Wesley, 2002, http://research.microsoft.com/users/lamport/tla/tla.html.

[2] S. Merz, "The specification language *TLA*$^+$," in *Logics of Specification Languages*, pp. 401–451, Springer, Berlin, Gemany, 2008.

[3] L. Lamport, "Temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, 1994.

[4] M. Abadi and L. Lamport, "Old-fashioned recipe for real time," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1543–1571, 1994.

[5] B. Alpern and F. B. Schneider, "Defining liveness," Tech. Rep., Ithaca, NY, USA, 1984.

[6] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991.

[7] S. Coupet-Grimal, "An axiomatization of linear temporal logic in the calculus of inductive constructions," *Journal of Logic and Computation*, vol. 13, no. 6, pp. 801–813, 2003.

[8] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development*, Coq'Art: The Calculus of Inductive Constructions, Springer, Berlin, Germany, 2004.

[9] S. Merz, "Yet another encoding of TLA in Isabelle," Rapport de Recherche, Institut für Informatik, TU München, Germany, 1997.

[10] O. Müller and T. Nipkow, "Combining model checking and deduction for I/O-automata," in *Tools and Algorithms For the Construction and Analysis of Systems*, pp. 1–16, Springer, 1995.

[11] O. Müller and T. Nipkow, "Traces of I/O automata in Isabelle/HOLCF," in *TAPSOFT'97: Theory and Practice of Software Development*, M. Bidoit and M. Dauchet, Eds., vol. 1214, pp. 580–594, 1997.

[12] M. H. Tsai and B. Y. Wang, "Formalization of CTL* in calculus of inductive constructions," in *ASIAN*, pp. 316–330, 2006.