Formalization of CTL* in Calculus of Inductive Constructions[⋆]

Ming-Hsien Tsai[1,2] and Bow-Yaw Wang[1,⋆⋆]

[1] Institute of Information Science
Academia Sinica, Taiwan
[2] Department of Information Management
National Taiwan University, Taiwan

Abstract. A modular formalization of the branching time temporal logic CTL* is presented. Our formalization subsumes prior formalizations of propositional linear temporal logic (PTL) and computation tree logic (CTL). Moreover, the modularity allows to instantiate our formalization for different formal security models. Validity of axioms and soundness of inference rules in axiomatizations of PTL, UB, CTL, and CTL* are discussed as well.

1 Introduction

The management of digital objects in modern information systems has become very sophisticated during past years. In digital rights management, for instance, a digital content may be accessible exclusively for a fixed period of time; if the contract is expired or the content is currently in use, no access will be allowed. Since traditional static usage control models could not express the dynamic authorizations found in these applications, temporal logics are introduced in recent models [19].

The introduction of temporal logics nevertheless induces new problems. Because of the complexity in the semantics of temporal operators, users often have difficulties in writing correct requirements or verifying them. Moreover, specifications of real-world usage control systems are rather complicated. Whether one can analyze such temporal specifications correctly by hand is not without questions.

One way to help users manage complicated specifications is to mechanize the process. Indeed, fully automated approaches such as model checking are able to analyze models against temporal logic specifications without user intervention. But the expressiveness of formal security models deviates from the simplicity of computation models in algorithmic approaches; various capability and computability issues are subsequently arisen in fully automated techniques.

⋆ The work is partly supported by NSC grands 95-3114-P-001-002-Y02, 95-2221-E-001-024-MY3, and the project SISARL of Academia Sinica.

⋆⋆ Part of the the work was done during the second author visited to the project ProVal supported by INRIA Futurs.

In order to have expressive models and circumvent undecidability, semi-automated approaches such as proof checking are used. In semi-automated techniques, usage control models and their temporal logic specifications are formulated in proof assistants. Security amounts to the entailment of respective temporal logic specification. Since each step of the proof is checked by the proof assistant, the correctness of analysis is therefore ensured.

But formulations of models and their specifications require domain knowledge about formal models, temporal logics, and proof assistants. Inappropriate formulations may result in ineffective or even faulty analysis. In this paper, we address the formulation problem of temporal logics in the proof assistant Coq. Specifically, the branching time temporal logic CTL* is formalized in Calculus of Inductive Constructions. We identify assumptions in formal models and modularize our formalization based on these assumptions. Users will be able to instantiate our formalization as long as their formal models conform to the identified assumptions.

The branching time temporal logic CTL* is a proper super class of the propositional linear temporal logic (PTL) and the computation tree logic (CTL). Prior formalizations of PTL and CTL are therefore subsumed by the present work. The expressiveness of CTL* gives users more freedom to specify the requirements of their security models. To the best of our knowledge, ours is the first formalization of CTL* in any proof assistant. Moreover, we have used the formalization to establish the validity of 31 (out of 33) axiom schemata, and the soundness of 8 (out of 10) inference rules in four complete axiomatizations of various temporal logics.

The modularity distinguishes our formalization from others as well. We identify assumptions needed in the formalization of CTL* and formally specify them in a Coq module type. The formalization of CTL* is carried out in a functor from modules of the aforementioned module type. Subsequently, any formalization of security models can instantiate our CTL* formalization, provided it is of the proper module type. In domains with versatile characteristics such as security analysis, our modular formalization greatly reduces the adoption effort.

The branching-time temporal logic μ-calculus has been formalized in Coq [10,16], LEGO [18], and ACL2 [8] with different intentions. A formalization of PTL can be found in Coq [3,2]. The temporal logic of actions TLA [7] has been formalized in Isabelle [9]. A shallow embedding of CTL is also available [1]. None of these formalizations admits both state and path formulae. Although μ-calculus is more expressive than CTL*, it is not accessible to practitioners due to its arcane syntax and semantics. CTL*, on the other hand, is an accessible generalization of both PTL and CTL. We feel our CTL* formalization would be more useful in practice.

The paper is structured as follows. A brief review of the syntax and semantics of CTL* is given in Section 2. Section 3 identifies assumptions in our formalization of Kripke structures. Based on these assumptions, we formalize paths and CTL* in Section 4 and 5 respectively. The validity of axiom schemata and the soundness of inference rules are discussed in Section 6. Finally, Section 7 concludes the paper and highlights future works.

2 Preliminaries

Let AP be the set of *atomic propositions*. The *syntax* of CTL* is defined as follows [4].

(S0) If p is an atomic proposition, p is a state formula;
(S1) If p and q are state formulae, $p \wedge q$ and $\neg p$ are state formulae;
(S2) If f is a path formula, $\mathbf{A}f$ and $\mathbf{E}f$ are state formulae;
(P0) If p is a state formula, p is a path formula;
(P1) If f and g are path formulae, $f \wedge g$ and $\neg f$ are path formulae;
(P2) If f and g are path formulae, $\mathbf{X}f$ and $f\mathbf{U}g$ are path formulae.

A *Kripke structure* $K = (S, \rightarrow, L)$ consists of a set of *states* S, a total *transition relation* $\rightarrow \subseteq S \times S$, and a labeling function $L : S \rightarrow 2^{AP}$. A *path* π in K is an infinite sequence of states $s_0 s_1 \cdots s_n \cdots$ such that $s_i \rightarrow s_{i+1}$ for all $i \geq 0$. We use the notations $\pi(i) = s_i$ and $\pi_i = s_i s_{i+1} \cdots$ to denote the *i-th state* and the *i-th suffix* of the path π respectively. Note that $\pi = \pi_0$. The *semantics* of a CTL* formula is defined as follows.

$$\begin{array}{ll}
K, s \models P & \text{if } P \in L(s), \text{ where } P \in AP \\
K, s \models \neg p & \text{if not } K, s \models p \\
K, s \models p \wedge q & \text{if } K, s \models p \text{ and } K, s \models q \\
K, s \models \mathbf{A}f & \text{if } K, \pi \models f \text{ for all } \pi \text{ with } \pi(0) = s \\
K, s \models \mathbf{E}f & \text{if } K, \pi \models f \text{ for some } \pi \text{ with } \pi(0) = s \\
K, \pi \models p & \text{if } K, \pi(0) \models p \\
K, \pi \models \neg f & \text{if not } K, \pi \models f \\
K, \pi \models f \wedge g & \text{if } K, \pi \models f \text{ and } K, \pi \models g \\
K, \pi \models \mathbf{X}f & \text{if } K, \pi_1 \models f \\
K, \pi \models f\mathbf{U}g & \text{if there is a } k \text{ such that } K, \pi_k \models g \text{ and } K, \pi_j \models f \text{ for all } 0 \leq j < k
\end{array}$$

We will use $p, q, r, \ldots$ for state formulae, $f, g, h, \ldots$ for path formulae, and $\phi, \psi, \ldots$ for CTL* formulae. Derived operators such as $\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$, $\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$, $\phi \Leftrightarrow \psi \equiv (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$, $\mathbf{F}f \equiv \mathit{true}\mathbf{U}f$, and $\mathbf{G}f \equiv \neg\mathbf{F}\neg f$ are also used. The operators $\mathbf{A}$ and $\mathbf{E}$ are called *path quantifiers*; $\mathbf{X}$, $\mathbf{U}$, $\mathbf{F}$, and $\mathbf{G}$ are *linear temporal operators*.

Both propositional linear temporal logic (PTL) and computational tree logic (CTL) are proper subclasses of CTL*. PTL formulae are constructed by the rules (P0) to (P2) where atomic propositions are the only state formulae. CTL consists of state formulae with the restriction that all linear temporal operators are prefixed by path quantifiers. That is, only the temporal operators **AX**, **AG**, **AF**, **AU**, **EX**, **EG**, **EF**, and **EU** are allowed. The system UB is a subclass of CTL, where only the temporal operators **AX**, **AG**, **AF**, **EX**, **EG**, and **EF**, are allowed.

Observe that all subformulae of a PTL formula are themselves PTL formulae, and PTL formulae in turn are path formulae. A formalization of pure path formulae suffices for PTL. Similarly, a formalization of pure state formulae would be sufficient for CTL. In comparison, the formalization of CTL* is greatly complicated by admitting both state and path formulae. The techniques found in [14,12,13,16,8,10] are therefore not directly applicable.

3 Kripke Structures

One distinguished feature of our formalization is the use of Coq module system. When this research was initiated, our goal was to build a unified verification framework in Coq [17]. In addition to the temporal logic CTL*, we also formalize a model specification language in our framework. It is but natural to use Kripke structures as the interface between both formalizations. Our formalization of CTL* therefore assumes an abstract interface of Kripke structures. This can be done by the following module type definition.

```
Module Type KRIPKE .
  Parameter st : Set .
  Prarmeter succ : st -> st -> Prop .
  ...
End KRIPKE .
```

Two parameters are assumed in the module type `KRIPKE`. The set `st` and the predicate `succ` formalize the set of states and the transition relation in a Kripke structure respectively. To formalize the totality of the transition relation, one might add the following requirement in the module type `KRIPKE`.[1]

Axiom totality_alt : forall s : st, exists s' : st, succ s s' .

However, Calculus of Inductive Constructions does not allow the state *s'* to be extracted from the axiom *totality_alt* lest inconsistency would incur. We therefore formalize the totality by the inductive type `post` and the axiom `totality`.

```
Inductive post (s : st) : Set :=
  | post_intro (s' : st) : succ s s' -> post s .
Axiom totality : forall s : st, post s .
```

For any state `s`, the set `post s` contains elements of the form `post_intro s'` where `s` and `s'` satisfy the transition relation `succ`. The axiom `totality` simply states that the set `post s` is not empty for all state `s`.

When a concrete Kripke structure is available, our formalization of CTL* can be instantiated to analyze properties on the Kripke structure. Note that the semantics of a CTL* formula varies from different Kripke structures. Each instantiation of our CTL* theory gives a specialized interpretation of formulae in the given Kripke structure.

[1] The keyword `Axiom` is perhaps a little misleading. These "axioms" need be established in module definitions; they do not hold automatically.

4 Paths

A path in a Kripke structure is an infinite state sequence where successive states satisfy the transition relation. Several formalizations of paths can be found in literature. In [11], a path is a function of type `nat -> st`, while [14,3,2] use coinductive data types. Although the coinductive formalization admits partiality and is therefore more general than the functional one [14], we feel that the benefit of generality would be better left for users to decide. Hence, we would rather not commit to a particular formalization of paths in our CTL* formalization. Since it is inessential to know exactly how paths are formalized, we can exploit the Coq module system to isolate the formalization of paths from their interface. Consider the following interface of paths.

```
Module Type PATH .
  Parameters st path : Set .
  Parameter succ : st -> st -> Prop .
  Parameter hd : path -> st .
  Parameter tl : path -> path .
  Parameter cons : forall (s : st) (pi : path), succ s (hd pi) -> path .
  ...
End PATH .
```

The parameters `st` and `succ` inherit from a concrete `KRIPKE` module. Paths are formalized as the set `path`. The parameters `hd` and `tl` retrieve the head and tail of a path respectively. Additionally, the parameter `cons` constructs a new path from a state and a path, provided that the state and the head of the path satisfy the transition relation.

Of course, the typing information alone does not entail the intended semantics. It is rather easy to impose semantic requirements on the parameters in the Coq module system. For instance, we enforce the following semantic constraints in the module type `PATH`.

```
  Axiom hd_cons : forall (s : st) (pi : path) (H : succ s (hd pi)),
    hd (cons s pi H) = s .
  Axiom tl_cons : forall (s : st) (pi : path) (H : succ s (hd pi)),
    tl (cons s pi H) = pi .
  Axiom pi_succ : forall pi : path, succ (hd pi) (hd (tl pi)) .
```

The axioms `hd_cons` and `tl_cons` specify the relations among the parameters `hd`, `tl`, and `cons`. The axiom `pi_succ` states that the first and the second states of any path satisfy the transition relation.

Another useful fact in our formalization is that each state has a path from it. More formally, we have the following axiom in the module type `PATH`.

```
Axiom ex_path : forall s : st, exists pi : path, hd pi = s .
```

Due to the totality of the transition relation in the underlying Kripke structure, one would expect the axiom `ex_path` in any reasonable formalization of paths. Indeed, the axiom will be handy when we prove the validity of axiom schemata in axiomatizations of various temporal logics in Section 6.

Our modular formalization of paths is a functor which takes modules of type `KRIPKE` and generates a module of type `PATH`. Furthermore, the generated module shares the formalizations of states (`st`) and the transition relation (`succ`) with the input module.

```
Module Path (KS : KRIPKE) : PATH with Definition st := KS.st
                                 with Definition succ := KS.succ .
  ...
End Path .
```

Similar to [14,3,2], our formalization of paths is based on coinductive data types. We start with the conventional coinductive definition of lazy lists.

```
CoInductive stream : Set := scons : st -> stream -> stream .
Definition shd (str : stream) : st := match str with scons s _ => s end .
```

A stream is simply an infinite sequence of states. The constructor `scons` takes a state and a stream to create a stream. The function `shd` retrieves the head of a stream. Unlike paths, there is no restriction on successive states in a stream. To assert the transition relation `succ` on successive states, the following coinductively defined predicate on streams is used.

```
CoInductive is_path : stream -> Prop :=
  | path_intro (s : st) (str : stream) :
    succ s (shd str) -> is_path str -> is_path (scons s str) .
```

To check if the stream `(scons s str)` is a path, it suffices to verify that

1. the state `s` and the head of `str` (`shd str`) satisfies the transition relation `succ`; and
2. the stream `str` is indeed a path.

Note that our definitions of `stream` and `is_path` are coinductive. In comparison, streams are defined by domain equations in [14]. The coinductively defined type `is_path` becomes a function on the stream domain.[2]

```
Fixpoint is_path_p (str : stream) : Prop :=
  match str with scons s tl => succ s (shd tl) /\ is_path_p tl end .
```

But the function *is_path_p* should be evaluated lazily. Special care must be taken to define it over the stream domain. Coinductive defined types in Calculus of Constructions greatly simplify our formalization. We therefore prefer our purely coinductive formalization.

It is now easy to define the set `path` as follows.

```
Definition path : Set := { str : stream | is_path str } .
```

[2] It is noted that the definition is ill-formed because `stream` is not an inductive type.

The set `path` consists of streams satisfying the predicate `is_path`. The function `hd` can now be defined.

```
Definition hd (pi : path) : st := let (str, _) := pi in shd str .
```

Since a path `pi` is merely a tuple of a stream `str` and its proof of "pathness," the head of `pi` can be computed by invoking the auxiliary function `shd`. Other parameters are defined similarly.

To finish the definition of the functor `Path`, we have to establish the axioms `hd_cons`, `tl_cons`, `pi_succ`, and `ex_path` with our definitions of `hd`, `tl`, and `cons`. Except for `ex_path`, all proofs are rather straightforward. The proof of `ex_path` requires the *decomposition lemma* in [2] and essentially defines a stream from any given state coinductively.

5 CTL*

Recall that CTL* formulae consist of two types of formulae: state and path formulae describe properties about states and paths respectively. In our formalization of CTL*, state and path formulae are of type `st -> Prop` and `path -> Prop` respectively, where `st` and `path` in turn inherit from modules of type `PATH`. An atomic proposition specifies properties about states and is thus of type `st -> Prop`.

```
Module CTLS (Path : PATH) .
  Definition st := Path.st .
  Definition path := Path.path .

  Definition atomic_proposition : Type := st -> Prop .
  Definition st_formula : Type := st -> Prop .
  Definition path_formula : Type := path -> Prop .
  ...
End CTLS .
```

Since a state formula is also a path formula, we define the function `st2path` to coerce state formulae.

```
Definition st2path (p : st_formula) : path_formula :=
  fun (pi : path) => p (Path.hd pi) .
Coercion st2path : st_formula >-> path_formula .
```

To help users construct CTL* formulae, we formalize each linear temporal operator and path quantifier in CTL* as an inductively defined type in Calculus of Inductive Constructions. Each CTL* formula is therefore a type expression in our formalization. To prove a CTL* formula amounts to building a term of the corresponding type expression by constructors of respective inductively defined types.

5.1 State Formulae

Given a state formula p, its negation corresponds to the type expression neg_s p. To construct a proof of its negation, it suffices to find a proof of ~ p s for any state s.

```
Inductive neg_s (p : st_formula) : st_formula :=
  | neg_s_intro : forall s : st, ~ p s -> neg_s p s .
```

Observe that the inductively defined type neg_s p is of sort st_formula as well. It can thus be used to construct more complicated type expressions. For instance, the corresponding type expression for the CTL* formula $\neg\neg p$ is neg_s neg_s p. The following notation for the type expression neg_s p is defined for convenience.

```
Notation "! p" := (neg_s p) (at level 75, right associativity) .
```

Henceforth, we will write ! p for the type corresponding to the formula $\neg p$. Other logical operators can be formalized similarly. We use the notations p && q, p || q, p ==> q, and p <==> q for the corresponding type expressions for the formulae $p \wedge q$, $p \vee q$, $p \Rightarrow q$, and $p \Leftrightarrow q$ respectively.

It is as easy to formalize path quantifiers in CTL* as well. For instance, proving the state formula $\mathbf{A}f$ on the state s is to demonstrate that all paths π from s satisfy f. Hence the following type is used for the state formula $\mathbf{A}f$.

```
Inductive A (f : path_formula) : st_formula :=
  | A_intro : forall s : st,
    (forall pi : path, s = Path.hd pi -> f pi) -> A f s .
```

We will use the notations A f and E f for the type expressions of the formulae $\mathbf{A}f$ and $\mathbf{E}f$ respectively.

5.2 Path Formulae

Logical operators for path formulae are similar to those of state formulae. The inductively defined type and_p f g, for instance, formalizes the conjunction of path formulae f and g.

```
Inductive and_p (f g : path_formula) : path_formula :=
  | and_p_intro : forall pi : path, f pi /\ g pi -> and_p f g pi .
```

The notations ‘! f, f ‘&& g, f ‘|| g, f ‘==> g, and f ‘<==> g are used for the negation, conjunction, disjunction, implication, and logical equivalence respectively. Note that the back quote (‘) distinguishes from the corresponding types for state formulae.

Our formalization of linear temporal operators essentially follows those of PTL in [3,2]. Instead of using streams as in prior formalizations, our formalization is based on paths in Kripke structures.

In order to show that a path π satisfies the path formula $\mathbf{X}f$, it is necessary to show that the tail of π satisfies the formula f. Thus, the inductively defined type X f is as follows.

```
Inductive X (f : path_formula) : path_formula :=
  | X_intro : forall pi : path, f (Path.tl pi) -> X f pi .
```

Now consider the path formula $\mathbf{G}f$. The path π satisfies $\mathbf{G}f$ if it satisfies f *and* its tail satisfies $\mathbf{G}f$. Note that a proof term of π satisfying $\mathbf{G}f$ is infinite for π is infinite. We therefore use a coinductively defined type `G f` for the formula $\mathbf{G}f$.

```
CoInductive G (f : path_formula) : path_formula :=
  | G_intro : forall pi : path, f pi -> G f (Path.tl pi) -> G f pi .
```

For the path formula $\mathbf{F}f$, there are two ways to demonstrate the path π satisfying the formula. If π satisfies f, we are done. Otherwise, the tail of π must satisfy $\mathbf{F}f$. Therefore, a proof term of type `F f` is built by the constructors `F0_intro` and `F_intro` inductively.

```
Inductive F (f : path_formula) : path_formula :=
  | F0_intro : forall pi : path, f pi -> F f pi
  | F_intro  : forall pi : path, F f (Path.tl pi) -> F f pi .
```

The definition of the type `U f g` is similar to `F f`. To show the path π satisfying $f\mathbf{U}g$ is to show that π satisfies g, *or* it satisfies f and its tail satisfies $f\mathbf{U}g$.

```
Inductive U (f g : path_formula) : path_formula :=
  | U0_intro : forall pi : path, g pi -> U f g pi
  | U_intro : forall pi : path, f pi -> U f g (Path.tl pi) -> U f g pi .
```

We will write `X f`, `G f`, `F f`, and `f U g` for the corresponding type expressions of formulae $\mathbf{X}f$, $\mathbf{G}f$, $\mathbf{F}f$, and $f\mathbf{U}g$ respectively. Observe that derived temporal operators are also formalized. They allow us to carry out formal proofs more intuitively.

To compare with the formalizations in [3,2], recall that a stream is an infinite sequence of states. There is no restriction imposed on successive states in a stream. A path, on the other hand, is a stream satisfying the co-inductively defined predicate `is_path`; successive states in a path satisfy the transition relation `succ`. Hence the computation of the underlying Kripke structure is implicit in our formalization.

In contrast, a stream filter *path_filter : stream -> Prop* is needed in statements about paths to witness the transition relation of the underlying computation model in [3,2]. For instance, the following axiom states that the state formula *fair* holds infinitely often along all paths from *s*.

```
Axiom fairness : forall (s : st) (str : stream),
    s = shd str -> path_filter str -> (G F fair) str .
```

Since there is only one implicit universal path quantifier in any PTL formula, adding *path_filter* does not incur too much overhead in [3,2]. However, it becomes rather cumbersome for CTL* where nested path quantifiers are allowed. Moreover, the proofs in prior formalizations would move between streams and paths for each path quantifier, even though paths are in fact of the main interest. Our formalization, on the other hand, is solely based on paths. Users do not see any reference to streams and can focus on key concepts in the our formalization.

6 Examples

With the formalization of CTL* in Section 5, we are able to prove validity of axiom schemata and soundness of inference rules in axiomatizations of temporal logics. Since PTL and CTL are subclasses of CTL*, restrictions of our CTL* formalization suffice for the proofs of respective theorems in their axiomatizations. It is unnecessary to have formal proofs in different formalizations. Moreover, all axiom schemata and inference rules in our modular formalization can be instantiated for different security models. In the following sections, we discuss the validity and soundness of axiom schemata and inference rules in axiomatizations of PTL, UB, CTL, and CTL* respectively.

6.1 PTL

Figure 1 shows the axiomatization of PTL in [6]. For each axiom schema, we would like to show it is indeed valid in our formalization. We say a PTL formula f is *valid* (denoted by $\models f$) if $K, \pi \models f$ for any Kripke structure K and path π. An *instance* of an axiom schema Ψ is a PTL formula obtained by substituting all variables in Ψ with PTL formulae. For instance, suppose $P \in AP$. Then $\mathbf{X}\neg\mathbf{G}P \Leftrightarrow \neg\mathbf{X}\mathbf{G}P$ is an instance of the axiom schema $(ax1)$. We say an axiom schema Ψ is *valid* if all instances of Ψ are valid.

To show the validity of the axiom schema $(ax2)$ in Figure 1, we first formalize the validity of path formulae as follows.

```
Definition model_p (f : path_formula) := forall pi : path, f pi .
Notation "‘|= f" := (model_p f) (at level 100, no associativity) .
```

$$\begin{array}{ll}
(ax1) & \vdash \mathbf{X}\neg f \Leftrightarrow \neg\mathbf{X}f \\
(ax2) & \vdash \mathbf{X}(f \Rightarrow g) \Rightarrow (\mathbf{X}f \Rightarrow \mathbf{X}g) \\
(ax3) & \vdash \mathbf{G}f \Rightarrow (f \wedge \mathbf{X}\mathbf{G}f) \\
(mp) & \dfrac{\vdash f \quad \vdash f \Rightarrow g}{\vdash g}
\end{array}$$

$$(nex)\ \frac{\vdash f}{\vdash \mathbf{X}f} \qquad (ind)\ \frac{\vdash f \Rightarrow g \quad \vdash f \Rightarrow \mathbf{X}f}{\vdash f \Rightarrow \mathbf{G}g}$$

Fig. 1. An Axiomatization of PTL

It is now straightforward to state the validity of each axiom schema. For example, the validity of the axiom schema $(ax2)$ is as follows.

```
Theorem ax2 : forall f g : path_formula,
  ‘|= (X (f ‘==> g)) ‘==> (X f) ‘==> (X g) .
```

The soundness of inference rules can be similarly formalized. The theorem `ind`, for instance, formalizes the soundness of the rule (ind).

```
Theorem ind : forall f g : path_formula,
  (‘|= f ‘==> g) /\ (‘|= f ‘==> X f) -> (‘|= f ‘==> G g) .
```

Note that the theorem `ax2` is in fact more general than the validity of axiom schema ($ax2$) in the pure PTL setting. The theorem states that the axiom schema ($ax2$) is valid not only for all PTL formulae, but also all path formulae in CTL*.[3] Similarly, the theorem `ind` is more general than the soundness of inference rule (ind). The proofs of validity and soundness are carried out in the default Coq environment. We are able to prove the validity of all axiom schemata and the soundness of all inference rules in Figure 1 with our formalization.

6.2 UB

As for the PTL axiomatization, we would like to prove the validity and soundness theorems of the axiomatization of UB in Figure 2 formally. Specifically, we say a state formula p is *valid* if $K, s \vdash p$ for any Kripke structure K and state s. An axiom schema is *valid* if all its instances are valid. The validity of state formulae is formalized as follows.

```
Definition model_s (p : st_formula) := forall s : st, p s .
Notation "|= p" := (model_s p) (at level 100, no associativity) .
```

$$
\begin{array}{l}
(A1) \vdash \mathbf{AG}(p \Rightarrow q) \Rightarrow (\mathbf{AG}p \Rightarrow \mathbf{AG}q) \\
(A2) \vdash \mathbf{AX}(p \Rightarrow q) \Rightarrow (\mathbf{AX}p \Rightarrow \mathbf{AX}q) \\
(A3) \vdash \mathbf{AG}p \Rightarrow \mathbf{AX}p \wedge \mathbf{AXAG}p \\
(A4) \vdash \mathbf{AG}(p \Rightarrow \mathbf{AX}p) \Rightarrow (p \Rightarrow \mathbf{AG}p) \\
(E1) \vdash \mathbf{AG}(p \Rightarrow q) \Rightarrow (\mathbf{EG}p \Rightarrow \mathbf{EG}q) \\
(E2) \vdash \mathbf{EG}p \Rightarrow p \wedge \mathbf{EXEG}p \\
(E3) \vdash \mathbf{AG}p \Rightarrow \mathbf{EG}p \\
(E4) \vdash \mathbf{AG}(p \Rightarrow \mathbf{EX}p) \Rightarrow (p \Rightarrow \mathbf{EG}p) \\
(R1)\ \dfrac{\vdash p \quad \vdash p \Rightarrow q}{\vdash q} \qquad (R2)\ \dfrac{\vdash p}{\vdash \mathbf{AG}p}
\end{array}
$$

Fig. 2. An Axiomatization of UB

The validity of the axiom schemata and the soundness of inference rules are formalized similarly. For instance, the validity of axiom schema ($A3$) is stated in the following theorem.

```
Theorem A3 : forall p : st_formula, |= (A G p) ==> (A X p && A X A G p) .
```

As in PTL, the theorem `A3` is more general than the validity of the axiom schema ($A3$) in the pure UB setting. Unlike PTL, however, the proofs of validity and soundness require switching between state and path formulae. Since our

[3] As an anonymous reviewer points to us, it is even valid for path predicates which are not expressible in CTL* because of the shallow embedding.

formalization is in fact for CTL*, each temporal operator in a UB formula has to be decomposed as a path quantifier followed by a linear temporal operator. Consider the validity of the axiom schema ($E3$).

```
Theorem E3 : forall p : st_formula, |= (A G p) ==> (E G p) .
```

A simple proof is to demonstrate a path from any given state satisfying $\mathbf{AG}p$ and show it indeed satisfies $\mathbf{G}p$. We therefore use the axiom `Path.ex_path` to construct an arbitrary path from the given state, and show that the path is indeed a witness of $\mathbf{G}p$ by the assumption $\mathbf{AG}p$.

We are able to prove the soundness of all inference rules in Figure 2 formally. We also establish the validity of all axiom schemata but ($E4$) in Calculus of Inductive Construction. To explain the difficulty in proving the validity of ($E4$), recall its formulation.

```
Theorem E4 : forall p : st_formula, |= (A G (p ==> E X p)) ==> p ==> E G p .
```

One possible proof of `E4` is to construct a path satisfying $\mathbf{G}p$. But eliminating the assumption `A G (p ==> E X p)` of sort `Prop` is not allowed in the construction of paths of sort `Set`. Alternatively, we fail to demonstrate the existence of a path satisfying $\mathbf{G}p$ in classical logic. The assumption suggests that any path can be modified to admit p in one more state. But the existence of a path satisfying $\mathbf{G}p$ in the limit eludes us. Currently, we do not know how to prove it with the present formalization.

6.3 CTL

The axiomatization of CTL in [5,4] is shown in Figure 3. The validity and soundness of axiom schemata and inference rules follow the same style in Section 6.2. The proof techniques used in the previous section are carried over without difficulties. Indeed, the axiomatization of UB in Figure 2 can be obtained from the axiomatization of CTL in Figure 3 [5]. It is not surprising to prove the validity

$$\begin{array}{l}
(Ax1) \vdash \mathbf{EF}p \Leftrightarrow \mathbf{E}[true\,\mathbf{U}p] \\
(Ax2) \vdash \mathbf{AF}p \Leftrightarrow \mathbf{A}[true\,\mathbf{U}p] \\
(Ax3) \vdash \mathbf{EX}(p \vee q) \Leftrightarrow \mathbf{EX}p \vee \mathbf{EX}q \\
(Ax4) \vdash \mathbf{AX}p \Leftrightarrow \neg\mathbf{EX}\neg p \\
(Ax5) \vdash \mathbf{E}(p\mathbf{U}q) \Leftrightarrow q \vee (p \wedge \mathbf{EXE}(p\mathbf{U}q)) \\
(Ax6) \vdash \mathbf{A}(p\mathbf{U}q) \Leftrightarrow q \vee (p \wedge \mathbf{AXA}(p\mathbf{U}q)) \\
(Ax7) \vdash \mathbf{EX}true \wedge \mathbf{AX}true
\end{array}$$

$$(R1)\ \frac{\vdash p \Rightarrow q}{\vdash \mathbf{EX}p \Rightarrow \mathbf{EX}q} \qquad (R2)\ \frac{\vdash r \Rightarrow (\neg q \wedge \mathbf{EX}r)}{\vdash r \Rightarrow \neg\mathbf{A}(p\mathbf{U}q)}$$

$$(R3)\ \frac{\vdash r \Rightarrow [\neg q \wedge \mathbf{AX}(r \vee \neg\mathbf{E}(p\mathbf{U}q))]}{\vdash r \Rightarrow \neg\mathbf{E}(p\mathbf{U}q)} \qquad (R4)\ \frac{\vdash p \quad \vdash p \Rightarrow q}{\vdash q}$$

Fig. 3. An Axiomatization of CTL

and soundness of axiom schemata and inference rules for CTL by generalizing the proof techniques used for UB.

We have succeeded in proving the validity of all axiom schemata in Figure 3. Unlike the proofs for the system UB, classical reasoning is used in a couple of axiom schemata. Specifically, contraposition and De Morgan's law are used in the proofs of $(Ax4)$ and $(Ax6)$ respectively. As for the soundness of inference rules, an obstacle similar to the axiom schema $(E4)$ in UB is encountered in the inference rule $(R2)$. For other inference rules, we are able to prove their soundness formally.

6.4 CTL*

Figure 4 shows the sound and complete axiomatization of CTL* in [15]. The side condition C in the axiom schema (AA) is syntactic and somewhat complicated. It requires pairwise inconsistency of atomic propositions in finite sets, and a function choosing atomic propositions to hold at the next state along a path (Definition 4 in [15]).

Unlike the axiomatizations in previous sections, both path and state formulae are present. The validity of path formulae is used in the axiom schemata $(C1)$ to $(C8)$, $(C11)$, and $(C15)$. The axiom schemata $(C9)$, $(C10)$, and $(C12)$ to $(C14)$ are valid as state formulae. Their formal proofs can be carried out in our formalization of CTL*. Prior formalizations, in comparison, would not even be able to formulate axiom schemata $(C12)$ nor $(C15)$. A CTL* formalization is therefore needed in establishing validity of axiom schemata formally.

Except the axiom schemata (LC), we have proved the validity of all axiom schemata in Figure 4. The axiom schema (LC) is another generalization of the axiom schema $(E4)$ in UB and the inference rule $(R3)$ in CTL. The side condition

$$(C1) \vdash \mathbf{F}\neg\neg f \Leftrightarrow \mathbf{F}f$$
$$(C2) \vdash \mathbf{G}(f \Rightarrow g) \Rightarrow (\mathbf{G}f \Rightarrow \mathbf{G}g)$$
$$(C3) \vdash \mathbf{G}f \Rightarrow (f \wedge \mathbf{X}f \wedge \mathbf{XG}f)$$
$$(C4) \vdash \mathbf{X}\neg f \Leftrightarrow \neg\mathbf{X}f$$
$$(C5) \vdash \mathbf{X}(f \Rightarrow g) \Rightarrow (\mathbf{X}f \Rightarrow \mathbf{X}g)$$
$$(C6) \vdash \mathbf{G}(f \Rightarrow \mathbf{X}f) \Rightarrow (f \Rightarrow \mathbf{G}f)$$
$$(C7) \vdash (f\mathbf{U}g) \Leftrightarrow (g \vee (f \wedge \mathbf{X}(f\mathbf{U}g)))$$
$$(C8) \vdash (f\mathbf{U}g) \Rightarrow \mathbf{F}g$$
$$(C9) \vdash \mathbf{A}(f \Rightarrow g) \Rightarrow (\mathbf{A}f \Rightarrow \mathbf{A}g)$$
$$(C10) \vdash \mathbf{A}f \Rightarrow \mathbf{AA}f \qquad (C11) \vdash \mathbf{A}f \Rightarrow f$$
$$(C12) \vdash f \Rightarrow \mathbf{AE}f \qquad (C13) \vdash \mathbf{A}\neg f \Leftrightarrow \neg\mathbf{E}f$$
$$(C14) \vdash p \Rightarrow \mathbf{A}p \qquad (C15) \vdash \mathbf{AX}f \Rightarrow \mathbf{XA}f$$
$$(LC) \vdash \mathbf{AG}(\mathbf{E}f \Rightarrow \mathbf{EX}((\mathbf{E}g)\mathbf{U}(\mathbf{E}f))) \Rightarrow (\mathbf{E}f \Rightarrow \mathbf{EG}((\mathbf{E}g)\mathbf{U}(\mathbf{E}f)))$$
$$(AA)\ \frac{\vdash \theta \Rightarrow \psi}{\vdash \psi}\ C$$

Fig. 4. An Axiomatization of CTL*

C in the inference rule (AA) is too complicated to formulate in our current formalization. The other axiom schemata but $(C1)$ are proved in the default Coq environment. The axiom schema $(C1)$, apparently, requires classical reasoning and is proved by importing the `Classical` theory.

7 Conclusion and Future Work

A modular formalization of CTL* is presented in the paper. The formalization subsumes prior works of PTL and CTL. We have succeeded in proving validity and soundness of an axiomatization of PTL formally. For the branching-time temporal logics UB, CTL, and CTL*, almost all validity of axiom schemata and soundness of inference rules have also been established in the new formalization. Furthermore, the modularity of our formalization allows to be instantiated for different security models, provided they satisfy certain assumptions. Theorems proved in this paper are reusable in any instantiation.

One possible way to resolve the difficulties in the validity of the axiom schemata $(E4)$ in UB, (LC) in CTL*, and the soundness of the inference rule $(R3)$ in CTL is by *strong specification* [2]. We are working on a new formalization based on strong specification to address the problem. For the syntactic side condition of the inference rule (AA) in CTL*, a formalization with syntactic representations of CTL* formulae would be necessary. A generalized version of the deep embedding of CTL* in [17] could be useful in formulating the side condition in (AA).

Acknowledgment. We would like to thank Yih-Kuen Tsay and anonymous reviewers for their constructive comments in improving the paper.

References

1. Bauer, G.: Some properties of CTL. Technische Universität München, Isabelle/Isar document (2001)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. In: Texts in Theoretical Computer Science, Springer, Heidelberg (2004)
3. Coupet-Grimal, S.: An axiomatization of linear temporal logic in the calculus of inductive constructions. Logic and Computation 13(6), 801–813 (2003)
4. Emerson, E.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 995–1072. Elsevier Science Publishers (1990)
5. Emerson, E., Halpern, J.: Decision procedures and expressiveness in the temporal logic of branching time. Journal of Computer and System Sciences 30, 1–24 (1985)
6. Kröger, F.: Temporal Logic of Programs. Springer, Heidelberg (1987)
7. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16(3), 872–923 (1994)
8. Manolios, P.: Mu-calculus model-checking. In: Computer-Aided Reasoning: ACL2 Case Studies, pp. 93–111. Kluwer Academic Publishers, Dordrecht (2000)

9. Merz, S.: Isabelle/TLA. Technische Universität München, Isabelle/Isar document (1998)
10. Miculan, M.: On the formalization of the modal μ-calculus in the calculus of inductive constructions. Information and Computation 164(1), 199–231 (2001)
11. Müller, O., Nipkow, T.: Combining model checking and deduction for I/O-automata. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 1–16. Springer, Heidelberg (1995)
12. Müller, O.: I/O automata and beyond - temporal logic and abstraction in Isabelle. In: Grundy, J., Newey, M. (eds.) Theorem Proving in Higher Order Logics. LNCS, vol. 1479, pp. 331–348. Springer, Heidelberg (1998)
13. Müller, O.: A Verification Environment for I/O Automata Based on Formalized Meta-Theory. PhD thesis, Technische Universität München (1998)
14. Müller, O., Nipkow, T.: Traces of I/O automata in Isabelle/HOLCF. In: Bidoit, M., Dauchet, M. (eds.) CAAP 1997, FASE 1997, and TAPSOFT 1997. LNCS, vol. 1214, pp. 580–595. Springer, Heidelberg (1997)
15. Reynolds, M.: An axiomatization of full computation tree logic. Journal of Symbolic Logic 66(3), 1011–1057 (2001)
16. Sprenger, C.: A verified model checker for the modal μ-calculus in Coq. In: Steffen, B. (ed.) ETAPS 1998 and TACAS 1998. LNCS, vol. 1384, pp. 167–183. Springer, Heidelberg (1998)
17. Tsai, M.H., Wang, B.Y.: Modular formalization of reactive modules and CTL* in Coq. submitted for publication (2006)
18. Yu, S., Luo, Z.: Implementing a model checker for LEGO. In: Fitzgerald, J., Jones, C.B., Lucas, P. (eds.) FME 1997. LNCS, vol. 1313, pp. 442–458. Springer, Heidelberg (1997)
19. Zhang, X., Parisi-Presicce, F., Sandhu, R.: Formal model and policy specification of usage control. ACM Transactions on Information and System Security 8(4), 351–387 (2005)