A shallow embedding of HyperCTL*

Markus N. Rabe Peter Lammich Andrei Popescu

Contents

1 Introduction

We formalize HyperCTL*, a temporal logic for expressing security properties introduced in [1,2]. We first define a shallow embedding of HyperCTL*, within which we prove inductive and coinductive rules for the operators. Then we show that a HyperCTL* formula captures Goguen-Meseguer noninterference, a landmark information flow property. We also define a deep embedding and connect it to the shallow embedding by a denotational semantics, for which we prove sanity w.r.t. dependence on the free variables. Finally, we show that under some finiteness assumptions about the model, noninterference is given by a (finitary) syntactic formula.

For the semantics of HyperCTL*, we mainly follow the earlier paper [1]. The Kripke structure for representing noninterference is essentially that of [1,Appendix B] – however, instead of using the formula from [1,Appendix B], we further add idle transitions to the Kripke structure and use the simpler formula from [1,Section 2.4].

[1] Bernd Finkbeiner, Markus N. Rabe and César Sánchez. A Temporal Logic for Hyperproperties. CoRR, abs/1306.6657, 2013.

[2] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe and César Sánchez. Temporal Logics for Hyperproperties. POST 2014, 265-284.

2 Preliminaries

abbreviation *any* **where** *any* ≡ *undefined*

lemma *append-singl-rev*: *a # as* = [*a*] @ *as* ⟨*proof*⟩

lemma *list-pair-induct*[*case-names Nil Cons*]:
assumes *P* [] **and** ⋀*a b list*. *P list* ⟹ *P* ((*a*,*b*) # *list*)
shows *P lista*
⟨*proof*⟩

lemma *list-pair-case*[*elim*, *case-names Nil Cons*]:
assumes *xs* = [] ⟹ *P* **and** ⋀*a b list*. *xs* = (*a*,*b*) # *list* ⟹ *P*
shows *P*
⟨*proof*⟩

definition *asList* :: *'a set* ⇒ *'a list* **where**
asList A ≡ *SOME as*. *distinct as* ∧ *set as* = *A*

lemma *asList*:
assumes *finite A* **shows** *distinct* (*asList A*) ∧ *set* (*asList A*) = *A*
⟨*proof*⟩

lemmas *distinct-asList* = *asList*[*THEN conjunct1*]
lemmas *set-asList* = *asList*[*THEN conjunct2*]

lemma *map-sdrop*[*simp*]: *sdrop 0* = *id*
⟨*proof*⟩

lemma *stl-o-sdrop*[*simp*]: *stl o sdrop n* = *sdrop* (*Suc n*)
⟨*proof*⟩

lemma *sdrop-o-stl*[*simp*]: *sdrop n o stl* = *sdrop* (*Suc n*)
⟨*proof*⟩

lemma *hd-stake*[*simp*]: $i > 0 \Longrightarrow hd\ (stake\ i\ \pi) = shd\ \pi$
⟨*proof*⟩

3 Shallow embedding of HyperCTL*

We define a notion of "shallow" HyperCTL* formula (sfmla) that captures HyperCTL* binders as meta-level HOL binders. We also define a proof system for this shallow embedding.

3.1 Kripke structures and paths

type-synonym (*'state*,*'aprop*) *path* = (*'state* × *'aprop set*) *stream*

abbreviation *stateOf* **where** $stateOf\ \pi \equiv fst\ (shd\ \pi)$
abbreviation *apropsOf* **where** $apropsOf\ \pi \equiv snd\ (shd\ \pi)$

locale *Kripke* =
  fixes S :: *'state set* **and** *s0* :: *'state* **and** δ :: *'state* ⇒ *'state set*
  and AP :: *'aprop set* **and** L :: *'state* ⇒ *'aprop set*
  assumes *s0*: $s0 \in S$ **and** δ: $\bigwedge s.\ s \in S \Longrightarrow \delta\ s \subseteq S$
  and L : $\bigwedge s.\ s \in S \Longrightarrow L\ s \subseteq AP$
begin

Well-formed paths

coinductive *wfp* :: *'aprop set* ⇒ (*'state*,*'aprop*) *path* ⇒ *bool*
for AP' :: *'aprop set*
where
intro:
$[\![s \in S;\ A \subseteq AP';\ A \cap AP = L\ s;\ stateOf\ \pi \in \delta\ s;\ wfp\ AP'\ \pi]\!]$
$\Longrightarrow$
$wfp\ AP'\ ((s,A)\ \#\#\ \pi)$

lemma *wfp*:
$wfp\ AP'\ \pi \longleftrightarrow$
$(\forall\ i.\ fst\ (\pi\ !!\ i) \in S \wedge snd\ (\pi\ !!\ i) \subseteq AP' \wedge$
$\quad snd\ (\pi\ !!\ i) \cap AP = L\ (fst\ (\pi\ !!\ i)) \wedge$
$\quad fst\ (\pi\ !!\ (Suc\ i)) \in \delta\ (fst\ (\pi\ !!\ i))$
$)$
(**is** $?L \longleftrightarrow (\forall\ i.\ ?R\ i)$)
⟨*proof*⟩

lemma *wfp-sdrop*[*simp*]:
$wfp\ AP'\ \pi \Longrightarrow wfp\ AP'\ (sdrop\ i\ \pi)$
⟨*proof*⟩

end-of-context Kripke

3.2 Shallow representations of formulas

A shallow (representation of a) HyperCTL* formula will be a predicate on lists of paths. The atomic formulas (operator *atom*) are parameterized by atomic propositions (as customary in temporal logic), and additionally by a number indicating the position, in the list of paths, of the path to which the atomic proposition refers – for example, *atom a i* holds for the list of paths πl just in case proposition a holds at the first state of $\pi l!i$, the i'th path in πl. The temporal operators *next* and *until* act on all the paths of the argument list πl synchronously. Finally, the existential quantifier refers to the existence of a path whose origin state is that of the last path in πl.

As an example: *exi* (*exi* (*until* (*atom a* 0) (*atom b* 1))) holds for the empty list iff there exist two paths ρ_0 and ρ_1 such that, synchronously, a holds on ρ_0 until b holds on ρ_1. Another example will be the formula encoding Goguen-Meseguer noninterference.

Shallow HyperCTL* formulas:

type-synonym (*'state*,*'aprop*) *sfmla* = (*'state*,*'aprop*) *path list* ⇒ *bool*

locale *Shallow* = *Kripke S s0 δ AP L*
  for *S* :: *'state set* **and** *s0* :: *'state* **and** *δ* :: *'state* ⇒ *'state set*
  and *AP* :: *'aprop set* **and** *L* :: *'state* ⇒ *'aprop set*
+
  fixes *AP'* **assumes** *AP-AP'*: *AP* ⊆ *AP'*
begin

Primitive operators

definition *fls* :: (*'state*,*'aprop*) *sfmla* **where**
fls πl ≡ *False*

definition *atom* :: *'aprop* ⇒ *nat* ⇒ (*'state*,*'aprop*) *sfmla* **where**
atom a i πl ≡ *a* ∈ *apropsOf* (*πl*!*i*)

definition *neg* :: (*'state*,*'aprop*) *sfmla* ⇒ (*'state*,*'aprop*) *sfmla* **where**
neg φ πl ≡ ¬ *φ πl*

definition *dis* :: (*'state*,*'aprop*) *sfmla* ⇒ (*'state*,*'aprop*) *sfmla* ⇒ (*'state*,*'aprop*) *sfmla* **where**
dis φ ψ πl ≡ *φ πl* ∨ *ψ πl*

definition *next* :: (*'state*,*'aprop*) *sfmla* ⇒ (*'state*,*'aprop*) *sfmla* **where**
next φ πl ≡ *φ* (*map stl πl*)

definition *until* :: (*'state*,*'aprop*) *sfmla* ⇒ (*'state*,*'aprop*) *sfmla* ⇒ (*'state*,*'aprop*) *sfmla* **where**
until φ ψ πl ≡
∃ *i*. *ψ* (*map* (*sdrop i*) *πl*) ∧ (∀ *j* ∈{*0*..<*i*}. *φ* (*map* (*sdrop j*) *πl*))

definition *exii* :: (*'state*,*'aprop*) *sfmla* ⇒ (*'state*,*'aprop*) *sfmla* **where**
exii φ πl ≡
∃ *π*. *wfp AP' π* ∧ *stateOf π* = (*if πl* ≠ [] *then stateOf* (*last πl*) *else s0*)
    ∧ *φ* (*πl* @ [*π*])

definition *exi* :: ((*'state*,*'aprop*) *path* ⇒ (*'state*,*'aprop*) *sfmla*) ⇒ (*'state*,*'aprop*) *sfmla* **where**
exi F π*l* ≡
∃ π. *wfp AP'* π ∧ *stateOf* π = (*if* π*l* ≠ [] *then stateOf* (*last* π*l*) *else s0*)
∧ *F* π π*l*

Derived operators

definition *tr* ≡ *neg fls*
definition *con* φ ψ ≡ *neg* (*dis* (*neg* φ) (*neg* ψ))
definition *imp* φ ψ ≡ *dis* (*neg* φ) ψ
definition *eq* φ ψ ≡ *con* (*imp* φ ψ) (*imp* ψ φ)
definition *fall F* ≡ *neg* (*exi* (λ π. *neg* (*F* π)))
definition *ev* φ ≡ *until tr* φ
definition *alw* φ ≡ *neg* (*ev* (*neg* φ))
definition *wuntil* φ ψ ≡ *dis* (*until* φ ψ) (*alw* φ)

lemmas *main-op-defs* =
fls-def atom-def neg-def dis-def next-def until-def exi-def

lemmas *der-op-defs* =
tr-def con-def imp-def eq-def ev-def alw-def wuntil-def fall-def

lemmas *op-defs* = *main-op-defs der-op-defs*

3.3 Reasoning rules

We provide introduction, elimination, unfolding and (co)induction rules for the connectives and quantifiers.

Boolean operators

lemma *fls-elim*[*elim*!]:
assumes *fls* π*l* **shows** φ
⟨*proof*⟩

lemma *tr-intro*[*intro*!, *simp*]: *tr* π*l*
⟨*proof*⟩

lemma *dis-introL*[*intro*]:
assumes φ π*l* **shows** *dis* φ ψ π*l*
⟨*proof*⟩

lemma *dis-introR*[*intro*]:
assumes ψ π*l* **shows** *dis* φ ψ π*l*
⟨*proof*⟩

lemma *dis-elim*[*elim*]:
assumes *dis* φ ψ π*l* **and** φ π*l* ⟹ χ **and** ψ π*l* ⟹ χ
shows χ
⟨*proof*⟩

lemma *con-intro*[*intro*!]:
assumes $\varphi\ \pi l$ **and** $\psi\ \pi l$ **shows** *con* $\varphi\ \psi\ \pi l$
⟨*proof*⟩

lemma *con-elim*[*elim*]:
assumes *con* $\varphi\ \psi\ \pi l$ **and** $\varphi\ \pi l \Longrightarrow \psi\ \pi l \Longrightarrow \chi$ **shows** χ
⟨*proof*⟩

lemma *neg-intro*[*intro*!]:
assumes $\varphi\ \pi l \Longrightarrow$ *False* **shows** *neg* $\varphi\ \pi l$
⟨*proof*⟩

lemma *neg-elim*[*elim*]:
assumes *neg* $\varphi\ \pi l$ **and** $\varphi\ \pi l$ **shows** χ
⟨*proof*⟩

lemma *imp-intro*[*intro*!]:
assumes $\varphi\ \pi l \Longrightarrow \psi\ \pi l$ **shows** *imp* $\varphi\ \psi\ \pi l$
⟨*proof*⟩

lemma *imp-elim*[*elim*]:
assumes *imp* $\varphi\ \psi\ \pi l$ **and** $\varphi\ \pi l$ **and** $\psi\ \pi l \Longrightarrow \chi$ **shows** χ
⟨*proof*⟩

lemma *imp-mp*[*elim*]:
assumes *imp* $\varphi\ \psi\ \pi l$ **and** $\varphi\ \pi l$ **shows** $\psi\ \pi l$
⟨*proof*⟩

lemma *eq-intro*[*intro*!]:
assumes $\varphi\ \pi l \Longrightarrow \psi\ \pi l$ **and** $\psi\ \pi l \Longrightarrow \varphi\ \pi l$ **shows** *eq* $\varphi\ \psi\ \pi l$
⟨*proof*⟩

lemma *eq-elimL*[*elim*]:
assumes *eq* $\varphi\ \psi\ \pi l$ **and** $\varphi\ \pi l$ **and** $\psi\ \pi l \Longrightarrow \chi$ **shows** χ
⟨*proof*⟩

lemma *eq-elimR*[*elim*]:
assumes *eq* $\varphi\ \psi\ \pi l$ **and** $\psi\ \pi l$ **and** $\varphi\ \pi l \Longrightarrow \chi$ **shows** χ
⟨*proof*⟩

lemma *eq-equals*: *eq* $\varphi\ \psi\ \pi l \longleftrightarrow \varphi\ \pi l = \psi\ \pi l$
⟨*proof*⟩

Quantifiers

lemma *exi-intro*[*intro*]:
assumes *wfp* $AP'\ \pi$
and $\pi l \neq [] \Longrightarrow$ *stateOf* $\pi =$ *stateOf* (*last* πl)
and $\pi l = [] \Longrightarrow$ *stateOf* $\pi =$ *s0*

and $F\ \pi\ \pi l$
shows *exi* $F\ \pi l$
⟨*proof*⟩

lemma *exi-elim*[*elim*]:
assumes *exi* $F\ \pi l$
and
$\bigwedge \pi.\ [\![wfp\ AP'\ \pi;\ \pi l \neq [] \Longrightarrow stateOf\ \pi = stateOf\ (last\ \pi l);\ \pi l = [] \Longrightarrow stateOf\ \pi = s0;\ F\ \pi\ \pi l]\!] \Longrightarrow \chi$
shows χ
⟨*proof*⟩

lemma *fall-intro*[*intro*]:
assumes
$\bigwedge \pi.\ [\![wfp\ AP'\ \pi;\ \pi l \neq [] \Longrightarrow stateOf\ \pi = stateOf\ (last\ \pi l)\ ;\ \pi l = [] \Longrightarrow stateOf\ \pi = s0]\!]$
$\Longrightarrow\ F\ \pi\ \pi l$
shows *fall* $F\ \pi l$
⟨*proof*⟩

lemma *fall-elim*[*elim*]:
assumes *fall* $F\ \pi l$
and
$(\bigwedge \pi.\ [\![wfp\ AP'\ \pi;\ \pi l \neq [] \Longrightarrow stateOf\ \pi = stateOf\ (last\ \pi l);\ \pi l = [] \Longrightarrow stateOf\ \pi = s0]\!]$
$\Longrightarrow F\ \pi\ \pi l)$
$\Longrightarrow \chi$
shows χ
⟨*proof*⟩

Temporal connectives

lemma *next-intro*[*intro*]:
assumes $\varphi\ (map\ stl\ \pi l)$ **shows** *next* $\varphi\ \pi l$
⟨*proof*⟩

lemma *next-elim*[*elim*]:
assumes *next* $\varphi\ \pi l$ **and** $\varphi\ (map\ stl\ \pi l) \Longrightarrow \chi$ **shows** χ
⟨*proof*⟩

lemma *until-introR*[*intro*]:
assumes $\psi\ \pi l$ **shows** *until* $\varphi\ \psi\ \pi l$
⟨*proof*⟩

lemma *until-introL*[*intro*]:
assumes φ: $\varphi\ \pi l$ **and** u: *until* $\varphi\ \psi\ (map\ stl\ \pi l)$
shows *until* $\varphi\ \psi\ \pi l$
⟨*proof*⟩

The elimination rules for until and eventually are induction rules.

lemma *until-induct*[*induct pred*: *until*, *consumes 1*, *case-names Base Step*]:
assumes u: *until* $\varphi\ \psi\ \pi l$
and b: $\bigwedge \pi l.\ \psi\ \pi l \Longrightarrow \chi\ \pi l$

and *i*: $\bigwedge \pi l.\ [\![\varphi\ \pi l;\ until\ \varphi\ \psi\ (map\ stl\ \pi l);\ \chi\ (map\ stl\ \pi l)]\!] \Longrightarrow \chi\ \pi l$
shows $\chi\ \pi l$
⟨*proof*⟩

lemma *until-unfold*:
$until\ \varphi\ \psi\ \pi l = (\psi\ \pi l \vee \varphi\ \pi l \wedge until\ \varphi\ \psi\ (map\ stl\ \pi l))$ (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *ev-introR*[*intro*]:
assumes $\varphi\ \pi l$ **shows** $ev\ \varphi\ \pi l$
⟨*proof*⟩

lemma *ev-introL*[*intro*]:
assumes $ev\ \varphi\ (map\ stl\ \pi l)$ **shows** $ev\ \varphi\ \pi l$
⟨*proof*⟩

lemma *ev-induct*[*induct pred*: *ev*, *consumes 1*, *case-names Base Step*]:
assumes $ev\ \varphi\ \pi l$ **and** $\bigwedge \pi l.\ \varphi\ \pi l \Longrightarrow \chi\ \pi l$
and $\bigwedge \pi l.\ [\![ev\ \varphi\ (map\ stl\ \pi l);\ \chi\ (map\ stl\ \pi l)]\!] \Longrightarrow \chi\ \pi l$
shows $\chi\ \pi l$
⟨*proof*⟩

lemma *ev-unfold*:
$ev\ \varphi\ \pi l = (\varphi\ \pi l \vee ev\ \varphi\ (map\ stl\ \pi l))$
⟨*proof*⟩

lemma *ev*: $ev\ \varphi\ \pi l \longleftrightarrow (\exists\ i.\ \varphi\ (map\ (sdrop\ i)\ \pi l))$
⟨*proof*⟩

The introduction rules for always and weak until are coinduction rules.

lemma *alw-coinduct*[*coinduct pred*: *alw*, *consumes 1*, *case-names Hyp*]:
assumes $\chi\ \pi l$
and $\bigwedge \pi l.\ \chi\ \pi l \Longrightarrow alw\ \varphi\ \pi l \vee (\varphi\ \pi l \wedge \chi\ (map\ stl\ \pi l))$
shows $alw\ \varphi\ \pi l$
⟨*proof*⟩

lemma *alw-elim*[*elim*]:
assumes $alw\ \varphi\ \pi l$
and $[\![\varphi\ \pi l;\ alw\ \varphi\ (map\ stl\ \pi l)]\!] \Longrightarrow \chi$
shows χ
⟨*proof*⟩

lemma *alw-destL*: $alw\ \varphi\ \pi l \Longrightarrow \varphi\ \pi l$ ⟨*proof*⟩
lemma *alw-destR*: $alw\ \varphi\ \pi l \Longrightarrow alw\ \varphi\ (map\ stl\ \pi l)$ ⟨*proof*⟩

lemma *alw-unfold*:
$alw\ \varphi\ \pi l = (\varphi\ \pi l \wedge alw\ \varphi\ (map\ stl\ \pi l))$
⟨*proof*⟩

lemma *alw*: *alw* φ $\pi l \longleftrightarrow (\forall\ i.\ \varphi\ (map\ (sdrop\ i)\ \pi l))$
⟨*proof*⟩

lemma *sdrop-imp-alw*:
assumes $\bigwedge\ i.\ (\bigwedge j.\ j \leq i \Longrightarrow \varphi\ [sdrop\ j\ \pi,\ sdrop\ j\ \pi']) \Longrightarrow \psi\ [sdrop\ i\ \pi,\ sdrop\ i\ \pi']$
shows *imp* (*alw* φ) (*alw* ψ) $[\pi,\ \pi']$
⟨*proof*⟩

lemma *wuntil-coinduct*[*coinduct pred*: *wuntil*, *consumes 1*, *case-names Hyp*]:
assumes χ: $\chi\ \pi l$
and *0*: $\bigwedge\ \pi l.\ \chi\ \pi l \Longrightarrow \psi\ \pi l \vee (\varphi\ \pi l \wedge \chi\ (map\ stl\ \pi l))$
shows *wuntil* φ ψ πl
⟨*proof*⟩

lemma *wuntil-elim*[*elim*]:
assumes *w*: *wuntil* φ ψ πl
and *1*: $\psi\ \pi l \Longrightarrow \chi$
and *2*: $[\![\varphi\ \pi l;\ wuntil\ \varphi\ \psi\ (map\ stl\ \pi l)]\!] \Longrightarrow \chi$
shows χ
⟨*proof*⟩

lemma *wuntil-unfold*:
wuntil φ ψ $\pi l = (\psi\ \pi l \vee \varphi\ \pi l \wedge wuntil\ \varphi\ \psi\ (map\ stl\ \pi l))$
⟨*proof*⟩

3.4 More derived operators

The conjunction of an arbitrary set of formulas:

definition *scon* ::
('*state*,'*aprop*) *sfmla set* ⇒ ('*state*,'*aprop*) *sfmla* **where**
scon φs $\pi l \equiv \forall\ \varphi \in \varphi s.\ \varphi\ \pi l$

lemma *mcon-intro*[*intro*!]:
assumes $\bigwedge\ \varphi.\ \varphi \in \varphi s \Longrightarrow \varphi\ \pi l$ **shows** *scon* φs πl
⟨*proof*⟩

lemma *scon-elim*[*elim*]:
assumes *scon* φs πl **and** $(\bigwedge\ \varphi.\ \varphi \in \varphi s \Longrightarrow \varphi\ \pi l) \Longrightarrow \chi$
shows χ
⟨*proof*⟩

Double-binding forall:

definition *fall2* $F \equiv fall\ (\lambda\ \pi.\ fall\ (F\ \pi))$

lemma *fall2-intro*[*intro*]:
assumes
$\bigwedge\ \pi\ \pi'.\ [\![wfp\ AP'\ \pi;\ wfp\ AP'\ \pi';$
$\pi l \neq [] \Longrightarrow stateOf\ \pi = stateOf\ (last\ \pi l);$
$\pi l = [] \Longrightarrow stateOf\ \pi = s0;$

stateOf $\pi' = stateOf\ \pi$
⟧
$\Longrightarrow$ $F\ \pi\ \pi'\ \pi l$
shows *fall2* $F\ \pi l$
⟨*proof*⟩

lemma *fall2-elim*[*elim*]:
assumes *fall2* $F\ \pi l$
and
$(\bigwedge \pi\ \pi'.\ [\![wfp\ AP'\ \pi;\ wfp\ AP'\ \pi';$
$\pi l \neq [] \Longrightarrow stateOf\ \pi = stateOf\ (last\ \pi l);\ \pi l = [] \Longrightarrow stateOf\ \pi = s0;$
stateOf $\pi' = stateOf\ \pi$
⟧
$\Longrightarrow F\ \pi\ \pi'\ \pi l)$
$\Longrightarrow \chi$
shows χ
⟨*proof*⟩

end-of-context Shallow

4 Noninterference à la Goguen and Meseguer

4.1 Goguen-Meseguer noninterference

Definition

locale *GM-sec-model* =
fixes *st0* :: *'St*
and *do* :: *'St* ⇒ *'U* ⇒ *'C* ⇒ *'St*
and *out* :: *'St* ⇒ *'U* ⇒ *'Out*
and *GH* :: *'U set*
and *GL* :: *'U set*
begin

Extension of "do" to sequences of pairs (user, command):

fun *doo* :: *'St* ⇒ (*'U* × *'C*) *list* ⇒ *'St* **where**
doo st [] = *st*
|*doo st* ((*u*,*c*) # *ucl*) = (*doo* (*do st u c*) *ucl*)

definition *purge* :: *'U set* ⇒ (*'U* × *'C*) *list* ⇒ (*'U* × *'C*) *list* **where**
purge G ucl ≡ *filter* (λ (*u*,*c*). *u* ∉ *G*) *ucl*

lemma *purge-Nil*[*simp*]: *purge G* [] = []
and *purge-Cons-in*[*simp*]: *u* ∉ *G* ⟹ *purge G* ((*u*,*c*) # *ucl*) = (*u*,*c*) # *purge G ucl*
and *purge-Cons-notIn*[*simp*]: *u* ∈ *G* ⟹ *purge G* ((*u*,*c*) # *ucl*) = *purge G ucl*
⟨*proof*⟩

lemma *purge-append*:
purge G (*ucl1* @ *ucl2*) = *purge G ucl1* @ *purge G ucl2*

⟨*proof*⟩

definition *nonint* :: *bool* **where**
nonint ≡ ∀ *ucl*. ∀ *u* ∈ *GL*. *out* (*doo st0 ucl*) *u* = *out* (*doo st0* (*purge GH ucl*)) *u*

end-of-context GM-sec-model

4.2 Specialized Kripke structures

As a preparation for representing noninterference in HyperCTL*, we define a specialized notion of Kripke structure. It is enriched with the following date: two binary state predicates f and g, intuitively capturing high-input and low-output equivalence, respectively; a set Sink of states immediately accessible from any state and such that, for the states in Sink, there exist self-transitions and f holds.

This specialized structure, represented by the locale Shallow-Idle, is an auxiliary that streamlines our proofs, easing the connection between finite paths (specific to Goguen-Meseguer noninterference) and infinite paths (specific to the HyperCTL* semantics). The desired Kripke structure produced from a Goguen-Meseguer model will actually be such a specialized structure.

locale *Shallow-Idle* = *Shallow S s0* δ *AP*
  for *S* :: *'state set* **and** *s0* :: *'state* **and** δ :: *'state* ⇒ *'state set*
  and *AP* :: *'aprop set*
  and *f* :: *'state* ⇒ *'state* ⇒ *bool* **and** *g* :: *'state* ⇒ *'state* ⇒ *bool*
  and *Sink* :: *'state set*
  +
  assumes *Sink-S*: *Sink* ⊆ *S*
  and *Sink*: ⋀ *s*. *s* ∈ *S* ⟹ ∃ *s'*. *s'* ∈ δ *s* ∩ *Sink*
  and *Sink-idle*: ⋀ *s*. *s* ∈ *Sink* ⟹ *s* ∈ δ *s*
  and *Sink-f*: ⋀ *s1 s2*. {*s1*,*s2*} ⊆ *Sink* ⟹ *f s1 s2*
begin

definition *toSink s* ≡ *SOME s'*. *s'* ∈ δ *s* ∩ *Sink*

lemma *toSink*: *s* ∈ *S* ⟹ *toSink s* ∈ δ *s* ∩ *Sink*
⟨*proof*⟩

lemma *fall2-imp-alw*:
fall2 (λ π' π πl. *imp* (*alw* (φ πl)) (*alw* (ψ πl)) (πl @ [π,π'])) []
⟷
(∀ π π'. *wfp AP'* π ∧ *wfp AP'* π' ∧ *stateOf* π = *s0* ∧ *stateOf* π' = *s0*
  ⟶ *imp* (*alw* (φ [])) (*alw* (ψ [])) [π,π']
)
⟨*proof*⟩

lemma *wfp-stateOf-shift-stake-sconst*:

fixes π i
defines $\pi 1 \equiv shift\ (stake\ (Suc\ i)\ \pi)\ (sconst\ (toSink\ (fst\ (\pi\ !!\ i)),\ L\ (toSink\ (fst\ (\pi\ !!\ i)))))$
assumes π: $wfp\ AP'\ \pi$
shows $wfp\ AP'\ \pi 1 \wedge stateOf\ \pi 1 = stateOf\ \pi$
⟨*proof*⟩

lemma *fall2-imp-alw-index*:
assumes *0*: $\bigwedge \pi\ \pi'.\ wfp\ AP'\ \pi \wedge wfp\ AP'\ \pi' \longrightarrow$
$\varphi\ []\ [\pi,\pi'] = f\ (stateOf\ \pi)\ (stateOf\ \pi') \wedge$
$\psi\ []\ [\pi,\pi'] = g\ (stateOf\ \pi)\ (stateOf\ \pi')$
shows
$fall2\ (\lambda\ \pi'\ \pi\ \pi l.\ imp\ (alw\ (\varphi\ \pi l))\ (alw\ (\psi\ \pi l))\ (\pi l\ @\ [\pi,\pi'])\ []$
$\longleftrightarrow$
$(\forall\ \pi\ \pi'.\ wfp\ AP'\ \pi \wedge wfp\ AP'\ \pi' \wedge stateOf\ \pi = s0 \wedge stateOf\ \pi' = s0$
$\longrightarrow$
$(\forall\ i.\ (\forall\ j \leq i.\ f\ (fst\ (\pi\ !!\ j))\ (fst\ (\pi'\ !!\ j))) \longrightarrow g\ (fst\ (\pi\ !!\ i))\ (fst\ (\pi'\ !!\ i)))$
)
(**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

end-of-context Shallow-Idle

4.3 Faithful representation as a HyperCTL* property

Starting with a Goguen-Meseguer model, we will produce a specialized Kripke structure and a shallow HyperCTL* formula. Then we we will prove that the structure satisfies the formula iff the Goguen-Meseguer model satisfies noninterference.

The Kripke structure has two kinds of states: "idle" states storing Goguen-Meseguer states, and normal states storing Goguen-Meseguer states, users and commands: the former will be used for synchronization and the latter for Goguen-Meseguer steps. The Kripke labels store user-command actions and user-output observations.

datatype $('St,'U,'C)\ state =$
$isIdle$: $Idle\ (getGMState$: $'St) \mid isState$: $State\ (getGMState$: $'St)\ (getGMUser$: $'U)\ (getGMCom$: $'C)$

datatype $('U,'C,'Out)\ aprop = Last\ 'U\ 'C \mid Obs\ 'U\ 'Out$

definition $getGMUserCom$ **where** $getGMUserCom\ s = (getGMUser\ s,\ getGMCom\ s)$

lemma $getGMUserCom[simp]$: $getGMUserCom\ (State\ st\ u\ c) = (u,c)$
⟨*proof*⟩

context *GM-sec-model*
begin

primrec $L :: ('St,'U,'C)\ state \Rightarrow ('U,'C,'Out)\ aprop\ set$ **where**
$L\ (Idle\ st) = \{Obs\ u'\ (out\ st\ u') \mid u'.\ True\}$
$|L\ (State\ st\ u\ c) = \{Last\ u\ c\} \cup \{Obs\ u'\ (out\ st\ u') \mid u'.\ True\}$

Get the Goguen-Meseguer state:

primrec *getGMState* **where**
getGMState (*Idle st*) = *st*
|*getGMState* (*State st u c*) = *st*

lemma *Last-in-L*[*simp*]: *Last u c* ∈ *L s* ⟷ (∃ *st*. *s* = *State st u c*)
⟨*proof*⟩

lemma *Obs-in-L*[*simp*]: *Obs u ou* ∈ *L s* ⟷ *ou* = *out* (*getGMState s*) *u*
⟨*proof*⟩

primrec δ :: ('*St*,'*U*,'*C*) *state* ⇒ ('*St*,'*U*,'*C*) *state set* **where**
δ (*Idle st*) = {*Idle st*} ∪ {*State* (*do st u*′ *c*′) *u*′ *c*′ | *u*′ *c*′. *True*}
|δ (*State st u c*) = {*Idle st*} ∪ {*State* (*do st u*′ *c*′) *u*′ *c*′ | *u*′ *c*′. *True*}

abbreviation *s0* **where** *s0* ≡ *State st0 any any*

definition *f* :: ('*a*, '*U*, '*b*) *state* ⇒ ('*c*, '*U*, '*b*) *state* ⇒ *bool*
where
f s s′ ≡
∀ *u c*. *u* ∉ *GH* ⟶ ((∃ *st*. *s* = *State st u c*) ⟷ (∃ *st*′. *s*′ = *State st*′ *u c*))

definition *g* :: ('*St*, '*a*, '*b*) *state* ⇒ ('*St*, '*c*, '*d*) *state* ⇒ *bool*
where
g s s′ ≡ ∀ *u1*. *u1* ∈ *GL* ⟶ *out* (*getGMState s*) *u1* = *out* (*getGMState s*′) *u1*

lemma *f-id*[*simp*,*intro*!]: *f s s* ⟨*proof*⟩

definition *Sink* :: ('*St*,'*U*,'*C*) *state set*
where
Sink = {*Idle st* | *st* . *True*}

end

sublocale *GM-sec-model* < *Shallow-Idle*
where *S* = *UNIV*::('*St*,'*U*,'*C*) *state set*
and *AP* = *UNIV* :: ('*U*,'*C*,'*Out*) *aprop set* **and** *AP*′ = *UNIV* :: ('*U*,'*C*,'*Out*) *aprop set*
and *s0* = *s0* **and** *L* = *L* **and** δ = δ **and** *f* = *f* **and** *g* = *g* **and** *Sink* = *Sink*
⟨*proof*⟩

context *GM-sec-model*
begin

lemma *apropsOf-L-stateOf*[*simp*]:
wfp AP′ π ⟹ *apropsOf* π = *L* (*stateOf* π)
⟨*proof*⟩

The equality of two states w.r.t. a given "last" user-command pair:

definition *eqOnUC* ::
nat ⇒ *nat* ⇒ *'U* ⇒ *'C* ⇒ ((*'St*,*'U*,*'C*) *state*,(*'U*,*'C*,*'Out*) *aprop*) *sfmla*
where
eqOnUC i i' u c ≡ *eq* (*atom* (*Last u c*) *i*) (*atom* (*Last u c*) *i'*)

The equality of two states w.r.t. all their "last" user-command pairs with the user not in GH:

definition *eqButGH* ::
nat ⇒ *nat* ⇒ ((*'St*,*'U*,*'C*) *state*,(*'U*,*'C*,*'Out*) *aprop*) *sfmla*
where
eqButGH i i' ≡ *scon* {*eqOnUC i i' u c* | *u c*. (*u*,*c*) ∈ (*UNIV* − *GH*) × *UNIV*}

The equality of two states w.r.t. a given "observed" user-observation pair:

definition *eqOnUOut* ::
nat ⇒ *nat* ⇒ *'U* ⇒ *'Out* ⇒ ((*'St*,*'U*,*'C*) *state*,(*'U*,*'C*,*'Out*) *aprop*) *sfmla*
where
eqOnUOut i i' u ou ≡ *eq* (*atom* (*Obs u ou*) *i*) (*atom* (*Obs u ou*) *i'*)

The equality of two states w.r.t. all their "observed" user-observation pairs with the user in GL:

definition *eqOnGL* ::
nat ⇒ *nat* ⇒ ((*'St*,*'U*,*'C*) *state*,(*'U*,*'C*,*'Out*) *aprop*) *sfmla*
where
eqOnGL i i' ≡ *scon* {*eqOnUOut i i' u ou* | *u ou*. (*u*,*ou*) ∈ *GL* × *UNIV*}

lemma *eqOnUC-0-Suc0*[*simp*]:
assumes *wfp AP' π* **and** *wfp AP' π'*
shows
eqOnUC 0 (*Suc 0*) *u c* [*π*, *π'*]
⟷
((∃ *st*. *stateOf π* = *State st u c*) =
(∃ *st'*. *stateOf π'* = *State st' u c*)
)
⟨*proof*⟩

lemma *eqOnUOut-0-Suc0*[*simp*]:
assumes *wfp AP' π* **and** *wfp AP' π'*
shows
eqOnUOut 0 (*Suc 0*) *u ou* [*π*, *π'*]
⟷
(*ou* = *out* (*getGMState* (*stateOf π*)) *u* ⟷
ou = *out* (*getGMState* (*stateOf π'*)) *u*
)
⟨*proof*⟩

The (shallow) noninterference formula – it will be proved equivalent to nonint, the original statement of noninterference.

definition *nonintSfmla* :: ((*'St*,*'U*,*'C*) *state*,(*'U*,*'C*,*'Out*) *aprop*) *sfmla* **where**
nonintSfmla ≡
fall2 (λ *π' π πl*.

```
  imp (alw (eqButGH (length πl) (Suc (length πl))))
      (alw (eqOnGL (length πl) (Suc (length πl))))
  (πl @ [π,π'])
 )
```

First, we show that nonintSfmla is equivalent to nonintSI, a variant of noninterference that speaks about Synchronized Infinite paths.

definition *nonintSI* :: *bool* **where**
nonintSI ≡
∀ π π'. *wfp UNIV* π ∧ *wfp UNIV* π' ∧ *stateOf* π = *s0* ∧ *stateOf* π' = *s0*
⟶
(∀ *i*. (∀ *j* ≤ *i*. *f* (*fst* (π !! *j*)) (*fst* (π' !! *j*))) ⟶ *g* (*fst* (π !! *i*)) (*fst* (π' !! *i*)))

lemma *nonintSfmla-nonintSI*: *nonintSfmla* [] ⟷ *nonintSI*
⟨*proof*⟩

In turn, nonintSI will be shown equivalent to nonintS, a variant speaking about Synchronized finite paths. To this end, we introduce a notion of well-formed finite path (wffp) – besides finiteness, another difference from the previously defined infinite paths is that, thanks to the fact that here AP coincides with AP', paths are mere sequences of states as opposed to pairs (state,set of atomic predicates).

inductive *wffp* :: ('*St*,'*U*,'*C*) *state list* ⇒ *bool*
where
Singl[*simp*,*intro*!]: *wffp* [*s*]
|
Cons[*intro*]:
⟦*s'* ∈ δ *s*; *wffp* (*s'* # *sl*)⟧
⟹
wffp (*s* # *s'* # *sl*)

lemma *wffp-induct2*[*consumes 1*, *case-names Singl Cons*]:
assumes *wffp sl*
and ⋀ *s*. *P* [*s*]
and ⋀ *s sl*. ⟦*hd sl* ∈ δ *s*; *wffp sl*; *P sl*⟧ ⟹ *P* (*s* # *sl*)
shows *P sl*
⟨*proof*⟩

definition *nonintS* :: *bool* **where**
nonintS ≡
∀ *sl sl'*. *wffp sl* ∧ *wffp sl'* ∧ *hd sl* = *s0* ∧ *hd sl'* = *s0* ∧
list-all2 f sl sl' ⟶ *g* (*last sl*) (*last sl'*)

lemma *wffp-NE*: **assumes** *wffp sl* **shows** *sl* ≠ []
⟨*proof*⟩

lemma *wffp*:
wffp sl ⟷ *sl* ≠ [] ∧ (∀ *i*. *Suc i* < *length sl* ⟶ *sl*!(*Suc i*) ∈ δ(*sl*!*i*))
(**is** *?L* ⟷ *?A* ∧ (∀ *i*. *?R i*))

⟨*proof*⟩

lemma *wffp-hdI*[*intro*]:
assumes *wffp sl* **and** *hd sl* ∈ δ *s*
shows *wffp* (*s* # *sl*)
⟨*proof*⟩

lemma *wffp-append*:
assumes *sl*: *wffp sl* **and** *sl1*: *wffp sl1* **and** *h*: *hd sl1* ∈ δ (*last sl*)
shows *wffp* (*sl* @ *sl1*)
⟨*proof*⟩

lemma *wffp-append-iff*:
wffp (*sl* @ *sl1*) ⟷
(*wffp sl* ∧ *sl1* = []) ∨
(*sl* = [] ∧ *wffp sl1*) ∨
(*wffp sl* ∧ *wffp sl1* ∧ *hd sl1* ∈ δ (*last sl*))
(**is** - ⟷ *?R*)
⟨*proof*⟩

lemma *wffp-to-wfp*:
assumes π-*def*: π = *map* (λ *s*. (*s*, *L s*)) *sl* @− *sconst* (*toSink* (*last sl*), *L* (*toSink* (*last sl*)))
assumes *sl*: *wffp sl*
shows
wfp UNIV π ∧
(∀ *i* < *length sl*. *sl* ! *i* = *fst* (π !! *i*)) ∧
(∀ *i* ≥ *length sl*. *fst* (π !! *i*) = *toSink* (*last sl*)) ∧
stateOf π = *hd sl*
⟨*proof*⟩

lemma *wffp-imp-appendL*: *wffp* (*sl1* @ *sl2*) ⟹ *sl1* ≠ [] ⟹ *wffp sl1*
⟨*proof*⟩

lemma *wffp-imp-appendR*: *wffp* (*sl1* @ *sl2*) ⟹ *sl2* ≠ [] ⟹ *wffp sl2*
⟨*proof*⟩

lemma *wffp-iff-map-Idle*:
assumes *wffp sl*
shows
∃ *n st*.
(*n* > *0* ∧ *sl* = *map Idle* (*replicate n st*)) ∨
(∃ *st1 u1 c1 sl1*. *sl* = *map Idle* (*replicate n st*) @ [*State st1 u1 c1*] @ *sl1*)
⟨*proof*⟩

lemma *wffp-cases3*[*elim*, *consumes 1*, *case-names Idle State Idle-State*]:
assumes *wffp sl*
obtains
n st **where**
n > *0* **and** *sl* = *map Idle* (*replicate n st*)

|
st u c sl1 **where**
sl = *State st u c* # *sl1* **and** *sl1* ≠ [] ⟹ *wffp sl1* ∧ *hd sl1* ∈ δ (*State st u c*)
|
n st u c sl1 **where**
n > *0* **and** *sl* = *map Idle* (*replicate n st*) @ [*State* (*do st u c*) *u c*] @ *sl1*
and *sl1* ≠ [] ⟹ *wffp sl1* ∧ *hd sl1* ∈ δ (*State* (*do st u c*) *u c*)
⟨*proof*⟩

lemma *wffp-cases2*[*elim*, *consumes 1*, *case-names Idle State*]:
assumes *wffp sl*
obtains
n st **where**
n > *0* **and** *sl* = *map Idle* (*replicate n st*)
|
n st st1 u c sl1 **where**
sl = *map Idle* (*replicate n st*) @ [*State st1 u c*] @ *sl1*
and *sl1* ≠ [] ⟹ *wffp sl1* ∧ *hd sl1* ∈ δ (*State st1 u c*)
⟨*proof*⟩

lemma *wffp-Idle-Idle*:
assumes *wffp* (*sl1* @ [*Idle st1*] @ [*Idle st2*] @ *sl2*)
shows *st2* = *st1*
⟨*proof*⟩

lemma *wffp-Idle-State*:
assumes *wffp* (*sl1* @ [*Idle st1*] @ [*State st2 u2 c2*] @ *sl2*)
shows *st2* = *st1* ∨ *st2* = *do st1 u2 c2*
⟨*proof*⟩

lemma *wffp-State-Idle*:
assumes *wffp* (*sl1* @ [*State st1 u1 c1*] @ [*Idle st2*] @ *sl2*)
shows *st2* = *st1*
⟨*proof*⟩

lemma *wffp-State-State*:
assumes *wffp* (*sl1* @ [*State st1 u1 c1*] @ [*State st2 u2 c2*] @ *sl2*)
shows *st2* = *do st1 u2 c2*
⟨*proof*⟩

lemma *wfp-to-wffp*:
assumes *sl-def*: *sl* = *map fst* (*stake i* π) **and** *i*: *i* > *0* **and** π: *wfp UNIV* π
shows
wffp sl ∧
(∀ *j* < *length sl*. *fst* (π !! *j*) = *sl* ! *j*) ∧
stateOf π = *hd sl*
⟨*proof*⟩

lemma *nonintSI-nonintS*: *nonintSI* ⟷ *nonintS*

⟨*proof*⟩

Finally, we show that nonintS is equivalent to standard noninterference (predicate nonint).

purgeIdle removes the idle steps from a finite path:

definition *purgeIdle* :: ('*St*, '*U*, '*C*) *state list* ⇒ ('*St*, '*U*, '*C*) *state list*
where *purgeIdle* ≡ *filter isState*

lemma *purgeIdle-simps*[*simp*]:
purgeIdle [] = []
purgeIdle ((*Idle st*) # *sl*) = *purgeIdle sl*
purgeIdle ((*State st u c*) # *sl*) = (*State st u c*) # *purgeIdle sl*
⟨*proof*⟩

lemma *purgeIdle-append*:
purgeIdle (*sl1* @ *sl2*) = *purgeIdle sl1* @ *purgeIdle sl2*
⟨*proof*⟩

lemma *purgeIdle-set-isState*:
assumes *s* ∈ *set* (*purgeIdle sl*)
shows *isState s*
⟨*proof*⟩

lemma *purgeIdle-Nil-iff*:
purgeIdle sl = [] ⟷ (∀ *s*∈*set sl*. ¬ *isState s*)
⟨*proof*⟩

lemma *purgeIdle-Cons-iff*:
purgeIdle sl = *s* # *sll*
⟷
(∃ *sl1 sl2*. *sl* = *sl1* @ *s* # *sl2* ∧
(∀ *s1*∈*set sl1*. ¬ *isState s1*) ∧ *isState s* ∧ *purgeIdle sl2* = *sll*)
⟨*proof*⟩

lemma *purgeIdle-map-Idle*[*simp*]:
purgeIdle (*map Idle s*) = []
⟨*proof*⟩

lemma *purgeIdle-replicate-Idle*[*simp*]:
purgeIdle (*replicate n* (*Idle st*)) = []
⟨*proof*⟩

lemma *wffp-purgeIdle-Nil*:
assumes *wffp sl* **and** *purgeIdle sl* = []
shows ∃ *n st*. *n* > *0* ∧ *sl* = *replicate n* (*Idle st*)
⟨*proof*⟩

lemma *wffp-hd-purgeIdle*:
assumes *wsl*: *wffp sl* **and** *psl*: *purgeIdle sl* ≠ []
and *ist*: *isState s* **and** *hsl*: *hd sl* ∈ δ *s*

shows *hd* (*purgeIdle sl*) ∈ δ *s*
⟨*proof*⟩

lemma *wffp-purgeIdle*:
assumes *wffp sl* **and** *purgeIdle sl* ≠ []
shows *wffp* (*purgeIdle sl*)
⟨*proof*⟩

lemma *isState-purgeIdle*:
(∃ *sl*. *purgeIdle sl* = *sll*) ⟷ *list-all isState sll*
⟨*proof*⟩

lemma *wffp-last-purgeIdle*:
assumes *wffp sl* **and** *purgeIdle sl* ≠ []
shows *getGMState* (*last* (*purgeIdle sl*)) = *getGMState* (*last sl*)
⟨*proof*⟩

lemma *wffp-isState-doo*:
assumes *wffp sl* **and** *list-all isState sl*
shows *doo* (*getGMState* (*hd sl*)) (*map getGMUserCom* (*tl sl*)) = *getGMState* (*last sl*)
⟨*proof*⟩

lemma *isState-hd-purgeIdle*:
assumes *wsl*: *wffp sl* **and** *ist*: *isState* (*hd sl*)
shows *purgeIdle sl* ≠ [] ∧ *hd* (*purgeIdle sl*) = *hd sl*
⟨*proof*⟩

lemma *wffp-isState-doo-purgeIdle*:
fixes *sl* **defines** *sll*: *sll* ≡ *purgeIdle sl*
assumes *wsl*: *wffp sl* **and** *ist*: *isState* (*hd sl*)
shows *doo* (*getGMState* (*hd sl*)) (*map getGMUserCom* (*tl sll*)) = *getGMState* (*last sl*)
⟨*proof*⟩

lemma *map-getGMUserCom-surj*:
assumes *isState s*
shows ∃ *sl*. *wffp sl* ∧ *list-all isState sl* ∧ *hd sl* = *s* ∧ *map getGMUserCom* (*tl sl*) = *ucl*
⟨*proof*⟩

lemma *purgeIdle-purge-ex*:
assumes *wffp sl* **and** *list-all isState sl* **and** *map getGMUserCom* (*tl sl*) = *ucl*
shows ∃ *sl'*. *hd sl'* = *ss'* ∧ *wffp sl'* ∧
list-all2 f (*tl sl*) (*tl sl'*) ∧
map getGMUserCom (*purgeIdle* (*tl sl'*)) = *purge GH ucl*
⟨*proof*⟩

lemma *purgeIdle-getGMUserCom-purge*:
fixes *sl sl'*
defines *ucl* ≡ *map getGMUserCom* (*purgeIdle* (*tl sl*))
and *ucl'* ≡ *map getGMUserCom* (*purgeIdle* (*tl sl'*))

assumes *wsl*: *wffp sl* **and** *wsl'*: *wffp sl'* **and** *f*: *list-all2 f sl sl'*
shows *purge GH ucl* = *purge GH ucl'*
⟨*proof*⟩

lemma *nonintS-iff-nonint*:
nonintS ⟷ *nonint*
⟨*proof*⟩

theorem *nonintSfmla-iff-nonint*:
nonintSfmla [] ⟷ *nonint*
⟨*proof*⟩

end-of-context GM-sec-model

5 Deep representation of HyperCTL* – syntax and semantics

5.1 Path variables and environments

datatype *pvar* = *Pvariable* (*natOf* : *nat*)

Deeply embedded (syntactic) formulas

datatype *'aprop dfmla* =
Atom 'aprop pvar |
Fls | *Neg 'aprop dfmla* | *Dis 'aprop dfmla 'aprop dfmla* |
Next 'aprop dfmla | *Until 'aprop dfmla 'aprop dfmla* |
Exi pvar 'aprop dfmla

Derived operators

definition *Tr* ≡ *Neg Fls*
definition *Con* φ ψ ≡ *Neg* (*Dis* (*Neg* φ) (*Neg* ψ))
definition *Imp* φ ψ ≡ *Dis* (*Neg* φ) ψ
definition *Eq* φ ψ ≡ *Con* (*Imp* φ ψ) (*Imp* ψ φ)
definition *Fall p* φ ≡ *Neg* (*Exi p* (*Neg* φ))
definition *Ev* φ ≡ *Until Tr* φ
definition *Alw* φ ≡ *Neg* (*Ev* (*Neg* φ))
definition *Wuntil* φ ψ ≡ *Dis* (*Until* φ ψ) (*Alw* φ)
definition *Fall2 p p'* φ ≡ *Fall p* (*Fall p'* φ)

lemmas *der-Op-defs* =
Tr-def Con-def Imp-def Eq-def Ev-def Alw-def Wuntil-def Fall-def Fall2-def

Well-formed formulas are those that do not have a temporal operator outside the scope of any quantifier – indeed, in HyperCTL* such a situation does not make sense, since the temporal operators refer to previously introduced/quantified paths.

primrec *wff* :: *'aprop dfmla* ⇒ *bool* **where**
wff (*Atom a p*) = *True*
|*wff Fls* = *True*
|*wff* (*Neg* φ) = *wff* φ

$|wff\ (Dis\ \varphi\ \psi) = (wff\ \varphi \wedge wff\ \psi)$
$|wff\ (Next\ \varphi) = False$
$|wff\ (Until\ \varphi\ \psi) = False$
$|wff\ (Exi\ p\ \varphi) = True$

The ability to pick a fresh variable

lemma *finite-fresh-pvar*:
assumes *finite* $(P :: pvar\ set)$
obtains p **where** $p \notin P$
⟨*proof*⟩

definition $getFresh :: pvar\ set \Rightarrow pvar$ **where**
$getFresh\ P \equiv SOME\ p.\ p \notin P$

lemma *getFresh*:
assumes *finite* P **shows** $getFresh\ P \notin P$
⟨*proof*⟩

The free-variables operator

primrec $FV :: 'aprop\ dfmla \Rightarrow pvar\ set$ **where**
$FV\ (Atom\ a\ p) = \{p\}$
$|FV\ Fls = \{\}$
$|FV\ (Neg\ \varphi) = FV\ \varphi$
$|FV\ (Dis\ \varphi\ \psi) = FV\ \varphi \cup FV\ \psi$
$|FV\ (Next\ \varphi) = FV\ \varphi$
$|FV\ (Until\ \varphi\ \psi) = FV\ \varphi \cup FV\ \psi$
$|FV\ (Exi\ p\ \varphi) = FV\ \varphi - \{p\}$

Environments

type-synonym $env = pvar \Rightarrow nat$

definition $eqOn\ P\ env\ env1 \equiv \forall\ p.\ p \in P \longrightarrow env\ p = env1\ p$

lemma *eqOn-Un*[*simp*]:
$eqOn\ (P \cup Q)\ env\ env1 \longleftrightarrow eqOn\ P\ env\ env1 \wedge eqOn\ Q\ env\ env1$
⟨*proof*⟩

lemma *eqOn-update*[*simp*]:
$eqOn\ P\ (env(p := \pi))\ (env1(p := \pi)) \longleftrightarrow eqOn\ (P - \{p\})\ env\ env1$
⟨*proof*⟩

lemma *eqOn-singl*[*simp*]:
$eqOn\ \{p\}\ env\ env1 \longleftrightarrow env\ p = env1\ p$
⟨*proof*⟩

context *Shallow*
begin

5.2 The semantic operator

The semantics will interpret deep (syntactic) formulas as shallow formulas. Recall that the latter are predicates on lists of paths – the interpretation will be parameterized by an environment mapping each path variable to a number indicating the index (in the list) for the path denoted by the variable.

The semantics will only be meaningful if the indexes of a formula's free variables are smaller than the length of the path list – we call this property "compatibility".

primrec *sem* :: *'aprop dfmla* ⇒ *env* ⇒ (*'state*,*'aprop*) *sfmla* **where**
sem (*Atom a p*) *env* = *atom a* (*env p*)
|*sem Fls env* = *fls*
|*sem* (*Neg φ*) *env* = *neg* (*sem φ env*)
|*sem* (*Dis φ ψ*) *env* = *dis* (*sem φ env*) (*sem ψ env*)
|*sem* (*Next φ*) *env* = *next* (*sem φ env*)
|*sem* (*Until φ ψ*) *env* = *until* (*sem φ env*) (*sem ψ env*)
|*sem* (*Exi p φ*) *env* = *exi* (λ *π πl*. *sem φ* (*env*(*p* := *length πl*)) (*πl* @ [*π*]))

lemma *sem-Exi-explicit*:
sem (*Exi p φ*) *env πl* ⟷
(∃ *π*. *wfp AP' π* ∧ *stateOf π* = (*if πl* ≠ [] *then stateOf* (*last πl*) *else s0*) ∧
sem φ (*env*(*p* := *length πl*)) (*πl* @ [*π*]))
⟨*proof*⟩

lemma *sem-derived*[*simp*]:
sem Tr env = *tr*
sem (*Con φ ψ*) *env* = *con* (*sem φ env*) (*sem ψ env*)
sem (*Imp φ ψ*) *env* = *imp* (*sem φ env*) (*sem ψ env*)
sem (*Eq φ ψ*) *env* = *eq* (*sem φ env*) (*sem ψ env*)
sem (*Fall p φ*) *env* = *fall* (λ *π πl*. *sem φ* (*env*(*p* := *length πl*)) (*πl* @ [*π*]))
sem (*Ev φ*) *env* = *ev* (*sem φ env*)
sem (*Alw φ*) *env* = *alw* (*sem φ env*)
sem (*Wuntil φ ψ*) *env* = *wuntil* (*sem φ env*) (*sem ψ env*)
⟨*proof*⟩

lemma *sem-Fall2*[*simp*]:
sem (*Fall2 p p' φ*) *env* =
fall2 (λ *π' π πl*. *sem φ* (*env* (*p* := *length πl*, *p'* := *Suc*(*length πl*))) (*πl* @ [*π*,*π'*]))
⟨*proof*⟩

Compatibility of a pair (environment,path list) on a set of variables means no out-or-range references:

definition *cpt P env πl* ≡ ∀ *p* ∈ *P*. *env p* < *length πl*

lemma *cpt-Un*[*simp*]:
cpt (*P* ∪ *Q*) *env πl* ⟷ *cpt P env πl* ∧ *cpt Q env πl*
⟨*proof*⟩

lemma *cpt-singl*[*simp*]:
cpt {*p*} *env πl* ⟷ *env p* < *length πl*
⟨*proof*⟩

lemma *cpt-map-stl*[*simp*]:
cpt P env $\pi l \Longrightarrow$ *cpt P env* (*map stl* πl)
⟨*proof*⟩

Next we prove that the semantics of well-formed formulas only depends on the interpretation of the free variables of a formula and on the current state of the last recorded path – we call the latter the "pointer" of the path list.

fun *pointerOf* :: (*'state*,*'aprop*) *path list* ⇒ *'state* **where**
pointerOf πl = (*if* $\pi l \neq []$ *then stateOf* (*last* πl) *else s0*)

Equality of two pairs (environment,path list) on a set of variables:

definition *eqOn* ::
pvar set ⇒ *env* ⇒ (*'state*,*'aprop*) *path list* ⇒ *env* ⇒ (*'state*,*'aprop*) *path list* ⇒ *bool*
where
eqOn P env πl *env1* $\pi l1 \equiv \forall\ p.\ p \in P \longrightarrow \pi l!(env\ p) = \pi l1!(env1\ p)$

lemma *eqOn-Un*[*simp*]:
eqOn (*P* ∪ *Q*) *env* πl *env1* $\pi l1 \longleftrightarrow$ *eqOn P env* πl *env1* $\pi l1 \wedge$ *eqOn Q env* πl *env1* $\pi l1$
⟨*proof*⟩

lemma *eqOn-singl*[*simp*]:
eqOn {*p*} *env* πl *env1* $\pi l1 \longleftrightarrow \pi l!(env\ p) = \pi l1!(env1\ p)$
⟨*proof*⟩

lemma *eqOn-map-stl*[*simp*]:
cpt P env $\pi l \Longrightarrow$ *cpt P env1* $\pi l1 \Longrightarrow$
eqOn P env πl *env1* $\pi l1 \Longrightarrow$ *eqOn P env* (*map stl* πl) *env1* (*map stl* $\pi l1$)
⟨*proof*⟩

lemma *cpt-map-sdrop*[*simp*]:
cpt P env $\pi l \Longrightarrow$ *cpt P env* (*map* (*sdrop i*) πl)
⟨*proof*⟩

lemma *cpt-update*[*simp*]:
assumes *cpt* (*P* − {*p*}) *env* πl
shows *cpt P* (*env*(*p* := *length* πl)) (πl @ [π])
⟨*proof*⟩

lemma *eqOn-map-sdrop*[*simp*]:
cpt V env $\pi l \Longrightarrow$ *cpt V env1* $\pi l1 \Longrightarrow$
eqOn V env πl *env1* $\pi l1 \Longrightarrow$ *eqOn V env* (*map* (*sdrop i*) πl) *env1* (*map* (*sdrop i*) $\pi l1$)
⟨*proof*⟩

lemma *eqOn-update*[*simp*]:
assumes *cpt* (*P* − {*p*}) *env* πl **and** *cpt* (*P* − {*p*}) *env1* $\pi l1$
shows
eqOn P (*env*(*p* := *length* πl)) (πl @ [π]) (*env1*(*p* := *length* $\pi l1$)) ($\pi l1$ @ [π])
⟷

eqOn $(P - \{p\})$ *env* πl *env1* $\pi l1$
⟨*proof*⟩

lemma *eqOn-FV-sem-NE*:
assumes *cpt* $(FV\ \varphi)$ *env* πl **and** *cpt* $(FV\ \varphi)$ *env1* $\pi l1$ **and** *eqOn* $(FV\ \varphi)$ *env* πl *env1* $\pi l1$
and $\pi l \neq []$ **and** $\pi l1 \neq []$ **and** *last* πl = *last* $\pi l1$
shows *sem* φ *env* πl = *sem* φ *env1* $\pi l1$
⟨*proof*⟩

The next theorem states that the semantics of a formula on an environment and a list of paths only depends on the pointer of the list of paths.

theorem *eqOn-FV-sem*:
assumes *wff* φ **and** *pointerOf* πl = *pointerOf* $\pi l1$
and *cpt* $(FV\ \varphi)$ *env* πl **and** *cpt* $(FV\ \varphi)$ *env1* $\pi l1$ **and** *eqOn* $(FV\ \varphi)$ *env* πl *env1* $\pi l1$
shows *sem* φ *env* πl = *sem* φ *env1* $\pi l1$
⟨*proof*⟩

corollary *FV-sem*:
assumes *wff* φ **and** $\forall\ p \in FV\ \varphi$. *env* p = *env1* p
and *cpt* $(FV\ \varphi)$ *env* πl **and** *cpt* $(FV\ \varphi)$ *env1* πl
shows *sem* φ *env* πl = *sem* φ *env1* πl
⟨*proof*⟩

As a consequence, the interpretation of a closed formula (i.e., a formula with no free variables) will not depend on the environment and, from the list of paths, will only depend on its pointer:

corollary *interp-closed*:
assumes *wff* φ **and** *FV* $\varphi = \{\}$ **and** *pointerOf* πl = *pointerOf* $\pi l1$
shows *sem* φ *env* πl = *sem* φ *env1* $\pi l1$
⟨*proof*⟩

Therefore, it makes sense to define the interpretation of a closed formula by choosing any environment and any list of paths such that its pointer is the initial state (e.g., the empty list) – knowing that the choices are irrelevant.

definition *semClosed* $\varphi \equiv$ *sem* φ (*any*::*env*) (*SOME* πl. *pointerOf* πl = *s0*)

lemma *semClosed*:
assumes φ: *wff* φ *FV* $\varphi = \{\}$ **and** *p*: *pointerOf* πl = *s0*
shows *semClosed* φ = *sem* φ *env* πl
⟨*proof*⟩

lemma *semClosed-Nil*:
assumes φ: *wff* φ *FV* $\varphi = \{\}$
shows *semClosed* φ = *sem* φ *env* []
⟨*proof*⟩

5.3 The conjunction of a finite set of formulas

This is defined by making the set into a list (by choosing any ordering of the elements) and iterating binary conjunction.

definition *Scon* :: *'aprop dfmla set* ⇒ *'aprop dfmla* **where**
Scon $\varphi s \equiv$ *foldr Con* (*asList* φs) *Tr*

lemma *sem-Scon*[*simp*]:
assumes *finite* φs
shows *sem* (*Scon* φs) *env* = *scon* ((λ φ. *sem* φ *env*) ' φs)
⟨*proof*⟩

lemma *FV-Scon*[*simp*]:
assumes *finite* φs
shows *FV* (*Scon* φs) = $\bigcup$ (*FV* ' φs)
⟨*proof*⟩

end-of-context Shallow

6 Noninterference for models with finitely many users, commands and outputs

In the Noninterference section, we showed how to express Goguen-Meseguer noninterference as a shallow HyperCTL* formula. Here we show that, if one assumes finiteness of the sets of users, commands and outputs, then one can express the property as (the denotation of) a syntactic formula. Note that we do *not* need to assume the state space finite – this is important for a potential application to infinite-state systems.

The Goguen-Meseguer security model with finiteness assumptions

locale *GM-sec-model-finite* = *GM-sec-model st0 do out*
  for *st0* :: *'St*
  and *do* :: *'St* ⇒ *'U* ⇒ *'C* ⇒ *'St*
  and *out* :: *'St* ⇒ *'U* ⇒ *'Out*
  +
  assumes *finite-U*: *finite* (*UNIV* :: *'U set*)
  and *finite-C*: *finite* (*UNIV* :: *'C set*)
  and *finite-Out*: *finite* (*UNIV* :: *'Out set*)
begin

lemma *finite-UminusGH*: *finite* (*UNIV* − *GH*)
⟨*proof*⟩

lemma *finite-GL*: *finite GL*
⟨*proof*⟩

definition *EqOnUC* ::
pvar ⇒ *pvar* ⇒ *'U* ⇒ *'C* ⇒ (*'U*,*'C*,*'Out*) *aprop dfmla*

where
EqOnUC p p′ u c ≡ *Eq* (*Atom* (*Last u c*) *p*) (*Atom* (*Last u c*) *p′*)

lemma *EqOnUC-eqOnUC*[*simp*]:
assumes *env p* = *i* **and** *env p′* = *i′*
shows *sem* (*EqOnUC p p′ u c*) *env* = *eqOnUC i i′ u c*
⟨*proof*⟩

definition *EqButGH* ::
pvar ⇒ *pvar* ⇒ (′*U*,′*C*,′*Out*) *aprop dfmla*
where
EqButGH p p′ ≡ *Scon* {*EqOnUC p p′ u c* | *u c*. (*u*,*c*) ∈ (*UNIV* − *GH*) × *UNIV*}

lemma *finite-EqButGH*:
finite {*EqOnUC p p′ u c* | *u c*. (*u*,*c*) ∈ (*UNIV* − *GH*) × *UNIV*} (**is** *finite ?K*)
⟨*proof*⟩

lemma *EqButGH-eqButGH*[*simp*]:
assumes *env p* = *i* **and** *env p′* = *i′*
shows *sem* (*EqButGH p p′*) *env* = *eqButGH i i′*
⟨*proof*⟩

lemma *FV-EqButGH*: *FV* (*EqButGH p p′*) ⊆ {*p*,*p′*} (**is** *?L* ⊆ *?R*)
⟨*proof*⟩

definition *EqOnUOut* ::
pvar ⇒ *pvar* ⇒ ′*U* ⇒ ′*Out* ⇒ (′*U*,′*C*,′*Out*) *aprop dfmla*
where
EqOnUOut p p′ u ou ≡ *Eq* (*Atom* (*Obs u ou*) *p*) (*Atom* (*Obs u ou*) *p′*)

lemma *EqOnUOut-eqOnUOut*[*simp*]:
assumes *env p* = *i* **and** *env p′* = *i′*
shows *sem* (*EqOnUOut p p′ u ou*) *env* = *eqOnUOut i i′ u ou*
⟨*proof*⟩

definition *EqOnGL* ::
pvar ⇒ *pvar* ⇒ (′*U*,′*C*,′*Out*) *aprop dfmla*
where
EqOnGL p p′ ≡ *Scon* {*EqOnUOut p p′ u ou* | *u ou*. (*u*,*ou*) ∈ *GL* × *UNIV*}

lemma *finite-EqOnGL*:
finite {*EqOnUOut p p′ u ou* | *u ou*. (*u*,*ou*) ∈ *GL* × *UNIV*} (**is** *finite ?K*)
⟨*proof*⟩

lemma *EqOnGL-eqOnGL*[*simp*]:
assumes *env p* = *i* **and** *env p′* = *i′*
shows *sem* (*EqOnGL p p′*) *env* = *eqOnGL i i′*
⟨*proof*⟩

lemma *FV-EqOnGL*: *FV* (*EqOnGL p p′*) ⊆ {*p*,*p′*} (**is** *?L* ⊆ *?R*)
⟨*proof*⟩

definition *p0* = *getFresh* {}
definition *p1* = *getFresh* {*p0*}

lemma *p0p1*[*simp*]: *p0* ≠ *p1* ⟨*proof*⟩

definition *nonintDfmla* :: ('*U*,'*C*,'*Out*) *aprop dfmla* **where**
nonintDfmla ≡
Fall2 p0 p1 (*Imp* (*Alw* (*EqButGH p0 p1*)) (*Alw* (*EqOnGL p0 p1*)))

lemma *sem-nonintDfmla*: *sem nonintDfmla env* = *nonintSfmla*
⟨*proof*⟩

lemma *wff-nonintDfmla*[*simp*]: *wff nonintDfmla*
⟨*proof*⟩

lemma *closed-nonintDfmla*[*simp*]: *FV nonintDfmla* = {}
⟨*proof*⟩

In the end, we obtain that the semantics of the closed (syntactic) formula nonintDfmla expresses noninterference faithfully:

theorem *semClosed-nonintDfmla*: *semClosed nonintDfmla* = *nonint*
⟨*proof*⟩

end-of-context GM-sec-model-finite