A Note on Expressing Policy Rules in Fusion Logic

Antonio Cau
STRL

November 1, 2011

Abstract

We have before expressed our policy rules in Interval Temporal Logic. This makes it possible to verify properties over those policies. Recently we have a model checker for Fusion Logic a logic as expressive as Propositional Interval Temporal Logic but which a very much restricted syntax. This note explains why expressing strong policy rules can be problematic in Fusion Logic and describes a way of solving this problem.

1 Propositional Interval Temporal Logic

Propositional Interval Temporal Logic (PITL) is a discrete, linear temporal logic which includes a basic construct for sequential composition and an analog of Kleene star[1].

1.1 Syntax of PITL

The syntax is described in Table 1 where

- skip is an interval (sequence) of 2 states.
- $f_1 ; f_2$ is called 'f_1 chop f_2' and denotes sequential composition of two intervals.
- f^* is called 'f chopstar' and denotes finite iteration of an interval.

1.2 Derived PITL formulae

$\bigcirc f \mathrel{\widehat{=}} \mathsf{skip} ; f$	next f, f holds from the next state.
$\textcircled{w} f \mathrel{\widehat{=}} \neg \bigcirc \neg f$	weak next f, f holds from the next state or interval is empty.
$\mathsf{more} \mathrel{\widehat{=}} \bigcirc \mathsf{true}$	unit interval, i.e., interval with ≥ 2 states.
$\mathsf{empty} \mathrel{\widehat{=}} \neg \mathsf{more}$	empty interval, i.e., an one state interval.
$\Diamond f \mathrel{\widehat{=}} \mathsf{true} ; f$	sometimes f, i.e., any interval such that f holds over a suffix of that interval.
$\Box f \mathrel{\widehat{=}} \neg \Diamond \neg f$	always f, i.e., any interval such that f holds for all suffixes of that interval.
$\textcircled{i} f \mathrel{\widehat{=}} f ; \mathsf{true}$	diamond-i f, i.e., any interval such that f holds over a prefix of that interval.

[1]Note: we restrict ourselves to finite time only in this note.

PITL formulae
$f ::= \quad p \mid \neg f \mid f_1 \vee f_2 \mid \mathsf{skip} \mid f_1 ; f_2 \mid f^*$

Table 1: Propositional Interval temporal Logic

$\boxdot\, f \mathrel{\widehat{=}} \neg(\diamondsuit \neg f)$ box-i f, i.e., any interval such that f holds for all prefixes of that interval.

$\diamondsuit\, f \mathrel{\widehat{=}} \mathsf{true}\,;f\,;\mathsf{true}$ diamond-a f, i.e., any interval such that f holds over a subinterval of that interval.

$\boxdot\, f \mathrel{\widehat{=}} \neg(\diamondsuit \neg f)$ box-i f, i.e., any interval such that f holds for all subintervals of that interval.

$\mathsf{fin}\ f \mathrel{\widehat{=}} \Box(\mathsf{more} \vee f)$ final f, i.e., f holds in the final state.

$\mathsf{len}(0) \mathrel{\widehat{=}} \mathsf{empty}$ interval of length zero.

$\mathsf{len}(n+1) \mathrel{\widehat{=}} \mathsf{skip}\,;\mathsf{len}(n)$ interval of length $n+1$, for $n \geq 0$.

1.3 Semantics of PITL

The main semantic notion is interval which is a sequence of states. Let Σ denotes the set of states, Σ^+ denote the set of non-empty finite sequences of states, σ denote an interval, $\sigma \in \Sigma^+$. Let $[\![\ ]\!]$ be the "meaning" (semantic) function from PITL Formulae $\times\ \Sigma^+$ to $\{\mathrm{tt}, \mathrm{ff}\}$.

- $[\![p]\!]_\sigma = \mathrm{tt}$ iff $\sigma_0(p) = \mathrm{tt}$
- $[\![\neg f]\!]_\sigma = \mathrm{tt}$ iff not $[\![f]\!]_\sigma = \mathrm{tt}$
- $[\![f_1 \vee f_2]\!]_\sigma = \mathrm{tt}$ iff $[\![f_1]\!]_\sigma = \mathrm{tt}$ or $[\![f_2]\!]_\sigma = \mathrm{tt}$
- $[\![\mathsf{skip}]\!]_\sigma = \mathrm{tt}$ iff $|\sigma| = 1$ where $|\sigma|$ denotes length of σ and is defined as number of states minus 1.
- $[\![f_1\,;f_2]\!]_\sigma = \mathrm{tt}$ iff (exists a k, s.t. $([\![f_1]\!]_{\sigma_0 \ldots \sigma_k} = \mathrm{tt}$ and $[\![f_2]\!]_{\sigma_k \ldots \sigma_{|\sigma|}} = \mathrm{tt})$
- $[\![f^*]\!]_\sigma = \mathrm{tt}$ iff
  (exist $l_0, \ldots, l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and
  for all $0 \leq i < n, l_i < l_{i+1}$ and $[\![f]\!]_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}} = \mathrm{tt}$)

For any interval σ and PITL formula f, if $[\![f]\!]_\sigma = \mathrm{tt}$, then f is said to be *satisfiable*. A PITL formula f satisfied by all intervals is *valid*, denoted as $\models f$.

Note: $\models f$ is *valid* iff $\neg f$ is not *satisfiable*, i.e., we can express validity of a PITL formula in terms of satisfiability.

2 Policy rules in PITL

Semantics of rules Policy rules define the behaviour of the access control variables. The consequence of a rule determines the type of the rule and the subjects, objects and actions the rule applies to. The operator *always-followed-by* [4] is used to capture the relation between the premise of a rule and its consequence. Let us first define the semantics of a premise.

The syntax that is used in the premise is actually a subset of ITL formulae. The semantics of a rule premise is then as follows:

$$
\begin{aligned}
[\![pr_1;pr_2]\!] &\mathrel{\widehat{=}} [\![pr_1]\!]\,;[\![pr_2]\!] \\
[\![pr_1 \text{ and } pr_2]\!] &\mathrel{\widehat{=}} [\![pr_1]\!] \wedge [\![pr_2]\!] \\
[\![pr_1 \text{ or } pr_2]\!] &\mathrel{\widehat{=}} [\![pr_1]\!] \vee [\![pr_2]\!] \\
[\![\text{sometime } pr]\!] &\mathrel{\widehat{=}} \diamondsuit [\![pr]\!] \\
[\![\text{always } pr]\!] &\mathrel{\widehat{=}} \Box [\![pr]\!] \\
[\![\text{keep } pr]\!] &\mathrel{\widehat{=}} \boxdot\ (\mathsf{skip} \supset [\![pr]\!]) \\
[\![\text{if } be \text{ then } pr_1 \text{ else } pr_2]\!] &\mathrel{\widehat{=}} ([\![be]\!] \wedge [\![pr_1]\!]) \vee (\neg [\![be]\!] \wedge [\![pr_2]\!]) \\
[\![e : pr]\!] &\mathrel{\widehat{=}} \mathsf{len}([\![e]\!]) \wedge [\![pr]\!]
\end{aligned}
$$

	state formulae
$W ::=$	$\mathsf{true} \mid p \mid W_1 \vee W_2 \mid \neg W$
	transition formulae
$T ::=$	$W \mid \bigcirc W \mid T_1 \vee T_2 \mid \neg T$
	fusion expressions
$E ::=$	$\mathsf{test}(W) \mid \mathsf{step}(T) \mid E_1 \vee E_2 \mid E_1 \,;\, E_2 \mid E^*$
	fusion logic formulae
$F ::=$	$\mathsf{true} \mid p \mid \neg F \mid F_1 \vee F_2 \mid \langle E \rangle F$

Table 2: Syntax of FL

The operator *always followed by* is defined as follows:

$$f \;\mapsto\; w \mathrel{\widehat{=}}\boxdot_{\mathrm{a}} (f \;\supset\; \mathsf{fin}\,(w)) \tag{1}$$

The intuition of the operator is that whenever f holds for a prefix interval then w holds in the last state of that interval.

The semantics of individual rule is then defined as follows:

$$\begin{array}{l} [\![\text{allow } (su, ob, ac) \text{ when } pr]\!] \;\widehat{=}[\![pr]\!] \;\mapsto\; \text{autho}^{+}([\![su]\!], [\![ob]\!], [\![ac]\!]) \\ [\![\text{deny } (su, ob, ac) \text{ when } pr]\!] \;\widehat{=}[\![pr]\!] \;\mapsto\; \text{autho}^{-}([\![su]\!], [\![ob]\!], [\![ac]\!]) \\ [\![\text{decide } (su, ob, ac) \text{ when } pr]\!] \widehat{=}[\![pr]\!] \;\mapsto\; \text{autho}([\![su]\!], [\![ob]\!], [\![ac]\!]) \end{array}$$

The operator *strong always followed by* is defined as follows:

$$f \;\leftrightarrow\; w \mathrel{\widehat{=}}\boxdot_{\mathrm{a}} (f \equiv \mathsf{fin}\,(w)) \tag{2}$$

The intuition of the operator is that f holds for a prefix interval iff w holds in the last state of that interval.

The 'strong always followed by' is used in the verification of properties of policies. So when one wants to use a model checker one has to use a rules with this operator.

3 Fusion Logic

We introduce Fusion Logic (FL) in order to model check policy rules, i.e., built a BDD based model for them.

Fusion Logic augments conventional Propositional Temporal Logic (PTL) [1] with the fusion operator. Note: the fusion-operator $\langle E \rangle F$ is basically a "chop" that does not have an explicit negation on the left hand (as fusion expression) side except in $\mathsf{test}(\,)$ and $\mathsf{step}(\,)$. The expressiveness of FL is the same as PITL. The main differences concern computational complexity, naturalness of expression for analytical purposes, and succinctness. Fusion Logic is closely related to Propositional Dynamic Logic (PDL) [2].

3.1 Syntax

Let p be a propositional variable. We introduce the four syntactic categories of Fusion Logic in Table 2.

Note: the fusion-operator $\langle E \rangle F$ is basically a 'chop' that does not have a negation on the left hand side.

3.2 Derived FL formulae

Before we give our policy rule construct in Fusion Logic we first introduce some derived fusion expressions.

$\mathsf{len}_e(0) \mathrel{\widehat{=}} \mathsf{test}(\mathsf{true})$ interval of length 0 (one state interval).

$\mathsf{len}_e(n+1) \mathrel{\widehat{=}} \mathsf{step}(\mathsf{true}) \,;\, \mathsf{len}_e(n)$ interval of length $n+1$, for $n \geq 0$.

$\mathsf{skip}_e \mathrel{\widehat{=}} \mathsf{len}_e(1)$ unit interval, i.e., interval of length 1 (two state interval).

$\bigcirc_e E \mathrel{\widehat{=}} \mathsf{skip}_e \,;\, E$ next E, E holds from the next state.

$\mathsf{finite}_e \mathrel{\widehat{=}} \mathsf{skip}_e^*$ finite interval, i.e., any interval of finite length.

$\mathsf{more}_e \mathrel{\widehat{=}} \bigcirc_e \mathsf{finite}_e$ non-empty interval, i.e., any interval of length at least one.

$\Diamond_e E \mathrel{\widehat{=}} \mathsf{finite}_e \,;\, E$ sometimes E, i.e., any interval such that E holds over a suffix of that interval.

We also introduce the following derived constructs which will enable us to express policy rules in FL.

$F_1 \wedge F_2 \mathrel{\widehat{=}} \neg(\neg F_1 \vee \neg F_2)$ the 'and' of two fusion logic formulae defined using the Morgan.

$\bigcirc F \mathrel{\widehat{=}} \langle \mathsf{skip}_e \rangle F$ next F, F holds from the next state.

$\mathsf{more} \mathrel{\widehat{=}} \bigcirc \mathsf{true}$ non-empty interval, i.e., any interval of length at least one.

$\mathsf{empty} \mathrel{\widehat{=}} \neg\, \mathsf{more}$ empty interval, i.e., any interval of length zero (just one state).

$\mathsf{skip} \mathrel{\widehat{=}} \bigcirc \mathsf{empty}$ unit interval, i.e., any interval of length 1.

$\Diamond F \mathrel{\widehat{=}} \langle \mathsf{finite}_e \rangle F$ sometimes F, i.e., any interval such that F holds over a suffix of that interval.

$\Box F \mathrel{\widehat{=}} \neg \Diamond \neg F$ always F, i.e., any interval such that F holds for all suffixes of that interval.

$\mathsf{fin}\ F \mathrel{\widehat{=}} \Box(\mathsf{more} \vee F)$ final F, i.e., F holds in the final state.

Policy rules can now be expressed as follows

$$E \ \mapsto\ W \mathrel{\widehat{=}} \neg(\langle \mathsf{finite}_e \,;\, E \rangle \neg W)$$

As can be seen the premise of a policy rule is a fusion expression. If the premise is a state formula we can expressed the policy rule as follows

$$W_1 \ \mapsto\ W_2 \mathrel{\widehat{=}} \neg(\langle \mathsf{finite}_e \,;\, \mathsf{test}(W_1) \,;\, \mathsf{finite}_e \rangle \neg W_2)$$

3.3 Semantics of FL

A state is a mapping $State : Exp$ from the set of propositional variables Var to the set of values $\{\mathrm{tt}, \mathrm{ff}\}$. An interval σ is a finite sequence of states

$$\sigma : \sigma_0 \sigma_1 \sigma_2 \ldots \sigma_{|\sigma|}$$

where $|\sigma|$ denotes the length of an interval σ and is equal to the number of states minus 1. Let Σ denote the set of all possible intervals.

Let $[\![\ ]\!]$ be the "meaning" function from $Formulae \times \Sigma$ to $\{\mathrm{tt}, \mathrm{ff}\}$ and let $\sigma = \sigma_0 \sigma_1 \ldots, \sigma_{|\sigma|}$ be an interval then (i) $\sigma_0 \ldots \sigma_k$ (where $0 \leq k \leq |\sigma|$) denotes a *prefix* interval of σ, (ii) $\sigma_k \ldots \sigma_{|\sigma|}$ (where $0 \leq k \leq |\sigma|$) denotes a *suffix* interval of σ and (iii) $\sigma_k \ldots \sigma_l$ (where $0 \leq k \leq l \leq |\sigma|$) denotes a *sub* interval of σ. For $x \in \{W, T, E, F\}$, $y \in \{W, T, F\}$ and $z \in \{E, F\}$.

- $[\![\mathsf{true}]\!]_\sigma = \mathrm{tt}$

- $[\![p]\!]_\sigma = \mathrm{tt}$ iff $\sigma_0(p) = \mathrm{tt}$
- $[\![x_1 \vee x_2]\!]_\sigma = \mathrm{tt}$ iff $[\![x_1]\!]_\sigma = \mathrm{tt}$ or $[\![x_2]\!]_\sigma = \mathrm{tt}$
- $[\![\neg y]\!]_\sigma = \mathrm{tt}$ iff not $[\![y]\!]_\sigma = \mathrm{tt}$
- $[\![\bigcirc W]\!]_\sigma = \mathrm{tt}$ iff $[\![W]\!]_{\sigma_1 \ldots \sigma_{|\sigma|}}$ and $|\sigma| > 0$
- $[\![\mathsf{test}(W)]\!]_\sigma = \mathrm{tt}$ iff $[\![W]\!]_{\sigma_0}$ and $|\sigma| = 0$
- $[\![\mathsf{step}(T)]\!]_\sigma = \mathrm{tt}$ iff $[\![T]\!]_{\sigma_0 \ldots \sigma_1}$ and $|\sigma| = 1$
- $[\![E\,;z]\!]_\sigma = \mathrm{tt}$ iff exists a k, s.t. $[\![E]\!]_{\sigma_0 \ldots \sigma_k} = \mathrm{tt}$ and $[\![z]\!]_{\sigma_k \ldots \sigma_{|\sigma|}} = \mathrm{tt}$
- $[\![E^*]\!]_\sigma = \mathrm{tt}$ iff
  (exist $l_0, \ldots, l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and
  for all $0 \leq i < n, l_i < l_{i+1}$ and $[\![E]\!]_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}} = \mathrm{tt}$)
- $[\![\langle E \rangle F]\!]_\sigma = \mathrm{tt}$ iff $[\![E\,;F]\!]_\sigma = \mathrm{tt}$

For any interval σ and Fusion Logic formula F, if $[\![F]\!]_\sigma = \mathrm{tt}$, then F is said to be *satisfiable*. A Fusion Logic formula F satisfied by all intervals is *valid*, denoted as $\models F$.

Note: $\models F$ is *valid* iff $\neg F$ is not *satisfiable*, i.e., we can express validity of a formula in terms of satisfiability.

4 Policy rules in FL

We want to investigate whether it is possible to express the 'strong always followed by' in Fusion Logic.

The following are a few well know lemmas

Lemma 1

a	$(\neg(p \equiv q)) \equiv ((\neg p) \equiv q)$
b	$\mathsf{fin}\ w \equiv \mathsf{true}\,;(\mathsf{empty} \wedge w)$
c	$\neg(\mathsf{fin}\ w) \equiv \mathsf{true}\,;(\mathsf{empty} \wedge \neg w)$
d	$w \equiv (\mathsf{empty} \wedge w)\,;\mathsf{true}$
e	$((\neg f) \equiv \mathsf{fin}\,(w)) \equiv ((\neg f \wedge \mathsf{fin}\,(w)) \vee (f \wedge \neg\,\mathsf{fin}\,(w)))$
f	$(f \wedge \neg\,\mathsf{fin}\,(w)) \equiv f\,;(\mathsf{empty} \wedge \neg w)$
g	$(\neg f \wedge \mathsf{fin}\,(w)) \equiv \neg f\,;(\mathsf{empty} \wedge w)$
h	$\boxdot_a\,(f) \equiv \Box(\boxdot_i\ f)$
i	$f_0\,;(f_1 \vee f_2)\,;f_3 \equiv f_0\,;f_1\,;f_3 \vee f_0\,;f_2\,;f_3$

Proof 1 *(a) – (e) have been proven with the FLCHECK tool. (f) – (i) are proven with the prover9 tool.*

Now we will rewrite $f \mapsto w$ as follows:

$f \mapsto w$ using definition of 'strong always followed by':

$$\boxdot_a\,(f \equiv \mathsf{fin}\,(w))$$

Using Lemma1 (h):

$$\Box(\boxdot_i\,(f \equiv \mathsf{fin}\,(w)))$$

Using definition of $\Box$:

$$\neg(\mathsf{true}\,;\neg(\boxdot_i\,(f \equiv \mathsf{fin}\,(w))))$$

Using definition of $\boxdot$:

$$\neg(\mathsf{true}\,;\neg((f \equiv \mathsf{fin}\,(w)))\,;\mathsf{true})$$

Using Lemma1(a):

$$\neg(\mathsf{true}\,;(((\neg f) \equiv \mathsf{fin}\,(w)))\,;\mathsf{true})$$

Using Lemma1(e):

$$\neg(\mathsf{true}\,;((\neg f \wedge \mathsf{fin}\,(w)) \vee (f \wedge \neg\,\mathsf{fin}\,(w)))\,;\mathsf{true})$$

Using Lemma1(f) and (g):

$$\neg(\mathsf{true}\,;(\neg f\,;(\mathsf{empty} \wedge w) \vee f\,;(\mathsf{empty} \wedge \neg\, w))\,;\mathsf{true})$$

Using Lemma1(i):

$$\neg(\mathsf{true}\,;(\neg f\,;(\mathsf{empty} \wedge w))\,;\mathsf{true} \vee \mathsf{true}\,;(f\,;(\mathsf{empty} \wedge \neg\, w))\,;\mathsf{true})$$

Using the De Morgan:

$$\neg(\mathsf{true}\,;(\neg f)\,;(\mathsf{empty} \wedge w)\,;\mathsf{true}) \wedge \neg(\mathsf{true}\,;f\,;(\mathsf{empty} \wedge \neg\, w)\,;\mathsf{true})$$

Using Lemma1(d):

$$\neg(\mathsf{true}\,;(\neg f)\,;w) \wedge \neg(\mathsf{true}\,;f\,;(\neg\, w))$$

We can easily see that $\neg(\mathsf{true}\,;f\,;(\neg\, w))$ can be written in FL if f is an fusion expression, i.e., $\neg(<\mathsf{finite}_e\,;f>(\neg\, w))$.

However, $\neg(\mathsf{true}\,;(\neg f)\,;w)$ is problematic because using the same as above would yield: $\neg(<\mathsf{finite}_e\,;(\neg f)>(w))$. We must must devise a strategy that can rewrite $(\neg f)$ (where f is a fusion expression) into a fusion expression without negation.

Lemma 2 can be used to rewrite the negation of an fusion expression. The idea is that a fusion expression starts either with a 'test' or a 'step'. You can use Lemma 2(a) and (b) to rewrite those kinds of fusion expression. Lemma 2(c) deals with the negation of an 'or' type of fusion expression where both 'or' branches start with a test. Lemma 2(d) deals with the case that one branch start with a 'test' and the other branch with a 'step'. One basically only rewrites the branch with the 'test'. Lemma 2(e) is similar to (d) but now the 'test' branch has only one 'test'. Lemma 2(f) deals with the case that both branches in the 'or' start with a 'step'. For the 'chop' and 'chopstar' fusion operator one can use again Lemma 2(a) and (b).

Lemma 2

a $\neg(\mathsf{test}(W)\,;E) \equiv (\mathsf{test}(\neg\, W)\,;\mathsf{finite}_e) \vee \neg\, E$

b $\neg(\mathsf{step}(T)\,;E) \equiv (\mathsf{step}(\neg\, T)\,;\mathsf{finite}_e) \vee \mathsf{test}(\mathsf{true}) \vee (\mathsf{step}(\mathsf{true})\,;\neg\, E)$

c $\neg((\mathsf{test}(W_0)\,;E_0) \vee (\mathsf{test}(W_1)\,;E_1)) \equiv (\mathsf{test}(\neg\, W_0 \wedge \neg\, W_1)\,;\mathsf{finite}_e) \vee$
$(\mathsf{test}(\neg\, W_0)\,;\neg\, E_1) \vee (\mathsf{test}(\neg\, W_1)\,;\neg\, E_0) \vee \neg(E_0 \vee E_1)$

d $\neg((\mathsf{test}(W)\,;E_0) \vee (\mathsf{step}(T)\,;E_1)) \equiv$
$(\mathsf{test}(W)\,;\mathsf{step}(\neg\, T)\,;\mathsf{finite}_e) \vee (\mathsf{test}(\neg\, W)\,;\mathsf{step}(\mathsf{true})\,;\neg\, E_1) \vee$
$\mathsf{test}(\neg\, W) \vee \neg(E_0 \vee \mathsf{step}(T)\,;E_1)$

e $\neg(\mathsf{test}(\mathsf{true}) \vee (\mathsf{step}(T)\,;E_1)) \equiv$
$(\mathsf{step}(\neg\, T)\,;\mathsf{finite}_e) \vee (\mathsf{step}(\mathsf{true})\,;\neg\, E_1)$

f $\neg((\mathsf{step}(T_0)\,;E_0) \vee (\mathsf{step}(T_1)\,;E_1)) \equiv$
$\mathsf{test}(\mathsf{true}) \vee (\mathsf{step}(\neg\, T_0 \wedge \neg\, T_1)\,;\mathsf{finite}_e) \vee$
$(\mathsf{step}(\neg\, T_0)\,;\neg\, E_1) \vee (\mathsf{step}(\neg\, T_1)\,;\neg\, E_0) \vee$
$(\mathsf{step}(\mathsf{true})\,;\neg(E_0 \vee E_1))$

Proof 2

(a)*:*

$$\neg(\mathsf{test}(W)\,;E)$$

Using basic ITL reasoning:

$$\neg((\mathsf{test}(W)\,;\mathsf{finite}_e) \wedge E)$$

Using De Morgan:

$$\neg(\mathsf{test}(W)\,;\mathsf{finite}_e) \vee (\neg E)$$

Using basic reasoning of test:

$$(\mathsf{test}(\neg W)\,;\mathsf{finite}_e) \vee (\neg E)$$

***(b)**:*

$$\neg(\mathsf{step}(T)\,;E)$$

Definition of step:

$$\neg((\mathsf{skip} \wedge T)\,;E)$$

Using basic ITL reasoning:

$$\neg(((\mathsf{skip} \wedge T)\,;\mathsf{finite}_e) \wedge (\mathsf{skip}\,;E))$$

Using De Morgan:

$$\neg((\mathsf{skip} \wedge T)\,;\mathsf{finite}_e) \vee \neg(\mathsf{skip}\,;E)$$

Using basic ITL reasoning:

$$(\mathsf{test}(\mathsf{true}) \vee ((\mathsf{skip} \wedge \neg T)\,;\mathsf{finite}_e)) \vee \neg(\mathsf{skip}\,;E)$$

Using basic ITL reasoning:

$$(\mathsf{test}(\mathsf{true}) \vee ((\mathsf{skip} \wedge \neg T)\,;\mathsf{finite}_e)) \vee (\mathsf{test}(\mathsf{true}) \vee (\mathsf{skip}\,;\neg E))$$

Simplifying, definition step( )*:*

$$\mathsf{test}(\mathsf{true}) \vee (\mathsf{step}(\neg T)\,;\mathsf{finite}_e) \vee (\mathsf{step}(\mathsf{true})\,;\neg E)$$

***(c)**:*

$$\neg((\mathsf{test}(W_0)\,;E_0) \vee (\mathsf{test}(W_1)\,;E_1))$$

Using De Morgan:

$$\neg((\mathsf{test}(W_0)\,;E_0) \wedge \neg(\mathsf{test}(W_1)\,;E_1))$$

*Using **(a)** twice:*

$$((\mathsf{test}(\neg W_0)\,;\mathsf{finite}_e) \vee \neg E_0) \wedge ((\mathsf{test}(\neg W_1)\,;\mathsf{finite}_e) \vee \neg E_1)$$

Simple rewriting:

$$\begin{array}{l}((\mathsf{test}(\neg W_0)\,;\mathsf{finite}_e) \wedge (\mathsf{test}(\neg W_1)\,;\mathsf{finite}_e)) \vee \\ ((\mathsf{test}(\neg W_0)\,;\mathsf{finite}_e) \wedge \neg E_1) \vee \\ ((\mathsf{test}(\neg W_1)\,;\mathsf{finite}_e) \wedge \neg E_0) \vee \\ (\neg E_0 \wedge \neg E_1)\end{array}$$

Using ITL reasoning, test reasoning twice and De Morgan:

$$\begin{array}{l}(\mathsf{test}(\neg W_0 \wedge \neg W_1)\,;\mathsf{finite}_e) \vee \\ (\mathsf{test}(\neg W_0)\,;(\neg E_1)) \vee \\ (\mathsf{test}(\neg W_1)\,;(\neg E_0)) \vee \\ \neg(E_0 \vee E_1)\end{array}$$

***(d)**:*

$$\neg((\mathsf{test}(W)\,;E_0) \vee (\mathsf{step}(T)\,;E_1))$$

Using De Morgan:

$$\neg(\mathsf{test}(W)\,;E_0) \wedge \neg(\mathsf{step}(T)\,;E_1)$$

*Using **(a)** and **(b)**:*

$$\begin{array}{l}((\mathsf{test}(\neg W)\,;\mathsf{finite}_e) \vee \neg E_0) \wedge \\ ((\mathsf{step}(\neg T)\,;\mathsf{finite}_e) \vee \mathsf{test}(\mathsf{true}) \vee (\mathsf{step}(\mathsf{true})\,;\neg E_1))\end{array}$$

Using simple rewriting:

$$\begin{array}{l}((\mathsf{test}(\neg W)\,;\mathsf{finite}_e) \wedge (\mathsf{step}(\neg T)\,;\mathsf{finite}_e)) \vee \\ ((\mathsf{test}(\neg W)\,;\mathsf{finite}_e) \wedge \mathsf{test}(\mathsf{true})) \vee \\ ((\mathsf{test}(\neg W)\,;\mathsf{finite}_e) \wedge (\mathsf{step}(\mathsf{true})\,;\neg E_1)) \vee \\ ((\neg E_0) \wedge ((\mathsf{step}(\neg T)\,;\mathsf{finite}_e) \vee \mathsf{test}(\mathsf{true}) \vee (\mathsf{step}(\mathsf{true})\,;\neg E_1)))\end{array}$$

Using ITL reasoning:

$$\begin{array}{l}((\mathsf{test}(\neg W)\,;\mathsf{step}(\neg T)\,;\mathsf{finite}_e)) \vee \\ (\mathsf{test}(\neg W)) \vee \\ (\mathsf{test}(\neg W)\,;\mathsf{step}(\mathsf{true})\,;(\neg E_1)) \vee \\ ((\neg E_0) \wedge ((\mathsf{step}(\neg T)\,;\mathsf{finite}_e) \vee \mathsf{test}(\mathsf{true}) \vee (\mathsf{step}(\mathsf{true})\,;\neg E_1)))\end{array}$$

*Using **(b)** reverse:*

$$\begin{array}{l}((\mathsf{test}(\neg W)\,;\mathsf{step}(\neg T)\,;\mathsf{finite}_e)) \vee \\ (\mathsf{test}(\neg W)) \vee \\ (\mathsf{test}(\neg W)\,;\mathsf{step}(\mathsf{true})\,;(\neg E_1)) \vee \\ ((\neg E_0) \wedge \neg(\mathsf{step}(T)\,;E_1))\end{array}$$

Using De Morgan:

$$\begin{array}{l}((\mathsf{test}(\neg W)\,;\mathsf{step}(\neg T)\,;\mathsf{finite}_e)) \vee \\ (\mathsf{test}(\neg W)) \vee \\ (\mathsf{test}(\neg W)\,;\mathsf{step}(\mathsf{true})\,;(\neg E_1)) \vee \\ \neg(E_0 \vee (\mathsf{step}(T)\,;E_1))\end{array}$$

***(e)**:*

$$\neg(\mathsf{test}(\mathsf{true}) \vee (\mathsf{step}(T)\,;E_1))$$

Using De Morgan:

$$\neg(\mathsf{test}(\mathsf{true})) \wedge \neg(\mathsf{step}(T)\,;E_1)$$

Using ITL reasoning:

$$(\mathsf{step}(\mathsf{true})\,;\mathsf{finite}_e) \wedge \neg(\mathsf{step}(T)\,;E_1)$$

*Using **(b)**:*

$$\begin{array}{l}(\mathsf{step}(\mathsf{true})\,;\mathsf{finite}_e) \wedge \\ ((\mathsf{step}(\neg T)\,;\mathsf{finite}_e) \vee \mathsf{test}(\mathsf{true}) \vee (\mathsf{step}(\mathsf{true})\,;\neg E_1))\end{array}$$

Using simple rewriting:

$$\begin{array}{l}((\mathsf{step}(\mathsf{true})\,;\mathsf{finite}_e) \wedge (\mathsf{step}(\neg T)\,;\mathsf{finite}_e)) \vee \\ ((\mathsf{step}(\mathsf{true})\,;\mathsf{finite}_e) \wedge \mathsf{test}(\mathsf{true})) \vee \\ ((\mathsf{step}(\mathsf{true})\,;\mathsf{finite}_e) \wedge (\mathsf{step}(\mathsf{true})\,;\neg E_1))\end{array}$$

Using ITL reasoning:

$$\begin{array}{l}(\mathsf{step}(\neg T)\,;\mathsf{finite}_e) \vee \\ (\mathsf{step}(\mathsf{true})\,;\neg E_1)\end{array}$$

***(f)**:*

$$\neg((\mathsf{step}(T_0)\,;E_0) \vee (\mathsf{step}(T_1)\,;E_1))$$

Using De Morgan:

$$\neg(\mathsf{step}(T_0)\,;E_0) \wedge \neg(\mathsf{step}(T_1)\,;E_1)$$

*Using **(b)**:*

$$\begin{array}{l}((\mathsf{step}(\neg T_0)\,;\mathsf{finite}_e) \vee \mathsf{test}(\mathsf{true}) \vee (\mathsf{step}(\mathsf{true})\,;\neg E_0)) \wedge \\ ((\mathsf{step}(\neg T_1)\,;\mathsf{finite}_e) \vee \mathsf{test}(\mathsf{true}) \vee (\mathsf{step}(\mathsf{true})\,;\neg E_1))\end{array}$$

Using simple rewriting:

$$
\begin{array}{l}
((\mathsf{step}(\neg T_0)\,;\mathsf{finite}_e) \wedge (\mathsf{step}(\neg T_1)\,;\mathsf{finite}_e)) \vee \\
((\mathsf{step}(\neg T_0)\,;\mathsf{finite}_e) \wedge \mathsf{test}(\mathsf{true})) \vee \\
((\mathsf{step}(\neg T_0)\,;\mathsf{finite}_e) \wedge (\mathsf{step}(\mathsf{true})\,;\neg E_1)) \vee \\
(\mathsf{test}(\mathsf{true}) \wedge (\mathsf{step}(\neg T_1)\,;\mathsf{finite}_e)) \vee \\
(\mathsf{test}(\mathsf{true}) \wedge \mathsf{test}(\mathsf{true})) \vee \\
(\mathsf{test}(\mathsf{true}) \wedge (\mathsf{step}(\mathsf{true})\,;\neg E_1)) \vee \\
((\mathsf{step}(\mathsf{true})\,;\neg E_0) \wedge (\mathsf{step}(\neg T_1)\,;\mathsf{finite}_e)) \vee \\
((\mathsf{step}(\mathsf{true})\,;\neg E_0) \wedge \mathsf{test}(\mathsf{true})) \vee \\
((\mathsf{step}(\mathsf{true})\,;\neg E_0) \wedge (\mathsf{step}(\mathsf{true})\,;\neg E_1))
\end{array}
$$

Using ITL reasoning and De Morgan:

$$
\begin{array}{l}
(\mathsf{step}(\neg T_0 \wedge \neg T_1)\,;\mathsf{finite}_e) \vee \\
(\mathsf{step}(\neg T_0)\,;\neg E_1) \vee \\
(\mathsf{test}(\mathsf{true})) \vee \\
(\mathsf{step}(\neg T_1)\,;\neg E_0) \vee \\
(\mathsf{step}(\mathsf{true})\,;\neg(E_0 \vee E_1))
\end{array}
$$

Lemma 3

$$
\begin{array}{ll}
a & \neg(E\,;\mathsf{test}(W)) \equiv (\mathsf{finite}_e\,;\mathsf{test}(\neg W)) \vee \neg E \\
b & \neg(E\,;\mathsf{step}(T)) \equiv (\mathsf{finite}_e\,;\mathsf{step}(\neg T)) \vee \mathsf{test}(\mathsf{true}) \vee ((\neg E)\,;\mathsf{step}(\mathsf{true})) \\
c & \neg((E_0\,;\mathsf{test}(W_0)) \vee (E_1\,;\mathsf{test}(W_1))) \equiv (\mathsf{finite}_e\,;\mathsf{test}(\neg W_0 \wedge \neg W_1)) \vee \\
 & \quad ((\neg E_1)\,;\mathsf{test}(\neg W_0)) \vee ((\neg E_0)\,;\mathsf{test}(\neg W_1)) \vee \neg(E_0 \vee E_1) \\
d & \neg((E_0\,;\mathsf{test}(W)) \vee (E_1\,;\mathsf{step}(T))) \equiv \\
 & (\mathsf{finite}_e\,;\mathsf{step}(\neg T)\,;\mathsf{test}(W)) \vee ((\neg E_1)\,;\mathsf{step}(\mathsf{true})\,;\mathsf{test}(\neg W)) \vee \\
 & \mathsf{test}(\neg W) \vee \neg(E_0 \vee (E_1\,;\mathsf{step}(T))) \\
e & \neg(\mathsf{test}(\mathsf{true}) \vee (E_1\,;\mathsf{step}(T))) \equiv \\
 & (\mathsf{finite}_e\,;\mathsf{step}(\neg T)) \vee ((\neg E_1)\,;\mathsf{step}(\mathsf{true})) \\
f & \neg((E_0\,;\mathsf{step}(T_0)) \vee (E_1\,;\mathsf{step}(T_1))) \equiv \\
 & \mathsf{test}(\mathsf{true}) \vee (\mathsf{finite}_e\,;\mathsf{step}(\neg T_0 \wedge \neg T_1)) \vee \\
 & ((\neg E_1)\,;\mathsf{step}(\neg T_0)) \vee ((\neg E_0)\,;\mathsf{step}(\neg T_1)) \vee \\
 & (\neg(E_0 \vee E_1)\,;\mathsf{step}(\mathsf{true}))
\end{array}
$$

References

[1] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification.* New York: Springer, 1992.

[2] D. Harel, D. Kozen, and J. Tiuryn, *Dynamic Logic.* Cambridge, MA: MIT Press, 2000.

[3] B. Moszkowski, "A hierarchical analysis of propositional temporal logic based on intervals," in *We Will Show Them: Essays in Honour of Dov Gabbay*, S. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, Eds. King's College, London: College Publications (formerly KCL Publications), 2005, vol. 2, pp. 371–440.

[4] F. Siewe, "A Compositional Framework for the Development of Secure Access Control Systems," Ph.D. dissertation, Software Technology Research Laboratory, Department of Computer Science and Engineering, De Montfort University, Leicester, 2005.