Automated Reasoning

Lecture 14: Linear Temporal Logic II

Jacques Fleuriot
jdf@inf.ed.ac.uk

Overview

- LTL in Isabelle
- Linear Temporal Logic over finite traces: LTL_f
- A soft semantics for LTL_f

Recall: Specifications

We are interested in specifying behaviours of systems over time.

- Use **Temporal Logic**

Specifications are built from:

1. Primitive properties of individual states
   e.g., "is on", "is off", "is active", "is in region", "is at position";
2. propositional connectives $\wedge, \vee, \neg, \rightarrow$;
3. and **temporal** connectives: *e.g.,*
   - At **all times**, the system is not simultaneously *reading* and *writing*.
   - If a *request* signal is asserted **at some time**, a corresponding *grant* signal will be asserted **within 10 time units**.
   - The robot's *position* will **eventually** be at **1 distance unit** from the shelf.

Linear Temporal Logic

- We introduced a commonly used syntax for LTL in the last lecture but in this lecture will use a variant.
  - This is inspired in part by an existing formalisation in the AFP.
  - Easier to provide a "soft" semantics (for our discussion of LTL over finite traces).

Our syntax for LTL formulas ϕ:

$$\phi ::= \mathit{true} \mid \mathit{false} \mid p \mid \neg p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathbf{X}\phi \mid \mathbf{F}\phi \mid \mathbf{G}\phi \mid \phi\mathbf{W}\phi \mid \phi\mathbf{M}\phi$$

Note: We have an explicitly negated atom so a literal is an atom or the negation of an atom.

Temporal operator:

- **X** — Next
- **F** — Future; Eventually
- **G** — Globally; Always
- **W** —Weak Until
- **M** —Strong Release

LTL – Informal Semantics for W and M

- Recall: The semantics of $\phi_1 \, \mathbf{W} \, \phi_2$ does not require a state to be reached for which ϕ_2 holds, unlike $\phi_1 \, \mathbf{U} \, \phi_2$. Thus, we have:

$$\phi_1 \, \mathbf{W} \, \phi_2 \equiv \phi_1 \, \mathbf{U} \, \phi_2 \vee \mathbf{G} \, \phi_1$$

- $\phi \mathbf{M} \psi$ holds if there is a future position where both ϕ becomes true, and ψ holds for all positions prior to and including that i.e. ϕ '(strongly) releases' ψ.
  - It is equivalent to $\neg(\neg\phi \mathbf{W} \neg\psi)$.
  - Thus $\mathbf{M}$ is the dual of $\mathbf{W}$.

Note: See previous lecture for the informal meaning of the other operators.

LTL in Isabelle: Syntax

```
datatype (atoms_ltl: 'a) ltl =
    True_ltl                                     ("true")
  | False_ltl                                    ("false")
  | Prop_ltl 'a                                  ("prop'(_')")
  | NotProp_ltl 'a                               ("nprop'(_')")
  | And_ltl "'a ltl" "'a ltl"                    ("_ and _" [82,82] 81)
  | Or_ltl "'a ltl" "'a ltl"                     ("_ or _" [81,81] 80)
  | Next_ltl "'a ltl"                            ("X _" [88] 87)
  | Eventually_ltl "'a ltl"                      ("F _" [88] 87)
  | Always_ltl "'a ltl"                          ("G _" [88] 87)
  | WUntil_ltl "'a ltl" "'a ltl"                 ("_ W _" [84,84] 83)
  | SRelease_ltl "'a ltl" "'a ltl"               ("_ M _" [84,84] 83)
```

- Deep embedding: the syntax is defined as an explicit datatype in Isabelle/HOL.
- Note the use of the type variable "`'a`", thus e.g. `prop(q)` stands for atom q while `nprop(q)` stands for $\neg q$.
- Negation is not taken as primitive (and thus has no constructor).

LTL in Isabelle: Not operation

```
fun Not_ltl :: "'a ltl ⇒ 'a ltl" ("not _" [85] 85)
where
  "not (true) = false"
| "not (false) = true"
| "not (prop(q)) = nprop(q)"
| "not (nprop(q)) = prop(q)"
| "not (φ and ψ) = (not φ) or (not ψ)"
| "not (φ or ψ) = (not φ) and (not ψ)"
| "not (X φ) = X (not φ)"
| "not (F φ) = G (not φ)"
| "not (G φ) = F (not φ)"
| "not (φ W ψ) = (not φ) M (not ψ)"
| "not (φ M ψ) = (not φ) W (not ψ)"
```

- (Some of the) Dualities given in the previous lecture are now used to define negation.
- Negation is defined using primitive recursion (we could have used `primrec` instead of `fun`).
- We can easily prove by induction that $\texttt{not}(\texttt{not}\,\phi) = \phi$.

LTL Semantics in Isabelle: Satisfaction by Path

Satisfaction: $\pi \models^i \phi$ – "path at position i satisfies formula ϕ"

```
primrec position_semantics_ltl :: "['a set word, nat, 'a ltl] ⇒ bool" ("_ ⊫_ _" [80,80,80] 80)
where
  "π ⊫i true = True"
| "π ⊫i false = False"
| "π ⊫i prop(q) = (q ∈ π i)"
| "π ⊫i nprop(q) = (q ∉ π i)"
| "π ⊫i φ and ψ = (π ⊫i φ ∧ π ⊫i ψ)"
| "π ⊫i φ or ψ = (π ⊫i φ ∨ π ⊫i ψ)"
| "π ⊫i X φ = (∃j. j=i+1 ∧ π ⊫j φ)"
| "π ⊫i F φ = (∃j≥i. π ⊫j φ)"
| "π ⊫i G φ = (∀j≥i. π ⊫j φ)"
| "π ⊫i φ W ψ = ((∀j≥i. π ⊫j φ) ∨ (∃j≥i. π ⊫j ψ ∧ (∀k. i ≤ k ∧ k < j ⟶  π ⊫k φ)))"
| "π ⊫i φ M ψ = (∃j≥i. π ⊫j φ ∧ (∀k. i ≤ k ∧ k ≤ j ⟶ π ⊫k ψ))"
```

- LTL semantics is defined as a primitive recursive function (relation) over our datatype.
- `type_synonym 'a word = nat ⇒ 'a`
- π `:: nat ⇒ 'a set`, i.e. $\pi(i)$ is the set of atoms that is true (in state) at position i of path π.

LTL Semantics in Isabelle: Satisfaction by Path

- Alternative semantics for temporal operators are easily derived as theorems:

```
lemma neXt: "π ⊫i X φ = π ⊫(i+1) φ"
lemma eventually: "π ⊫i F φ = (∃j. π ⊫(i + j) φ)"
lemma always: "π ⊫i G φ = (∀j. π ⊫(i + j) φ)"
lemma weak:
  "π ⊫i φ W ψ = ((∀j. π ⊫(i+j) φ) ∨
                  (∃j. π ⊫(i+j) ψ ∧ (∀k<j. π ⊫(i+k) φ)))"
lemma strong_release:
  "π ⊫i φ M ψ = (∃j. π ⊫(i+j) φ ∧ (∀k ≤ j. π ⊫(i+k) ψ))"
```

LTL Equivalences

- The expected equivalences follow:

```
lemma "π ⊫i G (φ and ψ) = (π ⊫i G φ and G ψ)"
lemma "π ⊫i F (φ or ψ) = (π ⊫i F φ or F ψ)"
lemma "π ⊫i σ W (φ or ψ) = (π ⊫i (σ W φ) or (σ W ψ))"
lemma "π ⊫i (φ and ψ) W σ = (π ⊫i (φ W σ) and (ψ W σ))"
lemma "π ⊫i G (F (G φ)) = π ⊫i F (G φ)"
lemma "π ⊫i F (G (F φ)) = π ⊫i G (F φ)"
lemma "π ⊫i G (F φ or F ψ) = π ⊫i G (F φ) or G (F ψ)"
```

LTL over Finite Traces

- Many real-word problems involve finite traces or paths rather than infinite ones, especially when dealing with terminating processes.
  - Example: trajectory constraints in AI planning and, in our research, constraining the finite trajectory being learnt by a neural network by asking it to e.g. avoid particular regions or reach specific points.
- LTL_f is defined on finite traces.
  - It uses the same syntax as LTL.
  - Some changes needed to our understanding of the operators e.g. what is the meaning of $\mathbf{X}\bot$ at the last position?
- How can we formalise finite paths/traces in Isabelle/HOL?

LTL_f Semantics in Isabelle: "Computational Definition"

- We can formalise a semantics recursively over paths, which are represented as lists.

```
function semantics_ltl_f :: "['a set list, 'a ltl] ⇒ bool" ("_ ⊨ _" [80,80] 80)
  where
  "[] ⊨ φ = False"
| "(s₀#π) ⊨ true = True"
| "(s₀#π) ⊨ false = False"
| "(s₀#π) ⊨ prop(q) = (q ∈ s₀)"
| "(s₀#π) ⊨ nprop(q) = (q ∉ s₀)"
| "(s₀#π) ⊨ φ and ψ = ((s₀#π) ⊨ φ ∧ (s₀#π) ⊨ ψ)"
| "(s₀#π) ⊨ φ or ψ = ((s₀#π) ⊨ φ ∨ (s₀#π) ⊨ ψ)"
| "(s₀#π) ⊨ X φ = π ⊨ φ"
| "(s₀#π) ⊨ F φ = ((s₀#π) ⊨ φ ∨ π ⊨ F φ)"
| "(s₀#π) ⊨ G φ = ((s₀#π) ⊨ φ ∧ (if π = [] then True else π ⊨ G φ))"
| "(s₀#π) ⊨ φ W ψ = ((((s₀#π) ⊨ φ) ∧ (if π = [] then True else π ⊨ φ W ψ)) ∨ (s₀#π) ⊨ ψ)"
| "(s₀#π) ⊨ φ M ψ = (((s₀#π) ⊨ φ and ψ) ∨ (if π = [] then False else (s₀#π) ⊨ ψ ∧ π ⊨ φ M ψ))"
by pat_completeness auto
termination by size_change
```

- This gives a more *computational* way of specifying the semantics, which can then be extracted faithfully as code (e.g. Ocaml or Haskell) and executed!
- We prove that this semantics is equivalently expressed in terms path suffixes (next slide).

LTL_f Semantics in Isabelle: *i*th Suffix of Path

- Semantics on terms of *i*th suffix of the path (see previous lecture) also uses a list but requires other list operations.

```
function semantics_ltl_f :: "['a set list, 'a ltl] ⇒ bool" ("_ ⊨ _" [80,80] 80)
  where
  "[] ⊨ φ = False"
| "(s₀#π) ⊨ true = True"
| "(s₀#π) ⊨ false = False"
| "(s₀#π) ⊨ prop(q) = (q ∈ s₀)"
| "(s₀#π) ⊨ nprop(q) = (q ∉ s₀)"
| "(s₀#π) ⊨ φ and ψ = ((s₀#π) ⊨ φ ∧ (s₀#π) ⊨ ψ)"
| "(s₀#π) ⊨ φ or ψ = ((s₀#π) ⊨ φ ∨ (s₀#π) ⊨ ψ)"
| "(s₀#π) ⊨ X φ = π ⊨ φ"
| "(s₀#π) ⊨ F φ = (∃j < length (s₀#π). drop j (s₀#π) ⊨ φ)"
| "(s₀#π) ⊨ G φ = (∀j < length (s₀#π). drop j (s₀#π) ⊨ φ)"
| "(s₀#π) ⊨ φ W ψ = (∀j < length (s₀#π). drop j (s₀#π) ⊨ φ ∨ (∃k ≤ j. drop k (s₀#π) ⊨ ψ))"
| "(s₀#π) ⊨ φ M ψ = (∃j < length (s₀#π). drop j (s₀#π) ⊨ φ ∧ (∀k ≤ j. drop k (s₀#π) ⊨ ψ))"
by pat_completeness auto
termination by size_change
```

  - The function `drop n xs` drops the first *n* elements of list *xs*.

- How do we deal with the end of the path? More generally, does $\neg \mathbf{X}\phi \equiv \mathbf{X}\,\neg\phi$ still hold?

LTL$_f$ Negation in Isabelle

Introduce a Weak Next operator:

```
definition WeakNext_ltl :: "'a ltl ⇒ 'a ltl" ("Xw _" [88] 87)
  where "WeakNext_ltl φ ≡ not X (not φ)"
```

that will enable us to define negation:

```
fun Notf_ltl :: "'a ltl ⇒ 'a ltl" ("notf _" [85] 85)
where
  "notf (true) = false"
| "notf (false) = true"
| "notf (prop(q)) = nprop(q)"
| "notf (nprop(q)) = prop(q)"
| "notf (φ and ψ) = (notf φ) or (notf ψ)"
| "notf (φ or ψ) = (notf φ) and (notf ψ)"
| "notf (X φ) = Xw (not φ)"
| "notf (F φ) = G (notf φ)"
| "notf (G φ) = F (notf φ)"
| "notf (φ W ψ) = (notf φ) M (notf ψ)"
| "notf (φ M ψ) = (notf φ) W (notf ψ)"
```

- It then follows that $\mathtt{not_f}(\mathtt{not_f}(\phi)) = \phi$.
- Most of the usual LTL equivalences are recovered.

From Discrete to RobustSemantics

- Recall: We are interested in specifying behaviours of systems over time.
- Informally, can we find a way of specifying such behaviours rigorously and then incorporating them into the training of a neural network to ensure it (attempts) to satisfy them?
  - Issue: The semantics of LTL and LTL_f are boolean so either a formula is satisfied or it is not. So, they are not robust (and "learning"-friendly).
  - Can we formulate a soft (robust) semantics that will approximate the standard one and thus enable us to quantify by how much a LTL_f constraint is violated during learning i.e. figure out the constraint-related *loss*?
  - Moreover, we would like this function to be differentiable so that we can use it to minimise the constraint loss during back propagation (cf. Mark's Guest Lecture).

Some Soft Functions

- We formalise soft versions of the functions max and minimum:

```
fun Max_gamma :: "real ⇒ real ⇒ real ⇒ real"  ("_ ⊓_ _" [82,82] 81) where
 "(a ⊓γ b) = (if γ ≤ 0 then max a b
              else γ * ln (exp (a / γ) + exp (b / γ)))"

fun Min_gamma :: "real ⇒ real ⇒ real ⇒ real"  ("_ ⊔_ _" [81,81] 80) where
 "(a ⊔γ b) = (if γ ≤ 0 then min a b
              else -γ * ln (exp (-a / γ) + exp (-b / γ)))"
```

- Each of these soft functions takes an additional smoothing parameter γ.
- We show that as $\gamma \to 0$, $\sqcap_\gamma \to$ max, $\sqcup_\gamma \to$ min, and that they are differentiable for $\gamma > 0$.

A Simple Soft LTL_f Semantics in Isabelle

- This is a simple formalisation but it illustrates how we can move from a boolean to a real-valued semantics:

```
function soft_semantics_ltl_f :: "['a set list, real, 'a ltl] ⇒ real" ("_ ⊨s_ _" [80,80,80] 80)
where
   "[] ⊨sγ φ = 1"
| "(s₀#π) ⊨sγ true = 0"
| "(s₀#π) ⊨sγ false = 1"
| "(s₀#π) ⊨sγ prop(q) = (if (q ∈ s₀) then 0 else 1)"
| "(s₀#π) ⊨sγ nprop(q) = (if (q ∉ s₀) then 0 else 1)"
| "(s₀#π) ⊨sγ φ and ψ = ((s₀#π) ⊨sγ φ) ⨅γ ((s₀#π) ⊨sγ ψ)"
| "(s₀#π) ⊨sγ φ or ψ =  ((s₀#π) ⊨sγ φ) ⨆γ ((s₀#π) ⊨sγ ψ)"
| "(s₀#π) ⊨sγ X φ = π ⊨sγ φ"
| "(s₀#π) ⊨sγ F φ =  ((s₀#π) ⊨sγ φ) ⨆γ (π ⊨sγ F φ)"
| "(s₀#π) ⊨sγ G φ = ((s₀#π) ⊨sγ φ) ⨅γ (if π = [] then 0 else π ⊨sγ G φ)"
| "(s₀#π) ⊨sγ φ W ψ =
     ((((s₀#π) ⊨sγ φ) ⨅γ (if π = [] then 0 else π ⊨sγ φ W ψ)) ⨆γ ((s₀#π) ⊨sγ ψ))"
| "(s₀#π) ⊨sγ φ M ψ =
     ((s₀#π) ⊨sγ φ and ψ) ⨆γ (if π = [] then 1 else ((s₀#π) ⊨sγ ψ) ⨅γ (π ⊨sγ φ M ψ))"
by pat_completeness auto
termination by size_change
```

- We have a literal translation of the standard LTL_f semantics we presented earlier, with $\wedge$ and $\vee$ replaced by our softmax $\sqcap_\gamma$ and softmin $\sqcup_\gamma$ functions respectively.

Is our Soft Semantics Sound?

Yes! We can formally prove:

- with $\gamma = 0$, it is exactly the standard boolean semantics:

```
lemma "(π ⊨s0 φ = 0) = (π ⊨ φ)"
```

- but more importantly, we have:

```
lemma "((λγ. π ⊨sγ φ) —0→ 0) = (π ⊨ φ)"
```

  where, mathematically, the theorem states:

$$\lim_{\gamma \to 0} (\pi \models_s \gamma\ \phi) = 0 \Longleftrightarrow \pi \models \phi$$

- We thus have a well-defined semantic function
  - In principle it can be used to implement the loss with respect to LTL constraints during learning by a neural network.
  - It will be rather limited and excruciatingly slow though; so a much more sophisticated formalisation is needed for practical use.
- Next Lecture: An overview of how this can be adapted and used for rigorous, constrained neural learning.

Summary

- Linear Temporal Logic (H&R 3.2)
  - Syntax and Semantics: see also https://www.isa-afp.org/sessions/ltl.
- Linear Temporal Logic over finite traces: LTL_f
  - See, for example: *Linear Temporal Logic and Linear Dynamic Logic on Finite Traces* by De Giacomo and Vardi, https://www.cs.rice.edu/~vardi/papers/ijcai13.pdf
- A soft semantics for LTL_f
  - See our paper: *Constrained Training of Neural Networks via Theorem Proving*, https://ceur-ws.org/Vol-3311/paper2.pdf and https://arxiv.org/pdf/2207.03880
  - See also: *Elaborating on Learned Demonstrations with Temporal Logic Specifications* by Innes and Ramamoorhy, https://arxiv.org/pdf/2002.00784