DejaVu: Checking Event Streams against Temporal Logic Formulas

Ruxandra Icleanu, Ruturaj Nanoti

CIS 6730: Computer-aided Verification
University of Pennsylvania

April 2023

What is DejaVu?

- DejaVu is a tool used to verify whether event streams (traces) satisfy specifications.

What is DejaVu?

- DejaVu is a tool used to verify whether event streams (traces) satisfy specifications.
- Developed in *Scala* — a high level programming language.

What is DejaVu?

- DejaVu is a tool used to verify whether event streams (traces) satisfy specifications.
- Developed in *Scala* — a high level programming language.
- Specifications are expressed in a first-order past-time temporal linear logic

What is DejaVu?

- DejaVu is a tool used to verify whether event streams (traces) satisfy specifications.
- Developed in *Scala* — a high level programming language.
- Specifications are expressed in a first-order past-time temporal linear logic
- It also supports recursive rules & macros

What is DejaVu?

- DejaVu is a tool used to verify whether event streams (traces) satisfy specifications.
- Developed in *Scala* — a high level programming language.
- Specifications are expressed in a first-order past-time temporal linear logic
- It also supports recursive rules & macros
- The tool is generally used for distributed systems and data processing

Overview

- Past-time first-order LTL
- DejaVu's syntax
- How it Works
- Examples

First-order past-time LTL

- recall: functional vs reactive systems

First-order past-time LTL

- recall: functional vs reactive systems
- Temporal logic [Pneuli] - useful framework to reason about reactive systems

First-order past-time LTL

- recall: functional vs reactive systems
- Temporal logic [Pneuli] - useful framework to reason about reactive systems
- TL: we can describe properties using atomic propositions that holds at some specific points in time, modalities and boolean connectives

First-order past-time LTL

- recall: functional vs reactive systems
- Temporal logic [Pneuli] - useful framework to reason about reactive systems
- TL: we can describe properties using atomic propositions that holds at some specific points in time, modalities and boolean connectives
- choice of particular modalities + their truth tables $\Rightarrow$ different temporal logics

First-order past-time LTL

- recall: functional vs reactive systems
- Temporal logic [Pneuli] - useful framework to reason about reactive systems
- TL: we can describe properties using atomic propositions that holds at some specific points in time, modalities and boolean connectives
- choice of particular modalities + their truth tables $\Rightarrow$ different temporal logics
- *Claim*: LTL satisfiability is P-complete (w.r.t. formula length)

First-order past-time LTL Cont.

Here, choose

- Boolean connectives
- existential quantifier
- modalities: Previously and Since

$\Rightarrow$ past-time LTL

FOPLTL it can be defined by the following grammar:

$$\phi = T \mid \neg\phi \mid \ \phi_1 \vee \phi_2 \mid \phi_1 \ S \ \phi_2 \mid @\ \phi \mid \exists x.\phi \mid (x = \lambda)$$

Other modalities can be derived:

- Once $\quad O\ \phi := TS\phi$
- Historically $\quad H\ \phi := \neg(O(\neg\phi))$

$(s, i, \gamma) \vdash \phi$: given that γ is an assignment over all free variables in ϕ, the sequence s of events satisfies ϕ at position i.

First-order past-time LTL Cont.

Aside: Future-time LTL vs Past-time LTL

- Recall: Future-time LTL had the modalities Always, Eventually, Next, Until
- Gabbay[1980]: past-time modalities do not expressive power to pure future LTL (derived an an for translating formulas with past-time modalities into equivalent pure future ones)
- Markey: But past-time LTL is more succint
  There exists a family of PLTL formulas ϕ_n *with size* $O(n)$ *such that the equivalent equivalent LTL formulas have size* $\Omega(2^n)$.

Syntax

```
true, false
id(v1,...,vn)    : event or call of predicate macro, where vi can be a constant or variable
p -> q           : p implies q
p | q            : p or q
p & q            : p and q
! p              : not p
p S q            : p since q (q was true in the past, and since then, including that point in time
p S[<=d] q       : p since q but where q occurred within d time units
p S[>d] q        : p since q but where q occurred earlier than d time units
p Z[<=d] q       : p since q but where q did not occur at the current time
[p,q)            : interval notation equivalent to: !q S p
@ p              : in previous state p is true
P p              : in some previous state p is true
P[<=d] p         : in some previous state within d time units p is true
P[>d] p          : in some previous state earlier than d time units p is true
H p              : in all previous states p is true
H[<=d] p         : in all previous states within d time units p is true
H[>d] p          : in all previous states earlier than d time units p is true
x op k           : x is related to variable or constant k via op. E.g.: x < 10,
                   x <= y, x = y, x >= 10, x > z
// -- quantification over seen values in the past, see (*) below:
exists x . p(x) : there exists an x such that seen(x) and p(x)
forall x . p(x) : for all x, if seen(x) then p(x)
// -- quantification over the infinite domain of all values:
Exists x . p(x) : there exists an x such that p(x)
Forall x . p(x) : for all x p(x)
(*) seen(x) holds if x has been observed in the past
```

How does it work?

- DejaVu uses a program ("translate") that, when a given FO-PLTL property, generates a monitor.

How does it work?

- DejaVu uses a program ("translate") that, when a given FO-PLTL property, generates a monitor.
- A parser parses the specification and produces an abstract syntax tree, which is then traversed and translated to create this monitor.

How does it work?

- DejaVu uses a program ("translate") that, when a given FO-PLTL property, generates a monitor.
- A parser parses the specification and produces an abstract syntax tree, which is then traversed and translated to create this monitor.
- This monitor takes as input a trace (i.e. a sequence of events), and returns a boolean for each position in the trace.

How does it work?

- DejaVu uses a program ("translate") that, when a given FO-PLTL property, generates a monitor.
- A parser parses the specification and produces an abstract syntax tree, which is then traversed and translated to create this monitor.
- This monitor takes as input a trace (i.e. a sequence of events), and returns a boolean for each position in the trace.
- The tool utilizes BDDs (**B**inary **D**ecision **D**iagrams) for assigning variables to quanitfiable entities.

Examples

Let's see how we can:

- write temporal logic specifications in practice

Examples

Let's see how we can:

- write temporal logic specifications in practice
- verify if event streams satisfy our specifications

Examples

Let's see how we can:

- write temporal logic specifications in practice
- verify if event streams satisfy our specifications
- reason about time

Examples

Let's see how we can:

- write temporal logic specifications in practice
- verify if event streams satisfy our specifications
- reason about time
- use recursive rules

1. Access

We want to specify a simple condition for allowing users to access files.

1. Access

We want to specify a simple condition for allowing users to access files. In order to be able to access the file:

- the user must have not logged out since they logged in
  $\Leftrightarrow$ `[login(user),logout(user))`

1. Access

We want to specify a simple condition for allowing users to access files. In order to be able to access the file:

- the user must have not logged out since they logged in
  $\Leftrightarrow$ `[login(user),logout(user))`
- the file must have not been closed since it was opened
  $\Leftrightarrow$ `[open(file),close(file))`

2. Locks

First, we want to formalise some useful properties in the context of threads acquiring locks:

- a thread that is going to sleep should not hold any locks
  $\Leftrightarrow$ `sleep(t) -> ![acq(t,l),rel(t,l))`

2. Locks

First, we want to formalise some useful properties in the context of threads acquiring locks:

- a thread that is going to sleep should not hold any locks
  $\Leftrightarrow$ `sleep(t) -> ![acq(t,l),rel(t,l))`
- a lock can be acquired by at most one thread at a time
  $\Leftrightarrow$ `acq(t,l) -> ! exists s . @ [acq(s,l),rel(s,l))`

2. Locks

First, we want to formalise some useful properties in the context of threads acquiring locks:

- a thread that is going to sleep should not hold any locks
  $\Leftrightarrow$ `sleep(t) -> ![acq(t,l),rel(t,l))`
- a lock can be acquired by at most one thread at a time
  $\Leftrightarrow$ `acq(t,l) -> ! exists s . @ [acq(s,l),rel(s,l))`
- a thread can only release a lock it has acquired (and not already released)
  $\Leftrightarrow$ `(rel(t,l) -> @ [acq(t,l),rel(t,l)))`

2. Locks Cont.

Dining philosopher problem
- N philosophers dine together at a circular table
- one fork between every two philosophers
- each philosopher can either think or eat at a time, and they can only eat if they have both the left and right fork

2. Locks Cont.

Dining philosopher problem
- N philosophers dine together at a circular table
- one fork between every two philosophers
- each philosopher can either think or eat at a time, and they can only eat if they have both the left and right fork

We want to design a protocol such that no philosopher will starve (i.e. each philosopher can alternate forever between thinking and eating) assuming that no philosopher knows when the others want to eat or think.

2. Locks Cont.

Dining philosopher problem
- N philosophers dine together at a circular table
- one fork between every two philosophers
- each philosopher can either think or eat at a time, and they can only eat if they have both the left and right fork

We want to design a protocol such that no philosopher will starve (i.e. each philosopher can alternate forever between thinking and eating) assuming that no philosopher knows when the others want to eat or think.

Observe: taking the forks in a cyclic manner cannot be part of such a protocol.

fork $\Leftrightarrow$ lock
philosopher $\Leftrightarrow$ thread
dining philosopher problem $\Leftrightarrow$ avoiding deadklocks

2. Locks Cont.

It should not be the case the threads wait in a circular fashion for the locks to be released.

```
(@ [acq(t1,l1),rel(t1,l1)) & acq(t1,l2))
-> (! @ P (@ [acq(t2,l2),rel(t2,l2)) & acq(t2,l1)))
```

3. Passwords

This examples shows another use case where the event stream stores the information about users that have logged in and logged out of with their respective passwords.

3. Passwords

This examples shows another use case where the event stream stores the information about users that have logged in and logged out of with their respective passwords.

- Predicates for quantifying the fact that a user has logged in or logged out.
  $\Leftrightarrow$ `pred login(u,pw) ; pred logout(u)`

3. Passwords

This examples shows another use case where the event stream stores the information about users that have logged in and logged out of with their respective passwords.

- Predicates for quantifying the fact that a user has logged in or logged out.
  $\Leftrightarrow$ `pred login(u,pw) ; pred logout(u)`
- The user is currently logged in.
  $\Leftrightarrow$ `pred isLoggedIn(u) = @ exists pw .`
  `[login(u,pw),logout(u))`

3. Passwords

This examples shows another use case where the event stream stores the information about users that have logged in and logged out of with their respective passwords.

- Predicates for quantifying the fact that a user has logged in or logged out.
  $\Leftrightarrow$ `pred login(u,pw) ; pred logout(u)`
- The user is currently logged in.
  $\Leftrightarrow$ `pred isLoggedIn(u) = @ exists pw . [login(u,pw),logout(u))`
- If a user logged out, he was logged in at some previous state
  $\Leftrightarrow$ `prop p : forall u . logout(u) -> isLoggedIn(u)`

4. Task spawning

Want to formalise a property of tasks that are spawned in an OS: some task y reports back data to some other task x if x spawned y (directly or not).

4. Task spawning

Want to formalise a property of tasks that are spawned in an OS: some task y reports back data to some other task x if x spawned y (directly or not).

Need to take the transitive closure. How can we do that with past-time FOLTL?

4. Task spawning

Want to formalise a property of tasks that are spawned in an OS: some task y reports back data to some other task x if x spawned y (directly or not).

Need to take the transitive closure. How can we do that with past-time FOLTL?

We cannot. But we have recursive rules! Define:

```
spawned(x,y) := @ spawned(x,y)
              | spawn(x,y)
              | Exists z . (@spawned(x,z) & spawn(z,y))
```

And now we can just write

```
report(y,x,d) -> spawned(x,y)
```

5. Telemetry

Consider a radio onboard a spaceship that communicates with Earth over different channels, which can be toggled, and they are initially off.

5. Telemetry

Consider a radio onboard a spaceship that communicates with Earth over different channels, which can be toggled, and they are initially off.

- Rule for OFF state.

  $\Leftrightarrow$ `closed(x) := (!@true & !toggle(x))`
  `| (@closed(x) & !toggle(x))`
  `| (@open(x) & toggle(x))`

5. Telemetry

Consider a radio onboard a spaceship that communicates with Earth over different channels, which can be toggled, and they are initially off.

- Rule for OFF state.

  $\Leftrightarrow$ `closed(x) := (!@true & !toggle(x))`
  `| (@closed(x) & !toggle(x))`
  `| (@open(x) & toggle(x))`

- Rule for ON state.

  $\Leftrightarrow$ `open(x) := (@open(x) & !toggle(x))`
  `| (@closed(x) & toggle(x))`

5. Telmetry Contd.

- Use these rules to design the prop.

  $\Leftrightarrow$

```
prop telemetry2:
    forall x .
      closed(x) -> !telem(x)
    where
    closed(x) :=
    (!@true & !toggle(x))
    | (@closed(x) & !toggle(x))
    | (@open(x) & toggle(x)),
    open(x) :=
    (@open(x) & !toggle(x))
    | (@closed(x) & toggle(x))
```

6. Money request

Here, a user sends over and requests money from other users.

- Define a specification that states a user can only request money that is sent over by another user to the user requesting it.

  $\Leftrightarrow$
  ```
  prop MoneyReqAdv:
      forall r . req(r) -> P exists s .
      [req(r),send(s,r))
  ```