DejaVu: Checking Event Streams against Temporal Logic Formulas

Ruturaj Nanoti, Ruxandra Icleanu

CIS 6730: Computer-Aided Verification

April 2023

1 Introduction

DejaVu [1] is a first-order past-time linear temporal logic based tool for monitoring event streams against temporal logic specifications. It also has a recursion mechanism, and one important use is reasoning with respect to time time, i.e., checking if some previous state of the system satisfies certain properties.

It is developed in Scala (a high level programming language), and it is mostly used for distributed systems and data processing. It utilises **BDD**s (**B**inary **D**ecision **D**iagrams) for assigning variables to quantifiable entities.

We discuss the underlying theory behind DejaVu, and then implement multiple examples which demonstrate the power of the logic the tool is based on.

2 Past Time First-Order LTL

DejaVu properties are expressed in past time temporal logic, which has proven to be [4] exponentially more succinct than future temporal logic. This permits formulating specification that include previous states, allowing the verification of a particular event stream.

According to [3], first-order past linear time temporal logic includes Boolean connectives, the existential quantifier, and the past temporal modalities

- previously, denoted by @
- since, denoted by S

FOPTL can be defined by the following grammar:

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

where $x = \lambda$ is an atomic formula which represents a dynamic test.

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

The tool uses a program (”translate”) that, when given a FO-PLTL property, generates a monitor program. The monitor takes as input a trace (i.e. a sequence of events), and returns a boolean for each position in the trace. [3]

3 About DejaVu

Since DejaVu is an event monitoring tool, it needs to use logs which are *.csv* files. These act as traces which the specification formulas must satisfy. We note that a strict requirement for the log files is that there should not be a comma at the end of each entry (as opposed to Microsft Excel default behaviour), which was one of the problems we faced when testing the tool.

The tool sports robust error messages that indicate precisely which parts of the log file failed to satisfy the property. The errors are divided into two categories: wellformedness errors and property violations. The wellformedness errors terminate the execution of the program and can mean one of the following:

- *syntax error*: a the property file does not follow the syntax and semantics of DejaVu.
- *free variable error*: a variable has been used in the property file that is not defined as a predicate parameter or a quantified variable
- *hiding*: a variable defined at the outer level is hidden due to a quantified expression.
- an *unused variable error*: a variable has been defined and then not used in the property
- *inconsistent error*: a predicate is called with either not enough or more arguments compared to the definition
- *duplicate error*: a predicate or property is defined after it was previously defined
- *undefined event*: an event has been used but not part of the the list of defined events
- *variable duplication error*: when variable or a macro is defined more than once
- *unprotected recursive rule definition*: a rule is called inside a rule definition without the use of the @ primitive, which is a previous time operator.

The second category is represented by the property violation errors - they are triggered by the logs given to DejaVu along with the property files. These errors point out the exact event where there was a violation of the specification formulated in the prop file - this allows for easy and quick debugging.

Along with these errors, there are warnings that arise from the formula specification which can be caused due to an *unused macro* or an *unused event.* As usual, warnings do not terminate the execution of the program.

Furthermore, DejaVu provides trace statistics which show the total number of events and their counts based on their type. At the end of each run, the user can check the execution statistics (the time spent on: parsing the formulated specification, compiling the Scala program, and verifying the event stream).

4 DejaVu's Syntax

The following, extracted from the Grammar Semantics section in [1], presents the syntax of various specification formulae.

```
true, false     : Boolean truth and falsehood
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 S q           : p since q (q was true in the past, and since then, including
                  that point in time, p has been true)
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. This form may be
                  easier to read.
! p             : not 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
```

These forms are used to create formal specification requirements for an event stream. The tool also supports rules which are a part of the specification. We will look at formal specifications in more depth through the examples in the next section.

5 Examples

In the following subsections, we will explore some possible applications of DejaVu that will illustate the important features of the tool.

5.1 Access

This example demonstrates a simple condition for allowing users to access files.

```
prop access :
  Forall user . Forall file .
    access(user,file) ->
      [login(user),logout(user))
        &
      [open(file),close(file))
```

or, equivalently

```
pred loggedIn(u) = [login(u),logout(u))
pred opened(f)   = [open(f),close(f))

prop access :
  Forall u . Forall f .
    access(u,f) -> loggedIn(u) & opened(f)
```

The property requires that for every user, for every file, if the user wants to access the file, then: the user has not logged out since they have logged in, and the file has not been closed since it has been been opened. Note that the second version of the property expresses the same requirement, but using predicate macros.

Now, we will analyse two examples. Consider the following:

```
login,john
open,data
access,john,data
close,data
logout,john
open,data
access,john,data
close,data
```

```
login john
open data
access john data
close data
logout john
open data
close data
```

For the example on the left, the property fails since the user John has logged out and has not logged in back before trying to access the file. The example on the right satisfies the property.

5.2 Files

This example demonstrates the verification of an event trace where different files were opened in the past and closed at a previous step. Following is the prop used:

```
prop file : forall f . close(f) -> exists m . @ [open(f,m),close(f))
```

Let's dissect what this means. Firstly, we have the *forall* form that defines the variable *f*, then there is the *close(f)* → clause that says if a file is closed that implies, which is then succeeded by the *exists m* form that says there exists some mode *m*. Finally, we have *[open(f,m),close(f))* which says not closed since opened.

To put everything together, this prop can be read as: forall f (files) such that (the . primitive) they have been closed, it implies that there exists a mode *m* in which they were opened in a previous state (the @ primitive) and have not been closed since. This is a simple prop that shows the temporal logic based reasoning that DejaVu is based on. Now, we need look at the logfile or the event stream that should satisfy this prop.

```
open,file1,write                                  open,file1,write
open,file2,read                                   open,file2,read
write,file1,hello                                 write,file1,hello
open,file1,read                                   close,file1
read,file2                                        read,file2
close,file2                                       close,file2
open,file1,read                                   close,file1
open,file1,write                                  open,file1,write
write,file1,world                                 write,file1,world
open,file1,read                                   close,file1
```

Consider the aforementioned log on the left, it matches all the requirements put down by the specification file, and hence is successfully verified by DejaVu. Notice that all the *close* events happen only when an *open* event is called in some previous state. This does not hold for the log presented on the right and hence is not verified by DejaVu.

5.3 Locks

5.3.1 Basic

This example illustrates some useful properties in the context of threads acquiring locks.

```
prop locksBasic :
  Forall t . Forall l .
    (
      (sleep(t) -> ![acq(t,l),rel(t,l))) &
      (acq(t,l) -> ! exists s . @ [acq(s,l),rel(s,l))) &
      (rel(t,l) -> @ [acq(t,l),rel(t,l)))
    )
```

The property requires that for every thread t, and for every lock l

- if the thread t is going to sleep (i.e. pauses its execution), then it is not the case that the thread t has not released lock l since it acquired it
  i.e. a thread that is going to sleep should not hold any locks
- if the thread acquired a lock l, then it is not the case that there is some thread s such that in the previous state the thread s has not released the lock l since it acquired it
  i.e. a lock can acquired by at most one thread at a time
- if the thread t is releasing the lock t, then in the previous state it was true that the thread t has not released the lock t since it acquired it
  i.e. a thread can only released a lock it has acquired (and not already released)

Consider the following two examples:

```
acq,t1,l1                                         acq,t1,l1
acq,t2,l2                                         acq,t2,l2
rel,t2,l2                                         rel,t2,l2
rel,t1,l1                                         rel,t1,l1
acq,t1,l1                                         acq,t1,l1
acq,t2,l1                                         rel,t1,l1
rel,t2,l1                                         acq,t3,l3
rel,t1,l1                                         rel t3, l3
acq,t3,l3                                         sleep,t3
sleep,t3
```

In the example on the left, the thread $t2$ tries to acquire lock $l1$, which has already been acquired by $t1$ (so this violates condition 1). Consequently, $t2$ cannot release $l1$ later either, since it never acquired it (thus violating condition 2). Finally, thread $t3$ goes to sleep immediately after acquiring lock $l3$ (violating condition 3). The example on the right fixes all of these problems, and thus satisfies the property.

5.3.2 Deadlocks

This example illustrates the dining philosopher problem, in the context of locks and threads.

First, we state the classical problem [2], originally formulated by Dijkstra: N philosophers dine together at a circular table, each having a fixed place. Between every two philosophers, there is a fork. Each philosopher can either think or eat at a time, and they can only eat if they have both the left and right fork. If a philosopher finishes eating, the philosopher will put down both forks. We want to design a protocols 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.

We omit stating the algorithm, but we do the following observation: taking forks in a cycling manner cannot lead to a solution.

Now, thinking of every fork as a lock, and of every philosopher as a thread, we want to formalise the property described above as a requirement that any potential protocol must satisfy.

```
prop locksDeadlocks :
  Forall t1 . Forall t2 . Forall l1 . Forall l2 .
    (@ [acq(t1,l1),rel(t1,l1)) & acq(t1,l2))
    ->
    (! @ P (@ [acq(t2,l2),rel(t2,l2)) & acq(t2,l1)))
```

The property requires that: if in the previous state it was true that the thread t_1 has not released the lock l_1 since it acquired it, and now t_1 wants to acquire l_2, then it must not be that case that in the previous state there existed a state such that in the previous state thread t_2 has not released the lock l_2 since it acquired it, and now t_1 wants to acquire l_1, i.e. it should not be the case the threads wait in a circular fashion for the locks to be released. This is known as a deadlock (circular wait type).

Consider the following two examples:

```
acq t1 l1                                   acq t1 l1
acq t1 l2                                   acq t1 l2
rel t1 l2                                   rel t1 l2
rel t1 l1                                   rel t1 l1
acq t2 l2                                   acq t2 l1
acq t2 l1                                   acq t2 l2
rel t2 l1                                   rel t2 l1
rel t2 l2                                   rel t2 l2
```

The example on the left fails to satisfy the property, while the example on the right does.

5.4 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. Following is the prop file that shows the specification.

```
pred login(u,pw)
pred logout(u)

pred isLoggedIn(u) = @ exists pw . [login(u,pw),logout(u))

prop p : forall u . logout(u) -> isLoggedIn(u)
```

Here the *pred* macro is used to define a predicate that can be used later in the specification. The *login* predicate takes in the *u* and *pw* parameters, which represent the user and the password. Similarly the *logout* predicate is used to represent an even where the user logs out. Then the *isLoggedin* predicate says that in some previuos states there exists a password such that a user hasn't logged out since logged in. Finally, the prop says that forall users such that a user has logged out, there exists a previous state in which the user was *logged in*, which here means the definitions of the *isLoggedIn* predicate.

Let's consider the following log presented on the right. Notice that the fourth event says that *ann* has logged out even after the fact that the third event is also *ann* logging out. Hence, this log will not satisfy the prop. Therefore, if we add another event to the log after the fourth event saying that *ann* has logged in, we can then verify the event stream, which is shown in the log on the left.

```
login,john,123
login,ann,ABC
logout,john
logout,ann
login,ann,A2B
logout,ann
```

```
login,john,123
login,ann,ABC
logout,john
logout,ann
logout,ann
```

5.5 Task Spawning

This example illustrates a useful property of threads that are spawned in an operating system.

```
prop spawning :
  Forall x . Forall y . Forall d .
    report(y,x,d) -> spawned(x,y)
    where
    spawned(x,y) := @ spawned(x,y) | spawn(x,y) | Exists z . (@spawned(x,z) & spawn(z,y))
```

The property requires that, for every parent thread x, for every child thread y, and for every data d, if y reports back to x the data d, then x must have spawned y.

We define the relation "x spawned y" by a recursive rule, an important feature of DejaVu. More exactly, we say that x spawned by y if either:

- in the previous state it was true that x spawned by y
- x is spawning y
- x spawned y indirectly, through a chain of other tasks (i.e. we need to take the transitive closure)

Consider the following two examples.

```
spawn 0 1
spawn 0 2
spawn 1 3
spawn 1 4
spawn 2 5
spawn 2 6
spawn 3 7
report 7 1 data7-1
report 7 2 data7-2
report 5 4 data5-4
```

```
spawn 0 1
spawn 0 2
spawn 1 3
spawn 1 4
spawn 2 5
spawn 2 6
spawn 3 7
report 7 1 data7-1
report 7 0 data7-0
report 5 2 data5-2
```

If we write the example as a tree, it becomes clear that spawned(x, y) holds only if y is the child/ grand-child/ grand-...-grand-child of x. The examples on the left fails to satisfy the property (task 7 has nit been spawned by 2, so it cannot report back to it, and task 5 has not been spawned by 4). The example on the right satisfies the property.

5.6 Telemtry

This example shows a radio onboard a spaceship that communicates with the earth over different channels, which can be turned on or off, where they are initially off. So, this prop shows a different way to write specifications for a particular task.

```
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))
```

The prop says that *forall x* (where x is a communication channel) such that *x* is *closed*, then it implies that there hasn't been a telemetry. The *closed* rule is defined as a disjunction of three alternatives. The first one says that this rule will be true in the initial state since initially the channel is off and isn't toggled. The second one says that it was closed in some previous state and isn't toggled. Finally, the third one states that the channel was open in a previous state and has been toggled which means it is off.

The *open* rule which is used in the closed rule is a disjunction of two alternatives where the first one says that a channel was open in some previous state and hasn't been toggled. The second one says the channel was closed in some previous state and has been toggled, which means the channel is now open.

Now that, we understand the prop, let's take a look at the log or the event stream. This log shows all the events that happened during the communication process.

```
telem,1
toggle,1
telem,1
telem,2
toggle,1
toggle,2
telem,2
telem,2
telem,1
toggle,2
toggle,3
telem,3
telem,3
telem,1
toggle,3
toggle,1
telem,1
toggle,2
telem,1
telem,2
telem,3
toggle,2
telem,1
telem,2
toggle,1
telem,1
```

```
toggle,1
telem,1
toggle,1
toggle,2
telem,2
telem,2
toggle,1
telem,1
toggle,2
toggle,3
telem,3
telem,3
telem,1
toggle,3
toggle,1
telem,1
toggle,2
telem,1
telem,2
toggle,3
telem,3
toggle,2
telem,1
toggle,2
telem,2
toggle,1
telem,1
```

Notice in the log on the left that initially in the first event the radio is trying to communicate, but that won't be possible so this will fail. Hence, we remove this event from the log. Further more if you look at the fourth event it is trying to communicate over the second channel which also is off initially, and hence this will also need to be removed. Instead of removing these lines we can also add a toggle event for that particular channel, but it's effect will propagate through the whole log which then needs to be evaluated. Then for the ninth event observe that channel 2 has been toggled twice, which means it is closed, and hence the communication will not be possible. Therefore, we add a toggle event to compensate for that. Similarly toggle events need to be added before event 24 and 26 to make this trace comply with our specification.

5.7 Money Request

This is an example that we wrote based on our understanding of the tool. Here, a user sends over money to another user.

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

This prop states that whenever a user has requested money, there has to be a previous state in which money was sent to the user that has requested it. Now, let's consider the following logfiles.

```
send,john,ray
req,ray
send,michael,alice
req,alice
send,alice,john
req,john
send,michael,ray
send,michael,jim
send,jim,ray
send,ray,alice
req,ray
req,ray
req,jim
req,alice
```

```
send,john,ray
req,ray
send,michael,alice
req,alice
send,alice,john
req,john
send,michael,ray
send,michael,jim
send,jim,ray
send,ray,alice
req,ray
req,ray
req,jim
req,alice
req,john
```

The log on the left correctly describes the functionality captured in the prop file and hence this event stream is verified by DejaVu. On the other hand the log on the right, has an additional request event from *john* at the end. Hence, this event stream conforms to the specification required.

5.8 Larger examples

We have tested the properties mentioned in 5.1 - 5.6 on larger data files that were not included here. These files can be found here, along with the confirmations that the log files have indeed satisfied the properties.

References

[1] *DejaVu Tool Documentation.* URL: `https://github.com/havelund/dejavu`.

[2] *Dining philosophers problem.* URL: `https://en.wikipedia.org/wiki/Dining_philosophers_problem`.

[3] Dogan Ulus Klaus Havelund Doron Peled. *DejaVu: A Monitoring Tool for First-Order Temporal Logic.* URL: `https://havelund.com/Publications/dejavu-mtcps-2018.pdf`.

[4] *Temporal Logic with Past is Exponentially More Succinct.* URL: `http://www.lsv.fr/Publis/PAPERS/PDF/NM-succinct.pdf`.