```
 _____       _    __       __
|  __ \     (_)   \ \     / /
| |  | | ___ _  __ \ \  / /   _
| |  | |/ _ \ |/ _` \ \/ / | | |
| |__| |  __/ | (_| |\  /| |_| |
|_____/ \___| |\__,_| \/  \__,_|
          _/ |
         |__/

 Version 1.2, December 28 - 2017
```

Overview

DejaVu is a program written in Scala for monitoring event streams (traces) against temporal logic formulas. The formulas are written in a first-order past time linear temporal logic. An example of a property is the following:

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

The property has the name `closeOnlyOpenFiles` and states that for any file `f`, if a `close(f)` event is observed, then there exists a mode `m` (e.g 'read' or 'write') such that in the previous step (`@`), some time in the past was observed an `open(f,m)` event, and since then no `close(f)` event has been observed.

The implementation uses BDDs (Binary Decision Diagrams) for representing assinments to quantified variables (such as `f` and `m` above).

The DejaVu System consists of:

- README.pdf : this readme file
- dejavu.jar : the dejavu system
- dejavu : script to run the system

Installation of DejaVu

DejavU is implemented in Scala.

1. Install the Scala programming language if not already installed (https://www.scala-lang.org/download)

2. Place the three files mentioned above in some directory **DIR** (standing for the total path to this directory).
3. cd to **DIR** and make the script executable:

```
chmod +x dejavu
```

4. Preferably define an alias in your shell profile to the dejavu script so it can be called from anywhere:

```
alias dejavu=DIR/dejavu
```

Running DejaVu

The script is applied as follows:

```
dejavu <specFile> <logFile> [<bitsPerVariable> [debug]]
```

The specfile (< specFile >) should follow the following grammar:

```
<spec> ::= <prop> ... <prop>
<prop> ::= 'prop' <id> ':' <form>
<form> ::=
     'true'
   | 'false'
   | <id> [ '(' <param> ',' ... ',' <param> `)' ]
   | <form> <binop> <form>
   | '[' <form> ',' <form> ')'
   | <unop> <form>
   | <id> <oper> (<id> | <const>)
   | '(' <form> ')'
   | <quantifier> <id> '.' <form>
<param> ::= <id> | <const>
<const> ::= <string> | <integer>
<binop> ::= '->' | '|' | '&' | 'S'
<unop>  ::= '!' |  '@' | 'P' | 'H'
<oper>  ::= '<' | '<=' | '=' | '>' | '>='
<quantifier> ::= 'exists' | 'forall' | 'Exists' | 'Forall'
```

With the following meaning:

```
prop id : p     : property (LTL formulas) p with name id
true, false     : Boolean truth and falsehood
id(v1,...,vn)   : event 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
[p,q)           : interval notation equivalent to !q S p
! p             : not p
@ p             : in previous state p is true
P p             : in some previous state p is true
H p             : in all previous states p is true
x op k          : x is related to variable or constant k via op
// -- 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
```

The log file (< logFile >) should be in comma separated value format (CSV): http://edoceo.com/utilitas/csv-file-format. For example, a file of the form:

```
login,John,10
login,Ann,42
```

with **no leading spaces** would mean two events:

```
login(John,10)
login(Ann,42)
```

The bits per variable (< bitsPerVariable >) indicates how many bits are assigned to each variable in the BDDs. This parameter is optional with the default value being 20. If the number is too low an error message will be issued during analysis as explained below. A too high number can have impact on the efficiency of the algorithm. Note that the number of values representable by N bits is 2^N, so one in general does not need very large numbers.

The algorithm/implementation will perform garbage collection on allocated BDDs, re-using BDDs that are no

longer needed for checking the property, depending on the form of the formula.

Debugging (debug) The debugging flag can only be provided if also < bitsPerVariable > is provided. Usually one picks a low number of bits for debugging purposes (e.g. 3). The result is debugging output showing the progress of formula evaluation for each event. Amongst the output is BDD graphs visualizable with GraphViz (http://www.graphviz.org).

Results from DejaVu

Property violations The tool will indicate a violation of a property by printing what event number it concerns and what event. For example:

```
*** Property secure violated on event number 701:

############################################################
#### access(John,passwordfile)
############################################################
```

indicates that event number 701 violates the property 'security', and that event is a line in the CSV file having the format:

```
access,John,passwordfile
```

Not enough bits per variable If not enough bits have been allocated for a variable to hold the number of values generated for that variable, an assertion violation like the following is printed:

```
*** java.lang.AssertionError: assertion failed:
    10 bits is not enough to represent variable u.
```

One can/should experiment with BDD sizes.

Other errors Syntax errors in the specifications.

Timing results The system will print the following timings:

- the time spent on parsing spec and synthesizing the monitor (a Scala program)
- the time spent on compiling the synthesized monitor
- the time spent on verifying trace with compiled monitor

Generated files The system will generate the following files (none of which need any attention from the user, but may be informative):

- TraceMonitor.scala : containing the synthesized monitor (a self-sufficient Scala program).
- ast.dot : file showing the structure of the formula (used for generating BDD updating code). This can be viewed with GraphViz (http://www.graphviz.org). These two files help illustrate how the algorithm works.
- dejavu-results : contains numbers of events that violated property if any violations occurred. Mostly used for unit testing purposes.

Authors

The DejaVu system was developed by (in alphabetic order): Klaus Havelund, Doron Peled, and Dogan Ulus.

Enjoy!