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

 Version 1.2.4, April 17 - 2018
```

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).

Installing DejaVu:

The directly `out` contains files and directories useful for installing and running DejaVu:

- README.pdf : this document in pdf format
- dejavu : script to run the system
- artifacts/dejavu_jar/dejavu.jar : the dejavu jar file
- papers : a directory containing papers published about DejaVu
- examples : an example directory containing properties and logs

DejavU is implemented in Scala.

1. Install the Scala programming language if not already installed (https://www.scala-lang.org/download)
2. Place the files `dejavu` and `dejavu.jar` 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 <docFile> <logFile> [<bitsPerVariable> [debug]]
```

- The `<docFile>` is the path to a file containing the specification document.
- The `<logFile>` is the path to a file containing the log in CSV format to be analyzed.
- The `<bitsPerVariable>` is a number indicating how many bits should be assigned to each variable in the BDD representation.
- The `debug` flag will cause debugging output to be generated on standard out.

These fields will be explained in more detail in the following.

The specification document file (`<docFile>`) should follow the following grammar:

```
<doc> ::= <def> ... <def>
<def> ::= <eventdef> | <macrodef> | <propertydef>

<eventdef> ::= 'pred' <event>,...,<event>
<event>    ::= <id> [ '(' <id> ',' ... ',' <id> ')' ]

<macrodef> ::= 'pred' <id> [ '(' <id> ',' ... ',' <id> ')' ] '=' <form>

<propertydef> ::= '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'
```

A specification document `<doc>` consists of a sequence of definitions. A definition `<def>` can either be a predicate/event definition, a predicate macro definition or a property to be checked.

An event definition introduces those events that the property will refer to. If no events are defined then the events are inferred to be those referred to in the property. It may help to reduce errors in properties to declare the events up front. This way each event is mentioned at least twice, with increased chance of avoiding spelling errors in event names and mismatch in number of arguments.

A predicate macro definition `<macrodef>` introduces a named shorthand for a formula, possibly parameterized with variable names (those introduced with quantifiers). For example the following macro definition defines a shorthand representing that a file is open:

```
pred isOpen(f) = !close(f) S open(f)
```

Predicate macros can be called in properties as predicates. Predicate macros can call other predicate macros. The order of the declaration is of no importance. They can furthermore be introduced after as well as before the properties referring to them.

A property definition `<propdef>` introduces a named property, which is a first-order past time temporal formula. The different formulas `<form>` have the following intuitive meaning:

```
prop id : p     : property (LTL formulas) p with name id
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 t
[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
H p             : in all previous states p is true
x op k          : x is related to variable or constant k via op. E.g.: x < 10, x <= y, x = y, 
// -- 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:

```
list,chair,500
bid,chair,700
bid,chair,650
sell,chair
```

with **no leading spaces** would mean the four events:

```
list(chair,500)
bid(chair,700)
bid(chair,650)
sell(chair)
```

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

Wellformedness errors

Error messages will be emitted if the specification document is not wellformed. A wellformedness violation terminates the program, and can be one of the following:

- *Syntax error*: the document does not follow the grammar.
- *Free variable*: a used variable has not been introduced as a predicate parameter or a quantified variable.
- *Hiding*: a quantified expression hides a name introduced at an outer level.
- *Unused variable*: a name introduced as a predicate parameter or quantifier is not used.
- *Inconsistent*: a predicate is called with inconsistent number of arguments compared to definition or other calls.
- *Duplicates*: duplicate definition of a predicate or property name.
- *Undefined event*: events have been defined but an event is used and not amongst the defined ones.
- *Variable duplication*: a variable is introduced more than once in an event or macro parameter list.

A warning does not terminate the program, and can be one of:

- *Unused macro*: a predicate macro is defined but not used.
- *Unused event*: an event is defined but not used.

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

```
*** Property incr violated on event number 3:

#########################################################
#### bid(chair,650)
#########################################################
```

indicates that event number 3 violates the property `incr`, and that event is a line in the CSV file having the format:

```
bid,chair,650
```

Trace statistics A trace statistics is printed, which indicates how many events were processed in total and how they were distributed over the different event types:

```
Processed 1100006 events

==================
Event Counts:
------------------
logout : 20001
open   : 520001
login  : 500000
close  : 40002
access : 20002
==================
```

Warnings will also be here be issued if:

- if there are events in the specification that do not occur in the trace, or dually
- if there are events in the trace that do not occur in the specification

Both can potentially be signs of a flawed specification, but might not need to be.

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 i.
```

One can/should experiment with BDD sizes.

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.

Examples

Auctions

We illustrate the logic with properties that an auction has to satisfy. The following observable events occur during an auction:

- `list(i,r)` : item `i` is listed for auction with the minimal reserve sales price `r` .
- `bid(i,a)` : the bidding of `a` dollars on item `i` .
- `sell(i)` : the selling of item `i` to highest bidder.

An auction system has to satisfy the four properties shown in below expressed over these three kinds of events, using a predicate macro `inAuction(x)` to express when an item `x` is in active auction (when it has been listed and not yet sold):

```
pred inAuction(x) = exists r . @ [list(x,r),sell(x))

prop incr :
  Forall i . Forall a1 . Forall a2 . @ P bid(i,a1) & bid(i,a2) -> a1 < a2

prop sell :
  Forall i . Forall r . P list(i,r) & sell(i) ->  exists a . P bid(i,a) & a >= r

prop open :
  Forall i . Forall a . (bid(i,a) | sell(i)) -> inAuction(i)

prop once :
  Forall i . Forall r . list(i,r) -> ! exists s . @ P list(i,s)
```

Property `incr` states that bidding must be increasing.

Property `sell` states that when an item is sold, there must exist a bidding on that item which is bigger than or equal to the reserve price.

Property `open` states that bidding on and selling of an item are only allowed if the item has been listed and not yet sold.

Finally, Property `once` states that an item can only be listed once.

Locks in a multithreaded system

We observe the following events:

- `acq(t,l)` : thread `t` acquires lock `l`.
- `rel(t,l)` : thread `t` releases lock `l`.
- `read(t,x)` : thread `t` reads variable `x`.
- `write(t,x)` : thread `t` writes variable `x`.
- `sleep(t)` : thread `t` goes to sleep.

Basic properties

The first set of properties are stated as a conjucntion of three sub-properties:

- A thread going to sleep should not hold any locks.
- At most one thread can acquire a lock at a time.
- A thread can only release a lock it has acquired.

This is formalized as follows:

```
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)))
    )
```

No deadlocks

Locks should not be acquired in a cyclic manner amongst threads (dining philosopher problem). That is, if a thread `t1` takes a lock `l1` and then a lock `l2` (without having released `l1`), then at no time should another thread `t2` take the locks in reverse order. Obeying this principle will prevent cyclic deadlocks.

This is formalized as follows:

```
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)))
```

No dataraces:

If two threads access (read or write) the same shared variable, and one of the threads write to the variable, there must exist a lock, which both threads hold whenever they access the variable.

This is formalized as follows:

```
prop locksDataraces :
  Forall t1 . Forall t2 . Forall x .
    (
      (P (read(t1,x) | write(t1,x)))
      &
      (P write(t2,x))
    )
    ->
    Exists l .
      (
        H ((read(t1,x) | write(t1,x)) -> [acq(t1,l),rel(t1,l)))
        &
        H ((read(t2,x) | write(t2,x)) -> [acq(t2,l),rel(t2,l)))
      )
```

Authors

The DejaVu system was developed by: Klaus Havelund, Doron Peled, and Dogan Ulus.

Enjoy!