interactive code checking with Cobra

a Tutorial

Gerard Holzmann
Nimble Research
gholzmann@acm.org

topics covered

1. *background* and principle of operation
   - installation and configuration
   - guide to online documentation
2. *pattern* queries and regular expressions
   - exercises
3. *interactive* queries
   - token attributes
   - sets and ranges
   - functions
   - reading files, libraries
   - exercises
4. *scripted* queries
   - recursive functions
   - associative arrays
   - the query libraries
   - using concurrency
   - exercises
5. *standalone* checkers
   - using concurrency: multi-threaded checkers
6. use of Cobra for *runtime verification*
   - using live data or event-logs

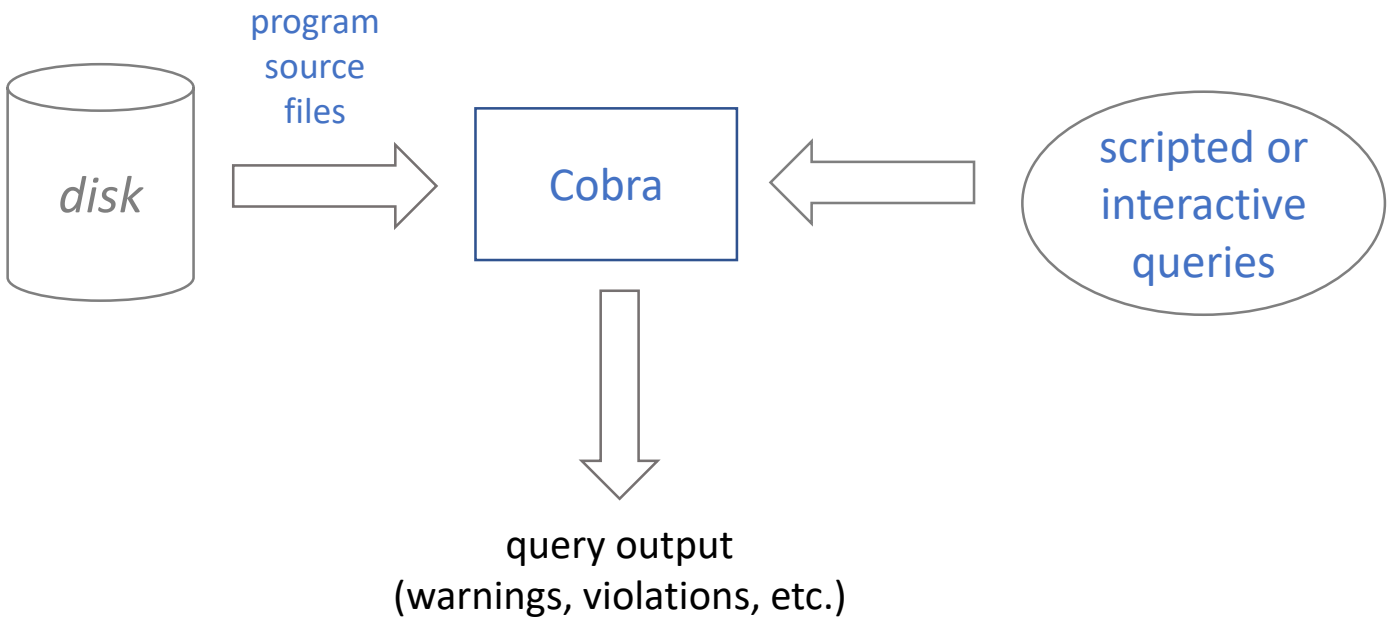
support for runtime verification

new features in Version 3.1

- earlier versions of Cobra, *when not provided with a list of filenames on the command-line*, read-in all data from the standard input (stdin) and then responded to queries
- starting with Version 3.1, Cobra supports two new options:
  1. when reading stdin Cobra reads the data in chunks, so that it can keep up with a arbitrary length event-stream, while executing queries (either query patterns or scripted queries)
  2. the preprocessing to categorize token-types can now be bypassed with a new command-line option –text
     - this increases performance when reading event streams for non-program source inputs

standard use of Cobra

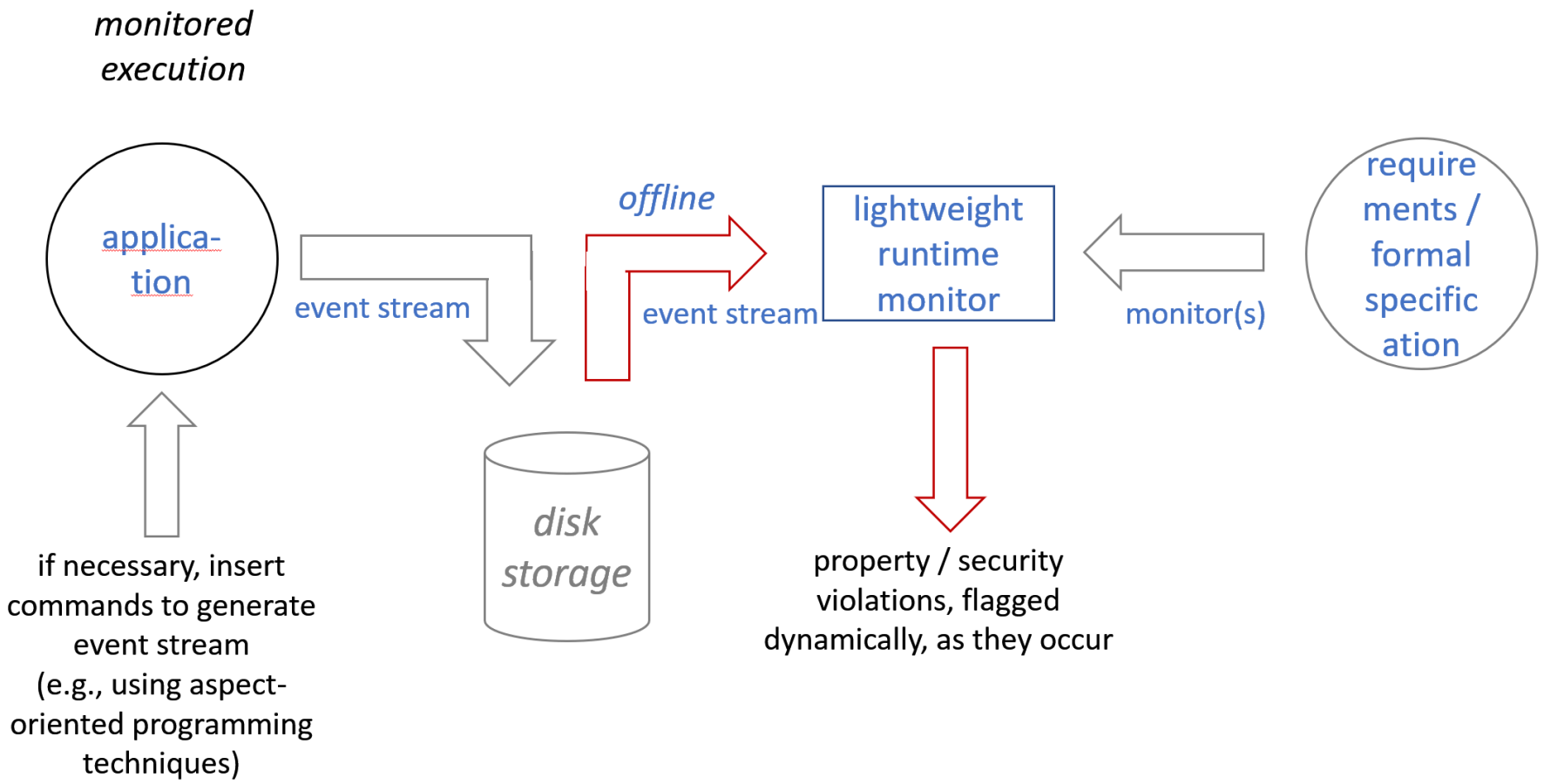
for reference



context of the standard mode of using Cobra

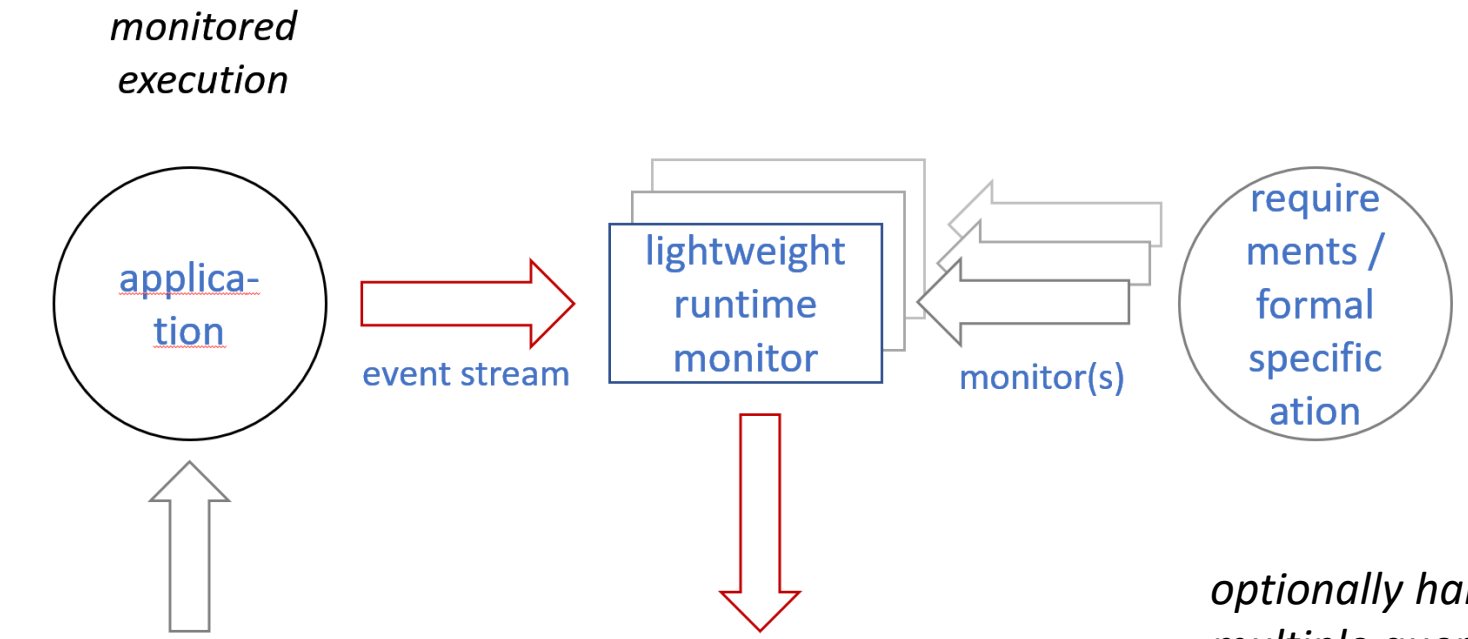
offline use of Cobra for event monitoring

handling recorded event streams, is unchanged



online use of Cobra for event monitoring

new: handling real-time event streams



optionally handling multiple queries in parallel by splitting the event stream

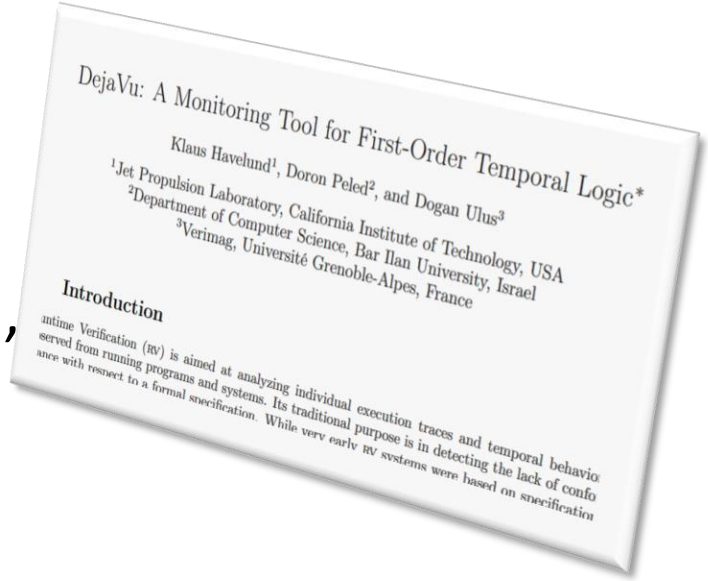
support for runtime verification

new features in Version 3.1

- the earlier behavior when reading from stdin can be recreated by first storing all input from stdin into a file before starting Cobra, and then querying the file contents as before
- the new default behavior for processing data from stdin allows us to keep up with real-time event-streams, but this means that:
  - only a *pattern search* or *command script* can be handled, since the query engine will now only see a sliding window of data
    - the query engine cannot go back to the start of the data stream, to rescan data
  - for the same reason, a command script can only look *forward* in the event/data stream, not backwards; Cobra checks this on startup

an example property

a comparison with the Déjavu runtime verification tool


DejaVu: A Monitoring Tool for First-Order Temporal Logic*

Klaus Havelund[1], Doron Peled[2], and Dogan Ulus[3]

[1]Jet Propulsion Laboratory, California Institute of Technology, USA
[2]Department of Computer Science, Bar Ilan University, Israel
[3]Verimag, Université Grenoble-Alpes, France

Introduction

antime Verification (RV) is aimed at analyzing individual execution traces and temporal behavio
served from running programs and systems. Its traditional purpose is in detecting the lack of confo
ance with respect to a formal specification. While very early RV systems were based on specificatio

- Déjavu is an impressive, existing runtime verification tool, designed to simplify formal specification of a rich set of monitoring properties and using Binary Decision Diagrams (BDDs) to reduce memory requirements
- an example property for monitoring file access, from the Déjavu benchmark suite, is described as:
  - *"If a user accesses a file, then the user must have logged in [and not logged out since then], and the file must have been opened [and not closed since then]."*
  - [the parts in blue are added to the informal English prose from the original, but should be clear from context]

a file access property

in Déjavu and in Cobra specification format

- the property is a good example of how Cobra's specification formalism differs from logic-based formalisms
- In Déjavu the property can be specified in temporal logic, using intervals, quantifiers, and name binding:

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

- the formalization assumes that at least the following five event types are generated by the monitored system, and can be queried by the monitor:
  1. access(user, file)
  2. login(user)
  3. logout(user)
  4. open(file)
  5. close(file)

cobra script

for checking the same property

- the script on the right checks the same property, but is now specified as a Cobra finite state-machine, handling each of the five event-types separately
- information that must be remembered (e.g., bound variable names) is stored in two associative arrays: LoggedIn[] and Opened[]
- property violations are trapped in assertions (or optionally could be reported with print statements)

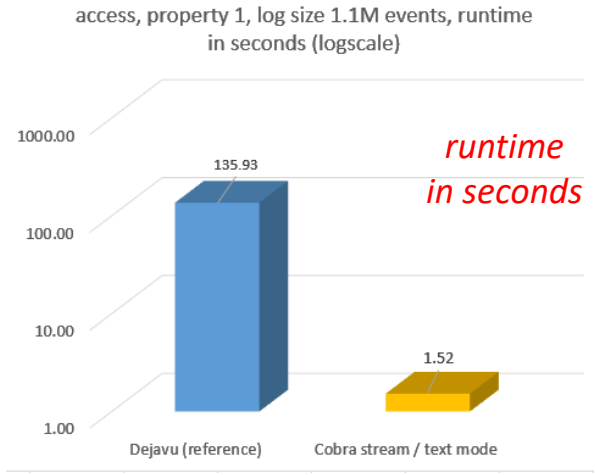
```
%{
if (#login)                     # login , user
{       . = .nxt; . = .nxt;     # user
        LoggedIn[.txt] = 1;
        Next;
}
if (#logout)                    # logout , user
{       . = .nxt; . = .nxt;     # user
        unset LoggedIn[.txt];
        Next;
}
if (#access)                    # access , user , file
{       . = .nxt; . = .nxt; u = .; # user
        . = .nxt; . = .nxt;        # file
        assert(LoggedIn[u.txt] > 0);
        assert(Opened[.txt] > 0);
        Next;
}
if (#open)                      # open , file
{       . = .nxt; . = .nxt;     # file
        Opened[.txt] = 1;
        Next;
}
if (#close)                     # close , file
{       . = .nxt; . = .nxt;     # file
        unset Opened[.txt];
        Next;
}
%}
```

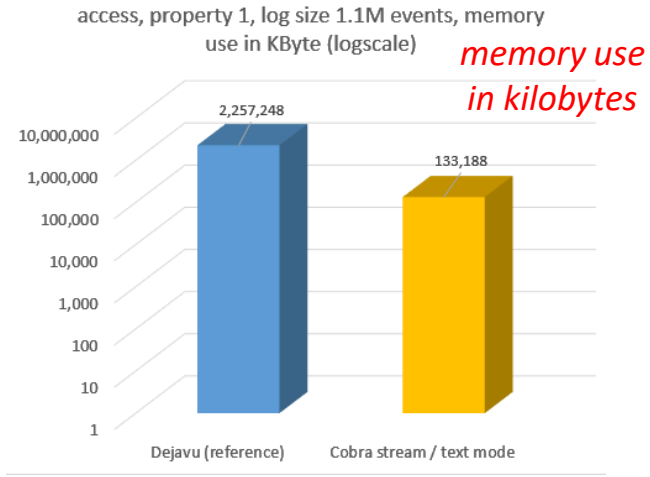
comparison

of these two formalisms

- the logic-based specification is shorter, but also harder to devise and debug
- the scripted version is explicit and easy to debug by adding print statements in various places to see what the monitor's actions are
- the scripting language is Turing complete, and can express anything that could be expressed in the Déjavu formalism (and possibly more)
- it is also simple to tune the scripted version for performance and memory use – this is harder with the logic-based formalism because the internals of the monitoring operations are hidden
- the Cobra tool has the edge in performance

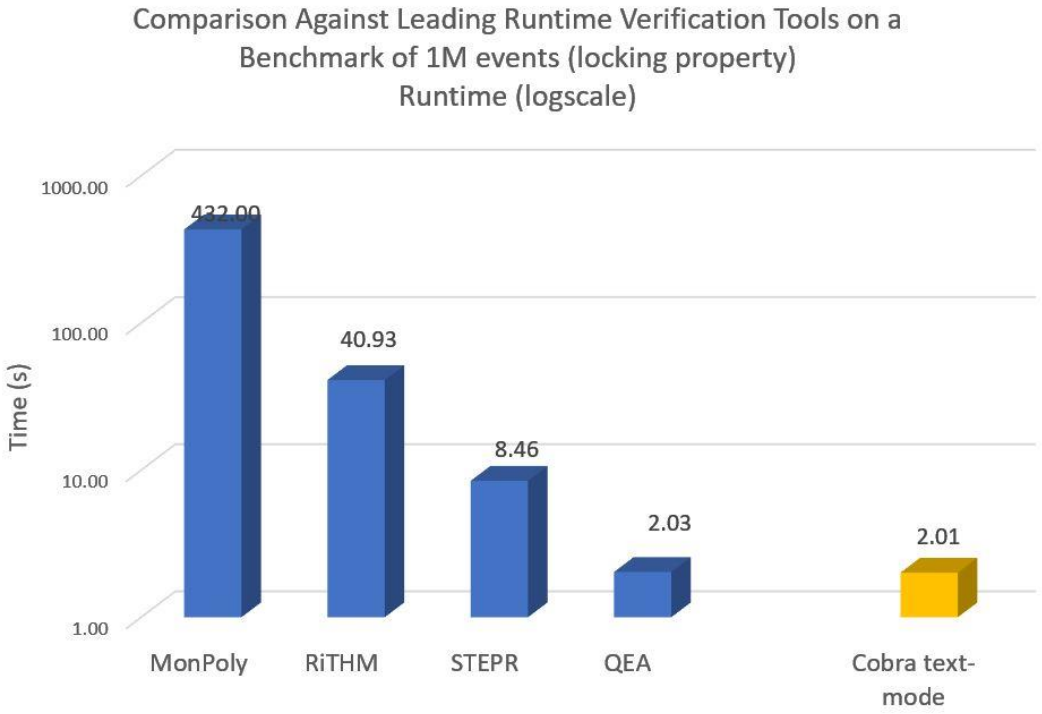
performance:

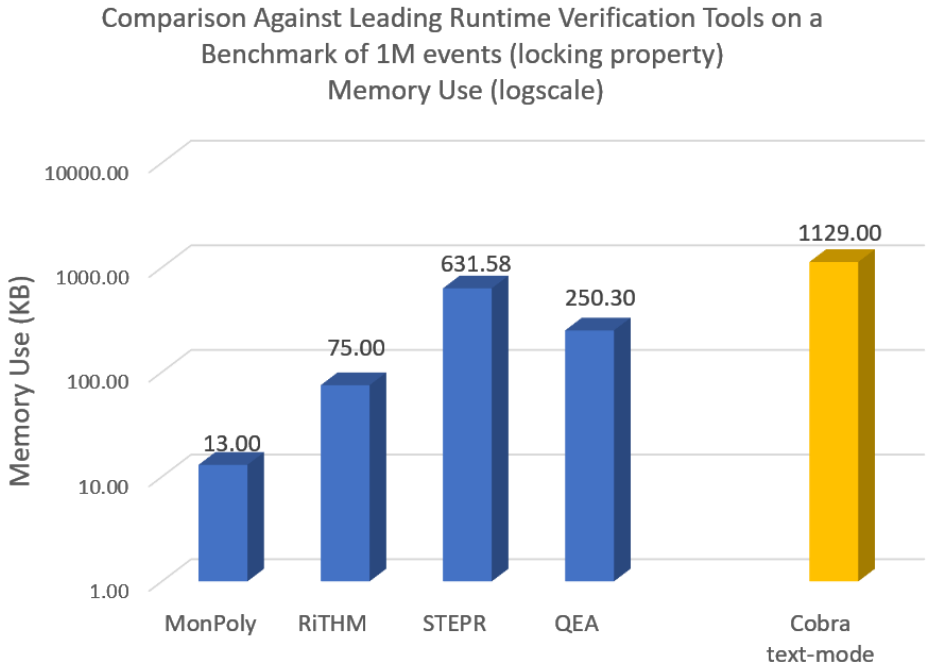




more on performance

a comparison against other leading rv tools shows a similar result



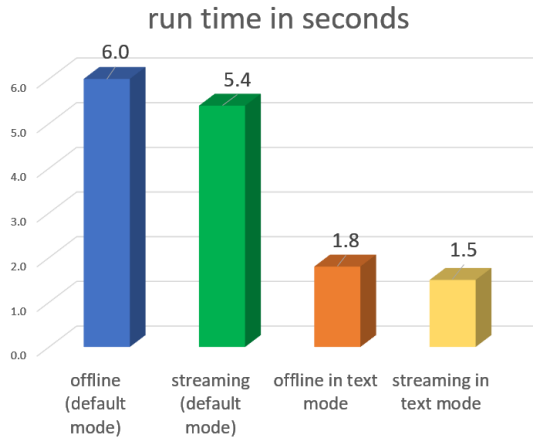


benchmarks from the first international competition on rv tools https://rv-competition.org/benchmarks/
measurements for leading rv tools were performed on a 3.4 GHz system
measurements for the Cobra tool were performed on a 3.2 GHz system

Cobra monitoring modes

when performing runtime verification tasks

- Cobra can be run in different modes when handling runtime verification tasks:
  1. in offline mode, with default settings, reading data from files
  2. in streaming mode, reading stdin, with default preprocessing of input
  3. in offline mode, with the new –text option to suppress preprocessing (categorizing token types in the input data, matching braces in pairs, etc.)
  4. in streaming mode, reading stdin, with the new –text option
- Because only a window of data is stored, memory use is lower in streaming mode, but the amount depends on how much data needs to be remembered (e.g., currenly logged in users, or currently open files)



processing 1.1M events checking a property

exercise

(taken from the Déjavu benchmark set)

- consider an event stream showing auction results. there are three types of events:
  - the listing of an item with a starting price
  - a bid for an item for a given price
  - the sale of an item
- an example event stream is shown on the right, as comma separated values. store this in a file called log1.csv
- write a cobra scripted query that checks that bids on specific items always strictly increase in price, and execute it by reading log1.csv from the standard input
  - that is, if a bid *b2* is submitted on an item, and in the past there was another bid b1 on the same item, then *b2* must be larger than *b1*
  - the desired result is shown at the bottom right

```
$ cat log1.csv
list,hat,100
bid,hat,50
bid,hat,110
sell,hat
list,painting,2000
bid,painting,1000
bid,painting,900
sell,painting
list,painting,1800
bid,painting,1850
sell,painting
bid,table,500
```

```
$ cat prop.cobra
```

```
$ cobra –f prop.cobra < log1.csv
:7: assertion violated at: bid,painting,900
```

congratulations
you've completed this tutorial!

for more information:

manual pages, tutorials, papers:
http://www.spinroot.com/cobra

source code, rule libraries, binaries:
https://github.com/nimble-code/Cobra