Parametric Log Checking

Faculty of Information Technology, Brno University of Technology

Aleš Smrčka < smrcka@fit.vut.cz >

This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 876852. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey.

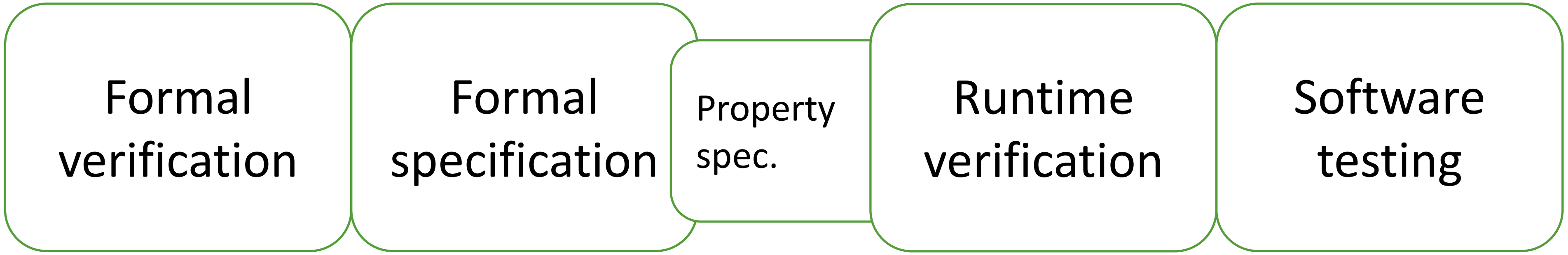
Formal methods

“Program testing can be used to show the presence of bugs, but never to show their absence.” – Edsger Dijkstra

- Formal methods = rigorous mathematical techniques used to specify, develop, and verify systems.
  - Formal specification
  - Formal verification

Static vs. dynamic analysis

- Static analysis
- Dynamic analysis



- State space exploration
- Abstract interpretation
- Q: Does SW (resp. its model) meet its specification?
  - (under specific conditions)

- Experimental evaluation
- Trace analysis
- Q: Did we witness a bug?
  - (in a set of runs)

Software testing vs. Runtime verification

- ”Classical” SW testing:

  - (1) Setup, (2) Exercise, (3) Verify, (4) Teardown (shared fixture strategies…)
  - Systematic/random (coverage driven, fuzz-testing, …)
  - Q1: What to test next?
  - Q2: Is it enough?

Test input ← | → Exp. output

Setup	Exercise	Verify
env1	input1	assert1
env1	input2	assert2
…	…	…

```
// setup
Jarvis.reset_scheduler()
cart = Cart(2, 50, 0)
c = CartCtl(cart, Jarvis)
// exercise
item = CargoReq('A', 'B', 20, 'helmet‘)
Jarvis.plan(0, add_load, (c, item))
Jarvis.run()
// verify
self.assertEqual(cart.pos, 'B‘)
self.assertTrue(cart.empty())
self.assertEqual(Status.Idle, c.status)
self.assertEqual(helmet.context, 'unloaded')
```

Software testing vs. Runtime verification

- ”Classical” SW testing:

  - (1) Setup, (2) Exercise, (3) Verify, (4) Teardown (shared fixture strategies…)
  - Systematic/random (coverage driven, fuzz-testing, …)
  - Q1: What to test next?
  - Q2: Is it enough?

Test input ← | → Exp. output

Setup	Exercise	Verify
env1	input1	assert1 = expect(input1)
env1	input2	assert2 = expect(input2)
…	…	…

```
// setup
Jarvis.reset_scheduler()
cart = Cart(2, 50, 0)
c = CartCtl(cart, Jarvis)
// exercise
item = CargoReq('A', 'B', 20, 'helmet‘)
Jarvis.plan(0, add_load, (c, item))
Jarvis.run()
// verify
self.assertEqual(cart.pos, 'B‘)
self.assertTrue(cart.empty())
self.assertEqual(Status.Idle, c.status)
self.assertEqual(helmet.context, 'unloaded')
```

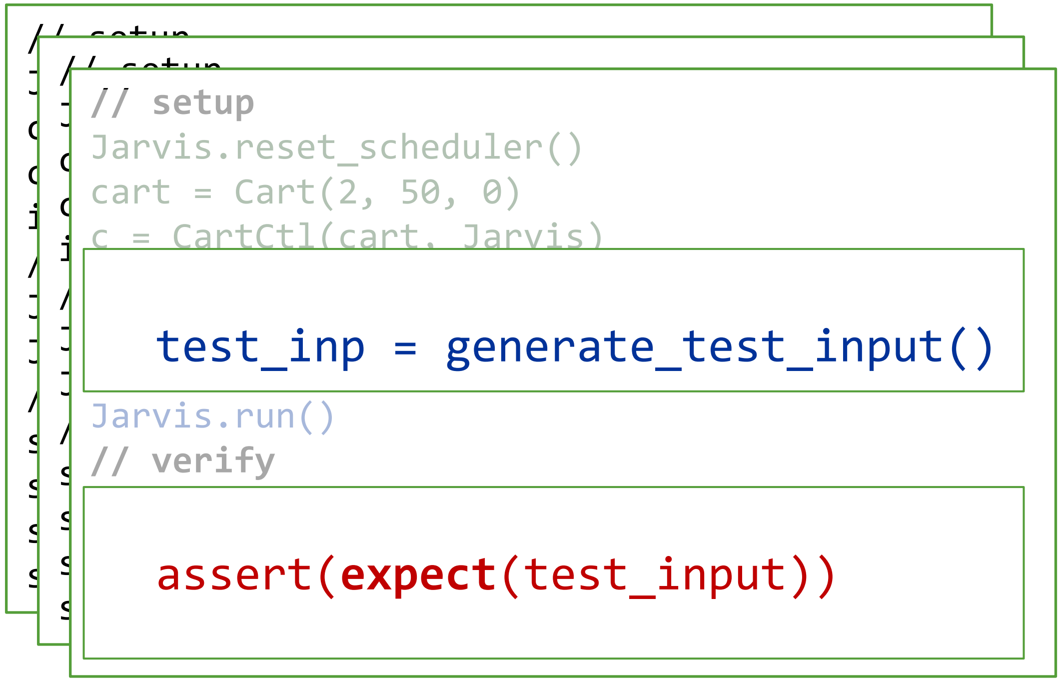
Software testing vs. Runtime verification

- ”Not so Classical” SW testing:

  - (1) Setup, (2) Exercise, (3) Verify, (4) Teardown (shared fixture strategies…)
  - Systematic/random (coverage driven, fuzz-testing, …)
  - Q1: What to test next?
  - Q2: Is it enough?

Test input ← | → Exp. output

Setup	Exercise	Verify
env1	input1	assert1 = expect(input1)
env1	input2	assert2 = expect(input2)
…	…	…



Software testing vs. Runtime verification (assert) part

- Fully automated test:
  - Bug-hunting ultimate Q: Under which condition the system behaves incorrectly?
  - Problem1: How to generate inputs? (cover test items, fuzz)
  - Problem2: Who provides me with the `expect()` function?

Test input ← | → Exp. output

Setup	Exercise	Verify
env1	input1	assert1 = expect(input1)
env1	input2	assert2 = expect(input2)
…	…	…

```
Jarvis.reset_scheduler()
cart = Cart(2, 50, 0)
c = CartCtl(cart, Jarvis)
while 42:
  test_inp = generate_test_input()
Jarvis.run()
// verify

  assert(expect(test_input))
```

Runtime verification – expect() function?

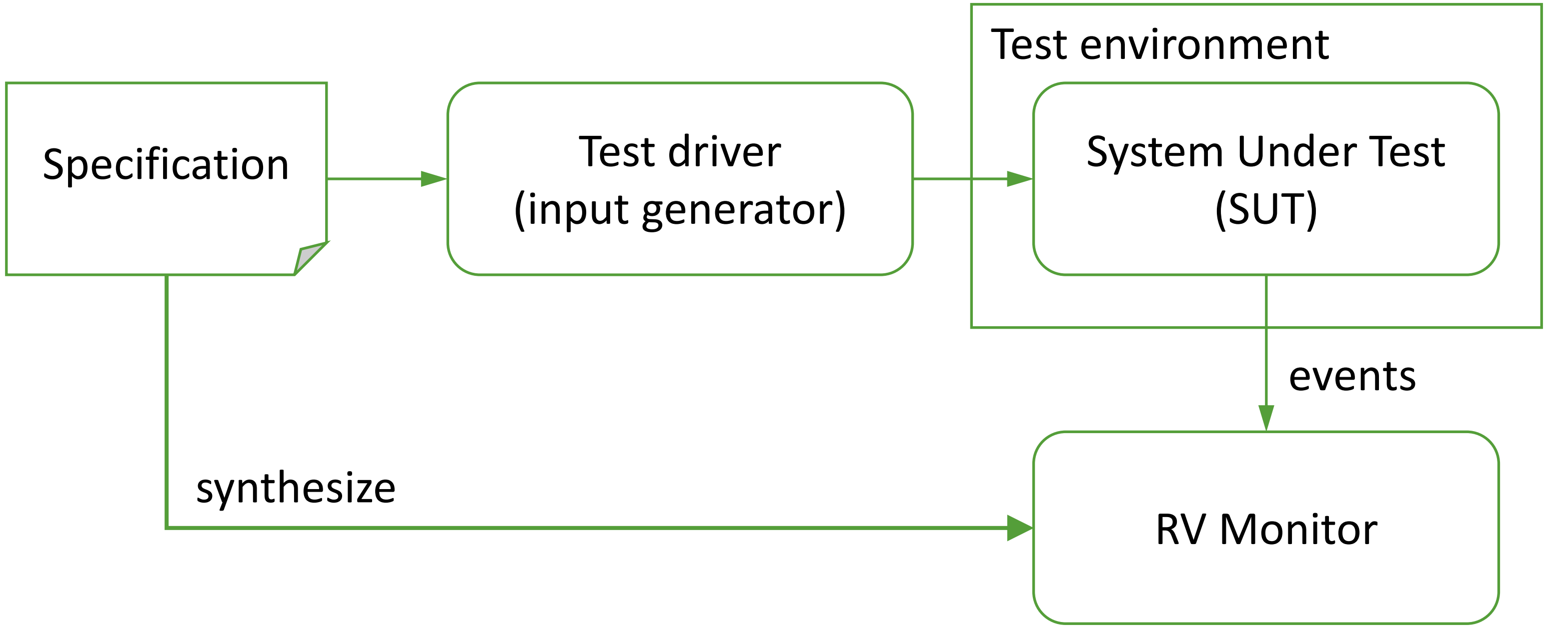
- There isn't "someone". There's just you. ☹
- Where does it come from?
  - From the SUT documentation & specification…
- Except when you are working on high critical-level app, there is no such thing as precise documentation. ☹☹
- Gap between developers and RV/FM practitioners:
  - How to easily write formal specification.
- Moreover, we don't want "expect" to represent just a set of post-conditions, rather:
- **Properties representing expected sequence of events in the SUT.**

Example: $[](\, request(d) \rightarrow \langle\rangle\, response(d))$

Runtime verification – properties

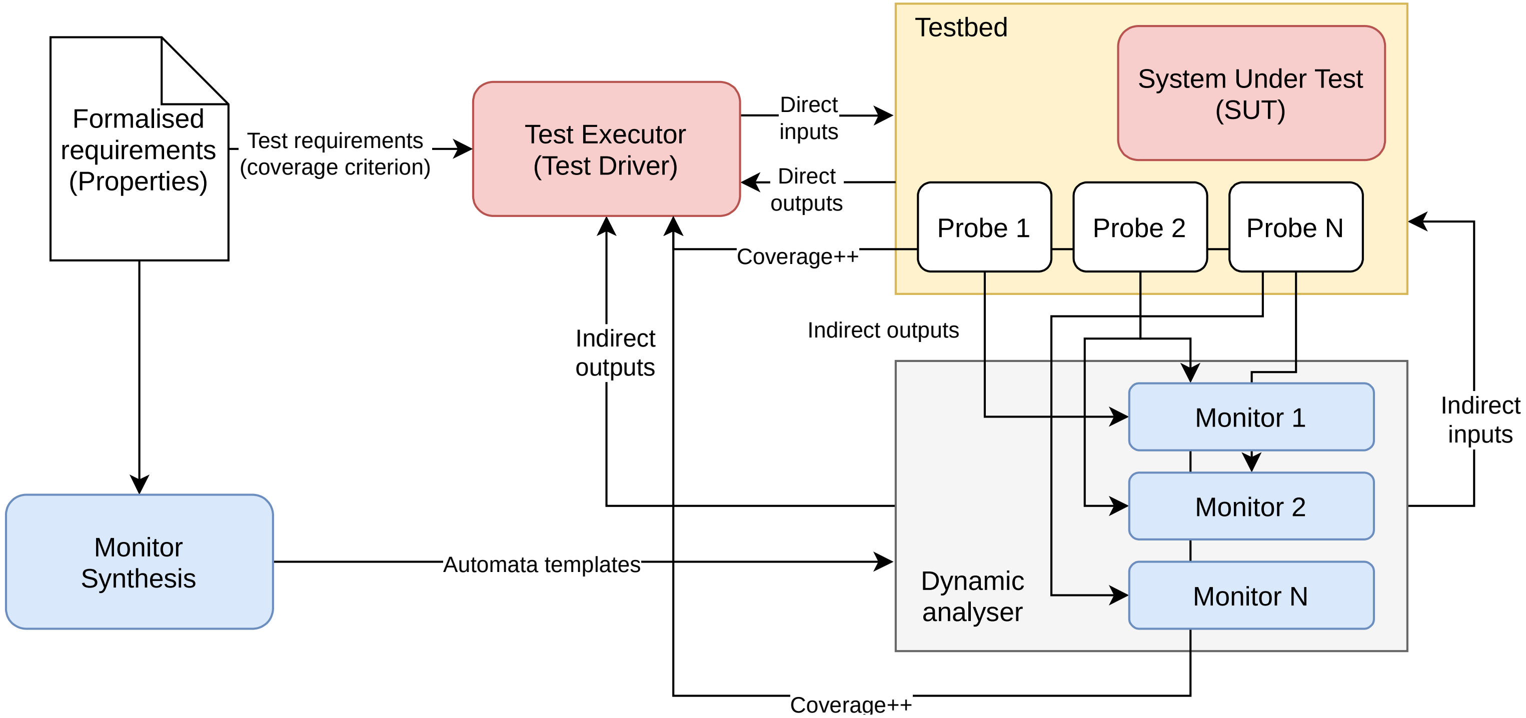
- Used different formalisms:
  - extended regular expressions,
  - state machines,
  - context-free grammar,
  - PSL, LTL, ptLTL, MTL, …
- How to monitor liveness?
  - Liveness is important for potentially endless programs.
  - RV cannot notify about violation of liveness during execution → We cannot decide if the system will eventually be back into an expected state. Two possible workarounds:
    - System must react in a specific time limit (add & check "heartbeat" in the system).
    - PSL extension with non true/false verdicts: Risk and Pending (all future obligations are/are not fulfilled).

Runtime verification – architecture



- Specification of SUT is synthesized to runtime monitors.
- SUT to exhibits sequence of events.
- Runtime monitors (or so called dynamic analysers) checks whether the next occurrence of an event is acceptable wrt. specification.

Runtime verification – classification



- Type of events? Who will provide them? Probe (code) injection.
- Inline vs. Outline analysers (monitor + probes as one or isolated)
- Online vs. Offline (runtime monitor vs. post-mortem analysis)

Runtime verification – instrumentation

- Code injection (probes and/or monitors) techniques:
  - Changing (adding/decorating) source code.
  - Instrumenting while compiling (a special compiler adds the code to join points).
  - Changing the binary when the program is loaded into the memory.
  - Using a code interceptors at runtime.
- Or, separate program communicating with SUT.
  - tracers,
  - log monitors, or even
  - profilers.

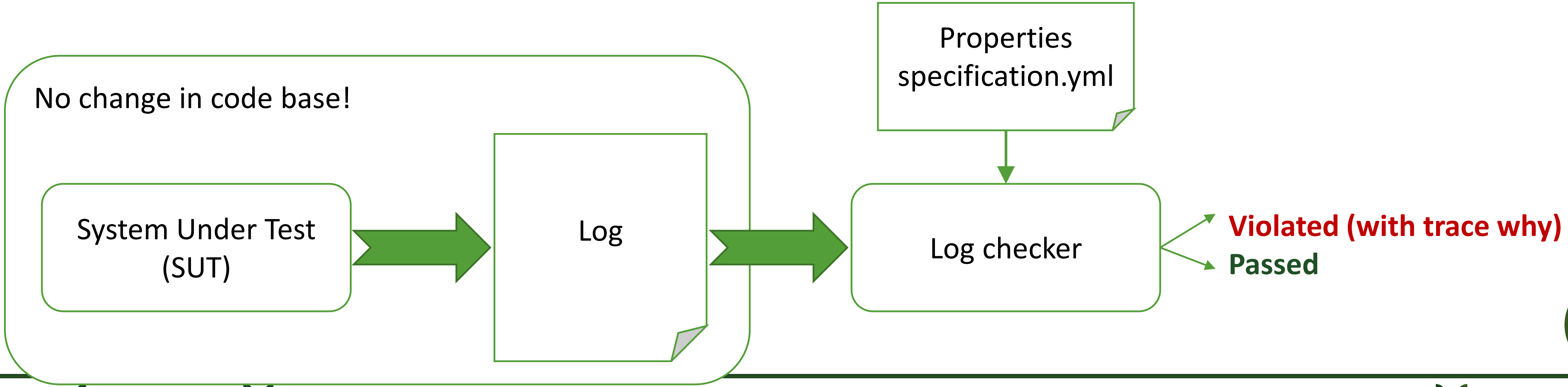
Runtime verification vs. runtime monitoring

- RV: Q: Does it behave correctly under different conditions (even unknown in advance)?
  - i.e. “I will give you inputs, you will provide me with pass/fail.”

- RM: Target: production system
  - Test environment + Test input generator → Real environment + Real input
  - Potentially endless programs:
    - Services, control systems, sensoring systems
- Q: Does it still behave correctly?
- If not, possible actions based on criticality level of the problem:
  - Notify.
  - Switch to partial-operational or safe-mode.
  - Restart (+ self healing).
  - Mission fail.

Log checking

- No code change – separate program,
- Outline RV – SUT generates event in logs, log checker reads it,
- Online/Offline RV – can analyse log as a stream,
- Most importantly, aim at easy specification:
  - Every developer should be able to specify property.



Log checking – property specification

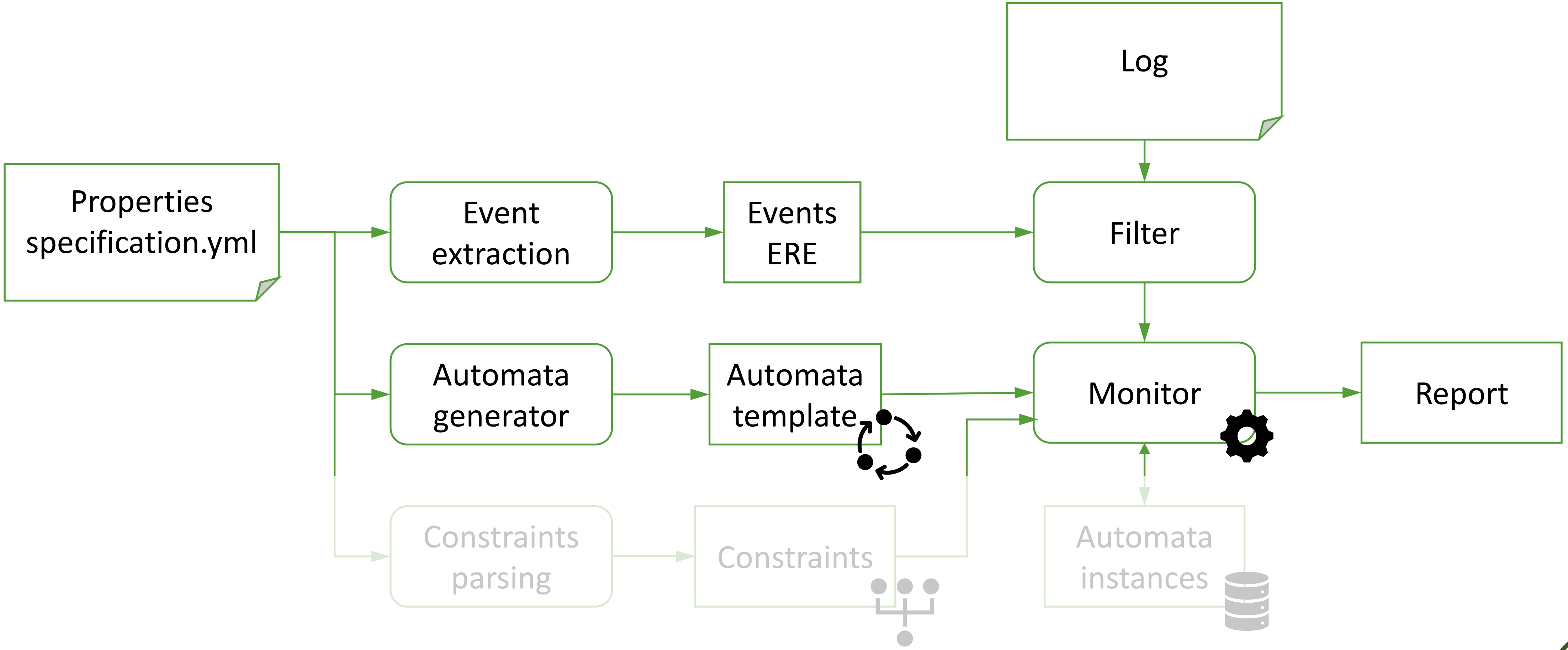
- Good vs. bad properties = expected or unexpected event sequences
- Good property:
  - Once the first event occurs, the whole sequence must occur.
  - Example: If a datastorage is opened, then possibly accessed, it must be closed.
    - “Open Use* Close”
  - Note: Violation can be witnessed at the end of the trace.
- Bad property:
  - If a sequence of events is witnessed, report the violation.
  - Example: Every response to a request to the store must be of status 200.
    - “RequestStore StatusOtherThan200”

Log checking – example httpd reload

```
bad_property:
  FailedReload: “HttpdReload HttpdFailed”
  FailedReload2: “HttpdReload HttpdConfigured! HttpdFailed”
events:
  HttpdReload: “.* systemd\[1\]: Reloading The Apache HTTP Server.”
  HttpdFailed: “.* systemd\[1\]: Reload failed for The Apache HTTP Server.”
  HttpdConfigured: “.* httpd\[\d+\]: Server configured, listening on: .*”
```

```
Aug 14 12:03:37 systemd[1]: Reloading The Apache HTTP Server.
Aug 14 12:03:37 httpd[3296577]: AH00526: Syntax error on line 1 of
/etc/httpd/conf/httpd.conf:
Aug 14 12:03:37 httpd[3296577]: Invalid command 'asdf', perhaps misspelled
or defined by a module not included in the server configuration
Aug 14 12:03:37 systemd[1]: httpd.service: Control process exited,
code=exited, status=1/FAILURE
Aug 14 12:03:37 systemd[1]: Reload failed for The Apache HTTP Server.
```

Log checking – how does it work?



Log checking – example httpd reload

```
Bad property 'HttpdReload HttpdConfigured! HttpdFailed' with id
'FailedReload2' was violated!
    Sequence of events that caused the violation:
      1. Aug …systemd[1]: Reloading The Apache HTTP Server.
        event id = HttpdReload
      5. Aug …systemd[1]: Reload failed for The Apache HTTP Server.
        event id = HttpdFailed
```

```
Aug 14 12:03:37 systemd[1]: Reloading The Apache HTTP Server.
Aug 14 12:03:37 httpd[3296577]: AH00526: Syntax error on line 1 of
/etc/httpd/conf/httpd.conf:
Aug 14 12:03:37 httpd[3296577]: Invalid command 'asdf', perhaps misspelled
or defined by a module not included in the server configuration
Aug 14 12:03:37 systemd[1]: httpd.service: Control process exited,
code=exited, status=1/FAILURE
Aug 14 12:03:37 systemd[1]: Reload failed for The Apache HTTP Server.
```

Log checking – parameters

- Consider property: “Open Use* Close”
  - “Every file which is opened, then used, must be eventually closed.”
- The trace of events with multiple violations:
  - Open(f1) Use(f1) Use(f1) Open(f2) Use(f2) Close(f1) Use(f2)
- We need to take into account parameters → Parametrised sequences: “Open(f) Use(f)* Close(f)”
- Events in a parametrised sequence share parameter values:
  - f1: Open(f1) Use(f1) Use(f1) Open(f2) Use(f2) Close(f1) Use(f2)
  - f2: Open(f1) Use(f1) Use(f1) Open(f2) Use(f2) Close(f1) Use(f2)

Log checking – more parameters

- Parameterised sequence can have more parameters:
  - Bad_property: “Create(c, i) Next(i)* Update(c) Next(i)”
  - Once an iterator over collection is created, it must not be used after collection update.

```
bad_property:
  UnsafeIter: “Create Next* Update Next”

events:
  Create: “%{WORD:col}.begin = %{WORD:it}”
  Next: “%{WORD:it}.next”
  Update: “%{WORD:col}.(remove|add|clear)”

constraints:
  - “Create.col = Update.col”
  - “Create.it = Next.it”
```

Log checking – example sshd

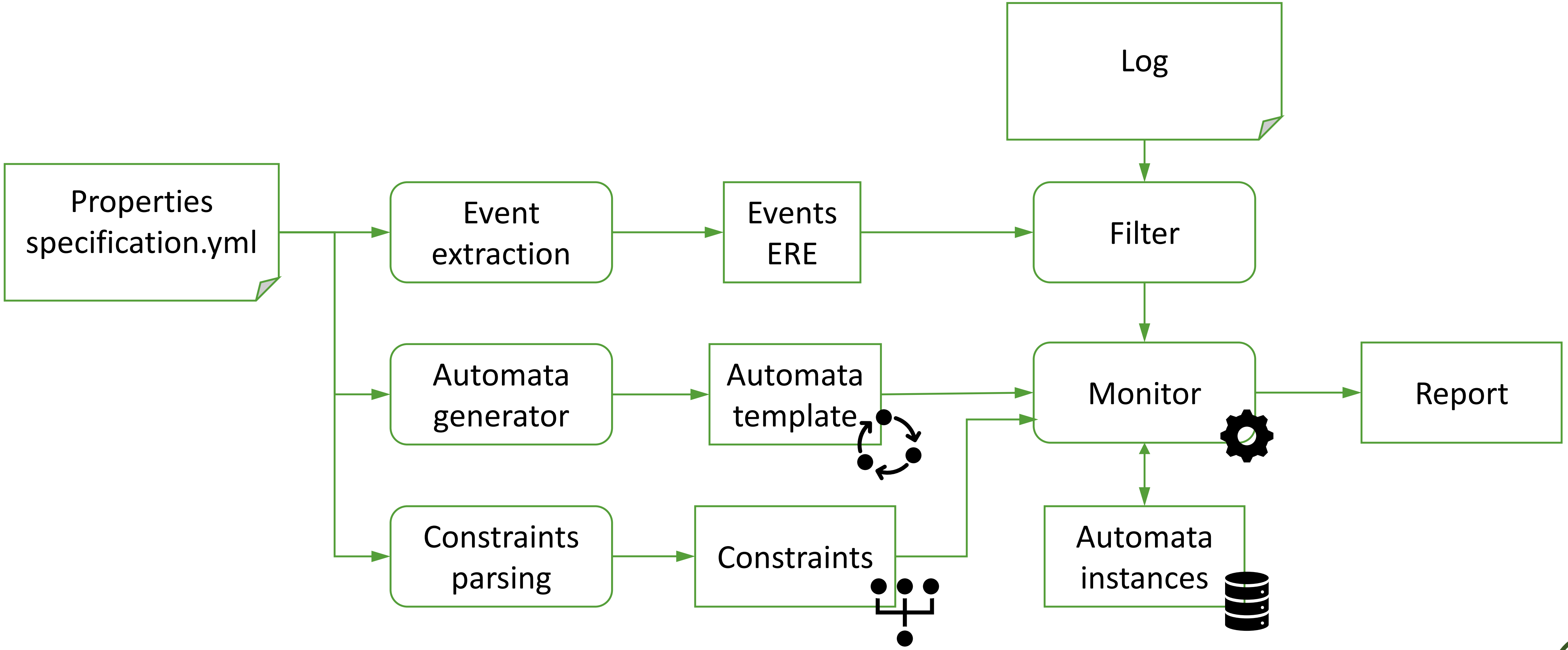
```
bad_properties:
  BFAttack: "F1 F2+ F3"

events:
  F1: "%{DATE_ISO8601:ts} .* sshd: Failed password for invalid user %{WORD:user} from %{IP:ip}"
  F2: "%{DATE_ISO8601:ts} .* sshd: Failed password for invalid user %{WORD:user} from %{IP:ip}"
  F3: "%{DATE_ISO8601:ts} .* sshd: Failed password for invalid user %{WORD:user} from %{IP:ip}"

constraints:
  - "F3.ts - F1.ts <= 0h0m10s"
  - "F1.ts < F3.ts"
  - "F1.ip = F2.ip"
  - "F2.ip = F3.ip"
```

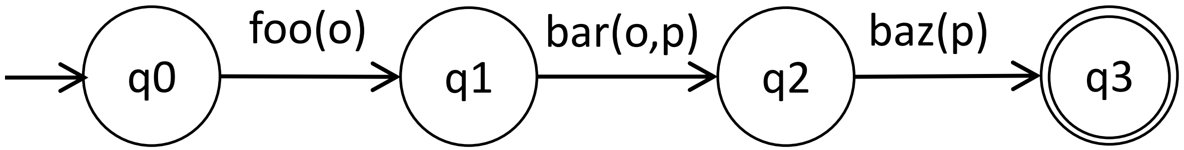
```
2022-09-15 05:43:10 wee sshd: Failed password for invalid user root from 61.177.173.10
2022-09-15 05:43:12 wee sshd: Failed password for invalid user root from 92.255.85.70
2022-09-15 05:43:23 wee sshd: Failed password for invalid user root from 61.177.173.10
2022-09-15 05:43:24 wee sshd: Failed password for invalid user mpb from 39.109.113.139
2022-09-15 05:43:25 wee sshd: Failed password for invalid user root from 61.177.173.10
2022-09-15 05:43:27 wee sshd: Failed password for invalid user root from 61.177.173.10
2022-09-15 05:43:37 wee sshd: Failed password for invalid user tomcat from 193.106.191.157
```

Log checking – parameters, how does it work?



Log checking – parameters, how does it work?

Property: foo(o) bar(o,p) baz(p) **FA template:**



Line	Trace	o=1	o=2	p=a	q=b
1	foo(1)	q1[1/o]			
2	bar(1,a)			q2[a/p]	
3	foo(2)		q1[2/o]		
4	bar(1,b)			q2[b/p]	
5	bar(2,b)			q2[b/p]	
6	baz(b)				q3[b/q]
7	foo(1)	q1[1/o]			

Plogchecker

- Prototype tool (v2.0), implemented in Golang:
  `plogchecker [-p prop.yml] [-l logfile] [-s JSON|TEXT]`
- Example: `$ journalctl | plogchecker -l sshdprop.yml -s JSON`
- Automata + parameters:
  - Every specified event has a collection of seen parameter values of the event.
  - Parameter values (collection items) reference to other collections (their predecessors in the sequence).
  - Once reached final event, the whole sequence is erased and possibly reported.
- Inefficient: More unique values or their combinations in the log mean more memory consumption.
  → Different approaches to store the parameters or to represent automata.
- Garbage collector:
  - Safe = all seen values are preserved (too consuming), only closed seqs. are cleared.
  - **Unsafe** = we may miss some faults, but it can still provide useful feedback.

Plogchecker – example RemoteStation-Car

```
Timestamp,RS name,ID,Event type,Data
1614582126860,Roboauto-RS-NLR-E01,0,RSEVENT,connectedToGateway
1614582147614,Roboauto-RS-NLR-E01,1,RSEVENT,connectingToVehicle
1614582148114,Roboauto-RS-NLR-E01,49,RSCOMMAND,ReadyAck
1614582148114,Roboauto-RS-NLR-E01,50,RSEVENT,connectedToVehicle
1614582187719,Roboauto-RS-NLR-E01,75434,RSCOMMAND,DriveAck
1614582187719,Roboauto-RS-NLR-E01,75435,RSEVENT,drivingVehicle
1614582220730,Roboauto-RS-NLR-E01,143104,RSCOMMAND,PauseAck
1614582220730,Roboauto-RS-NLR-E01,143105,RSEVENT,connectedToVehicle
1614582224075,Roboauto-RS-NLR-E01,149700,RSCOMMAND,DriveAck
1614582224075,Roboauto-RS-NLR-E01,149701,RSEVENT,drivingVehicle
1614582238584,Roboauto-RS-NLR-E01,179557,RSCOMMAND,PauseAck
1614582238584,Roboauto-RS-NLR-E01,179558,RSEVENT,connectedToVehicle
```

```
$ plogchecker -p UC2_RS_cmds.yml -l lab_robot.csv
{
  "properties": {
  ”rs_pause": {
  "property": ”P D E F",
  "violated": [{
    ...
"events_sequence": [{
  "event_id": ”P",
  "log_file": "lab_robot.csv",
  "log_lineno": 143106,
  "log_line": "1614582220730,Roboauto-RS-NLR-
E01,143104,RSCOMMAND,PauseAck"
} ...
```

```
properties:

    # connection established...drivingVehicle
    rs_conn: “A B O* C O* D O* E O* F O*”

    # pause in the middle and continue
    rs_pause: “P D E F”

    # close connection, odometry receive possible
    rs_end: “F O* R”

events:
    A: .*RSEVENT,connectedToGateway
    B: .*RSEVENT,connectingToVehicle
    C: .*RSCOMMAND,ReadyAck
    D: .*RSEVENT,connectedToVehicle
    E: .*RSCOMMAND,DriveAck
    F: .*RSEVENT,drivingVehicle
    O: .*RSODOMETRY,
    P: .*RSCOMMAND,PauseAck
    R: .*RSCOMMAND,Release
```

Plogchecker – example Open-Shadow

1

SUT

```
strace  grep ^root: /etc/shadow
```

LOG

2

```
execve("/bin/grep", ["grep", "^root:",
"/etc/shadow"], 0x7ffffd6b9c30, /* 14 vars */ ) = 0
arch_prctl(ARCH_SET_FS, 0x7f667f70ab48) = 0
set_tid_address(0x7f667f70af90)         = 138
mprotect(0x7f667f707000, 4096, PROT_READ) = 0
mprotect(0x55a28597e000, 16384, PROT_READ) = 0
getuid()                                = 0
open("/etc/shadow", O_RDONLY|O_LARGEFILE) = 3
mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f667f673000
read(3, "root:!::0:::::\nbin:"..., 1024) = 448
mmap(NULL, 16384, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f667f66f000
munmap(0x7f667f66f000, 16384)           = 0
ioctl(1, TIOCGWINSZ, {ws_row=52, ws_col=94,
ws_xpixel=0, ws_ypixel=0}) = 0
writev(1, [{iov_base="root:!::0:::::", iov_len=14},
{iov_base="\n", iov_len=1}], 2) = 15
read(3, "", 1024)                       = 0
close(3)                                = 0
munmap(0x7f667f673000, 4096)            = 0
```

shadow.yml

Specification

3

```
bad_properties:
  shadow: "O R"
events:
  O: 'open(at)?\("/etc/shadow",.* = %{NUMBER:fd}$'
  R: 'read\(%{NUMBER:fd},.* = %{NUMBER:bytes}$'
constraints:
  - "O.fd >= 0"
  - "O.fd = R.fd"
  - "R.bytes > 0"
```

4

```
$ strace … | plogchecker -p shadow.yml -s TEXT
Bad property 'O R' with id 'shadow' was violated!
   Sequence of events that caused the violation:
  7. openat("/etc/shadow", O_RDONLY|O_LARGEFILE) = 3
        event id = O
 10. read(3, "root:!::0:::::\nbin"..., 1024) = 448
        event id = R
```

Conclusion

- Runtime verification improves your software testing.
- Runtime verification complements formal verification when:
  - it is computationally impossible to exhaustively verify,
  - it is too expensive/time consuming (missing well-suited tools/skills),
  - model assumptions do not meet the real system.
- Log checking:
  - relatively easy to specify properties,
  - possible wide acceptance (almost all systems log their events),
  - enables post-mortem/offline analysis,
  - can be used in production environment.
- Still much work to make it universal, e.g.:
  - efficient garbage collector for parametrised instances,
  - high throughput + low overhead: currently 2.7k lines/sec (Ryzen7), >30MB worst case
  - more data types (e.g. json),
  - multi-line log events,
  - other than text logs.

More on Runtime verification

- Runtime Verification conference: https://runtime-verification.github.io/
- D. Jin, M. Dennis, and G.G. Rosu, "Garbage collection for monitoring parametric properties", 2011. doi: 10.1.1.360.5666
- D. Jin, P. O. Meredith, C. Lee and G. Roşu, "JavaMOP: Efficient parametric runtime monitoring framework," *ICSE* 2012, pp. 1427-1430, doi: 10.1109/ICSE.2012.6227231.
- Chen, Z., Wang, Z., Zhu, Y., Xi, H., Yang, Z. (2016). "Parametric Runtime Verification of C Programs". In: Chechik, M., Raskin, JF. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2016. LNCS 9636. Springer. doi: 10.1007/978-3-662-49674-9_17 .
- Zhe Chen, Parametric runtime verification is NP-complete and coNP-complete, Information Processing Letters, Volume 123, 2017, Pages 14-20, ISSN 0020-0190, doi: 10.1016/j.ipl.2017.02.006.
- de Oliveira, D.B., Cucinotta, T., de Oliveira, R.S. (2019). Efficient Formal Verification for the Linux Kernel. In: Ölveczky, P., Salaün, G. (eds) Software Engineering and Formal Methods. SEFM 2019. Lecture Notes in Computer Science(), vol 11724. Springer, Cham. doi: 10.1007/978-3-030-30446-1_17
  - Runtime Verification – The Linux Kernel Documentation (2022-07-29)
  - https://kernel.org/doc/html/latest//trace/rv/runtime-verification.html

DISCLAIMER: The ECSEL JU and the European Commission are not responsible for the content of this video or any use that may be made of the information it contains

VALU3S

Verification and Validation of Automated Systems' Safety and Security

This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 876852. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey.

www.valu3s.eu

YouTube