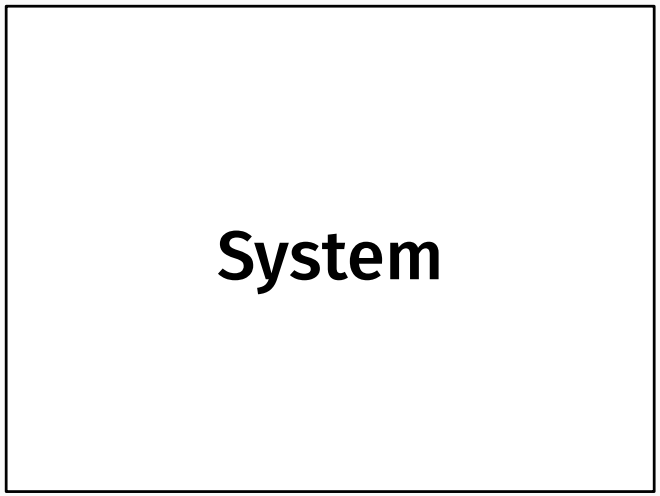
Leveraging DTrace for runtime verification

Carl Martin Rosenberg

June 7th, 2016

Department of Informatics, University of Oslo

Context: Runtime verification



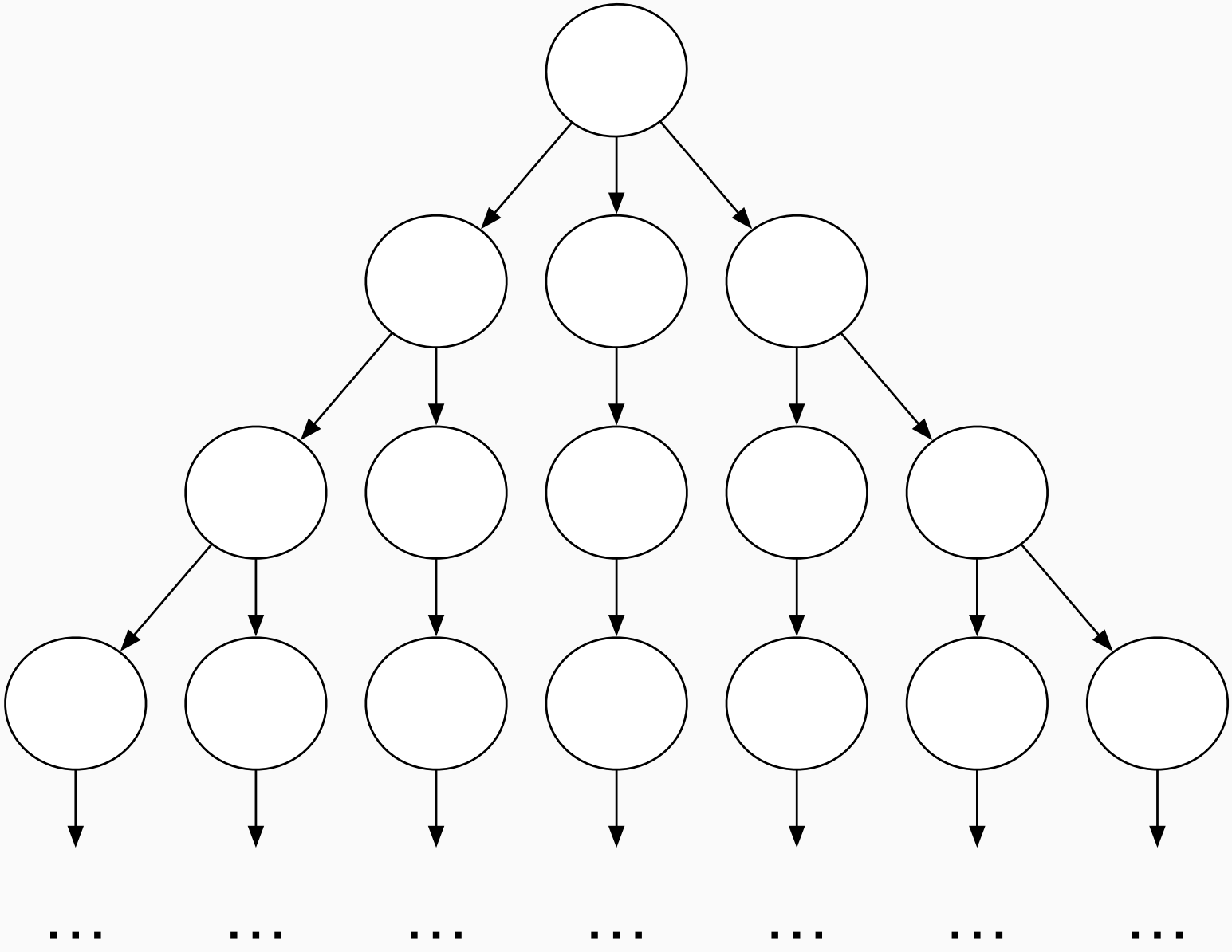
Desired properties

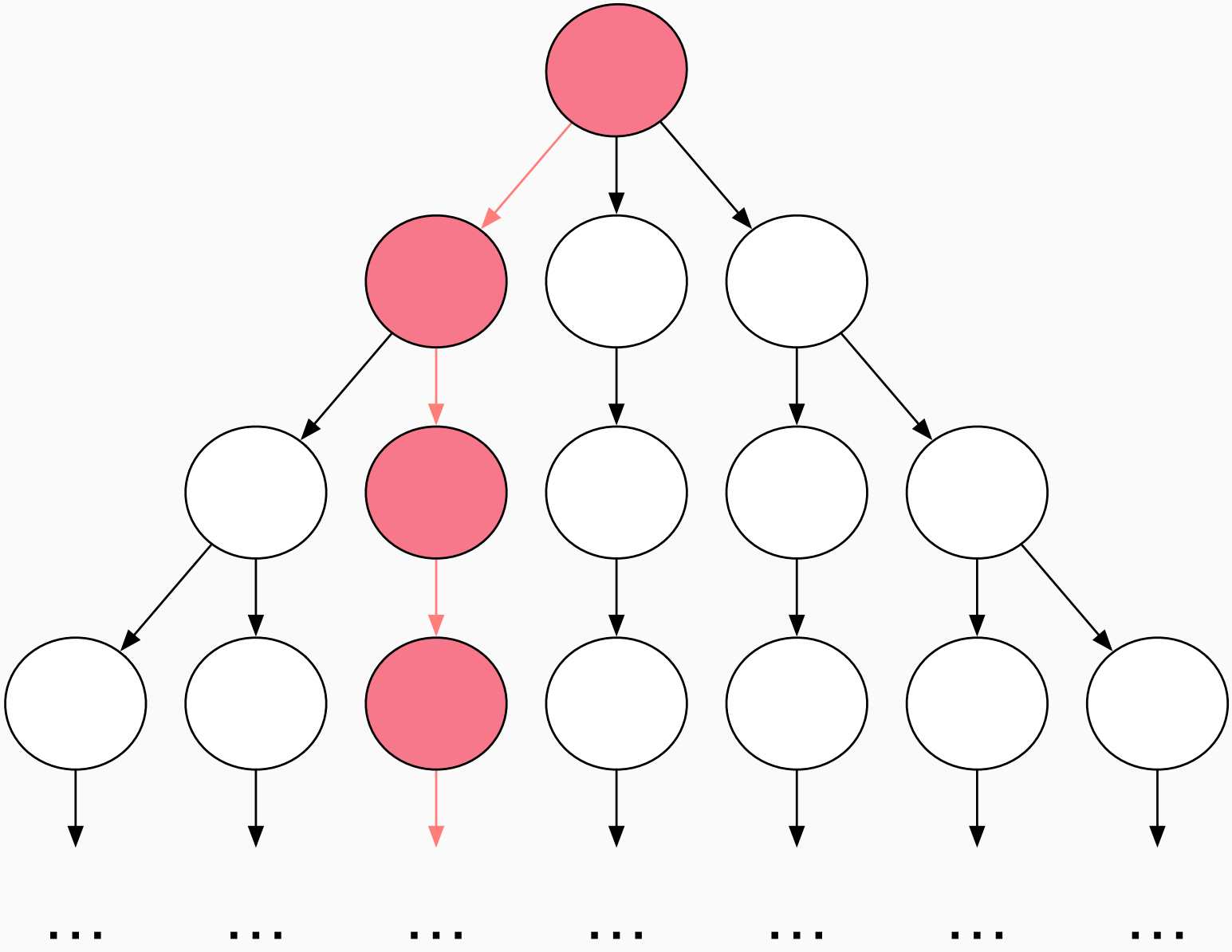
"Every request gets an answer"

"Buffers should never overflow"

"Variables should never enter an inconsistent state"

“Runtime verification is the discipline of computer science that deals with the study, development and application of those verification techniques that allow checking whether ***a run*** *of a system under scrutiny [...] satisfies or violates a given* ***correctness property****.” [6, p. 36, emphasis mine]*





How to do runtime verification

1. Rigorously specify a correctness property (typically using formal logic).
2. Collect runtime data from the system.
3. Find a way to automatically check if the collected data indicates that the correctness property is *violated* or *satisfied*.

Key idea: Exploit formal connections between logic and automata theory

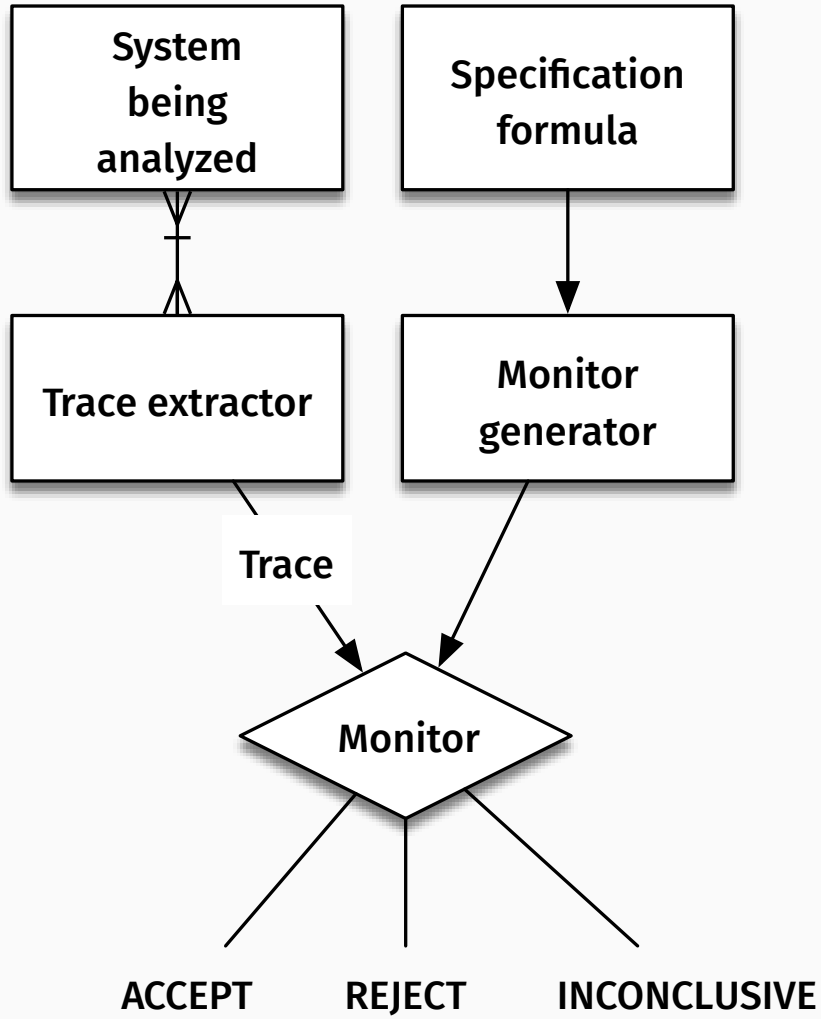
How to do runtime verification

1. There are ways of deriving automata from logical expressions that *accept* or *reject* their input based on whether the input satisfies or falsifies a logical formula.
2. This can be used to create *monitors*.

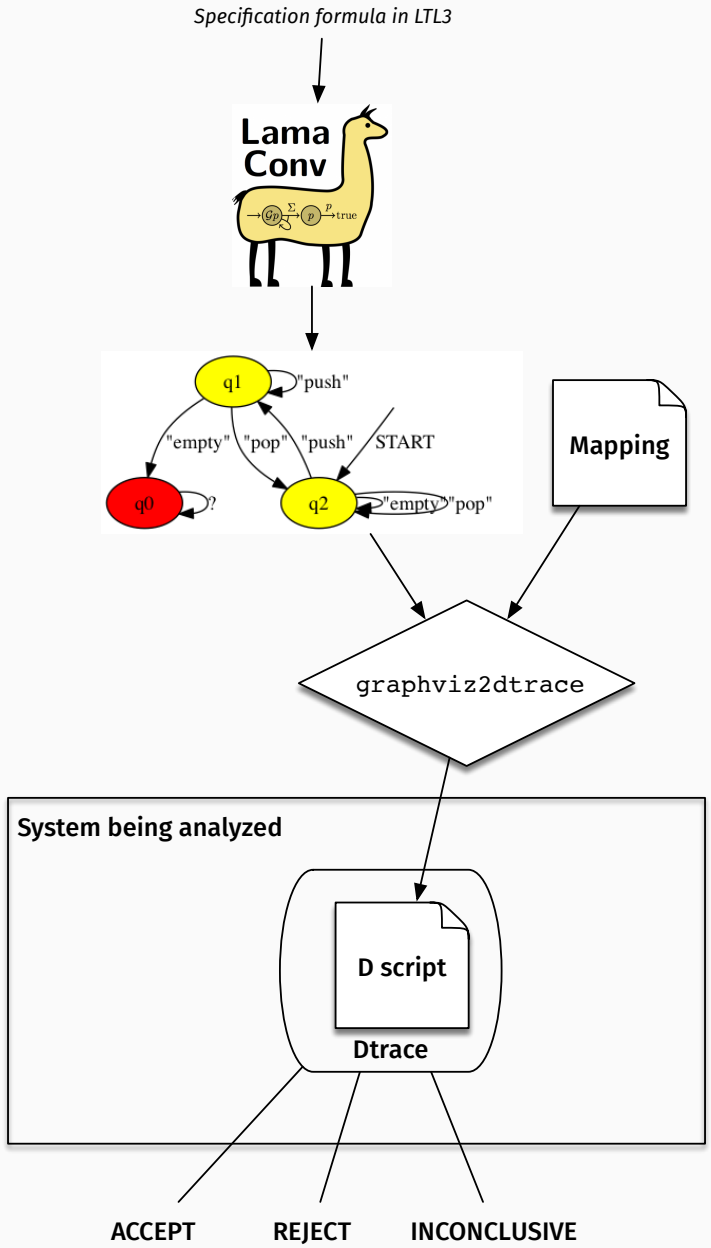
Monitors

A monitor is a program that consumes data from the system under scrutiny, interprets this data as a *run* of a system and reports whether the specified property is

- satisfied by the data,
- falsified by the data or
- that the data is insufficient to derive a verdict [7, p. 294–295].


System being analyzed
Specification formula
Trace extractor
Monitor generator
Trace
Monitor
ACCEPT
REJECT
INCONCLUSIVE

$$\Box((\text{push} \wedge \Diamond\text{empty}) \rightarrow (\neg\text{empty}\ U\ \text{pop}))$$



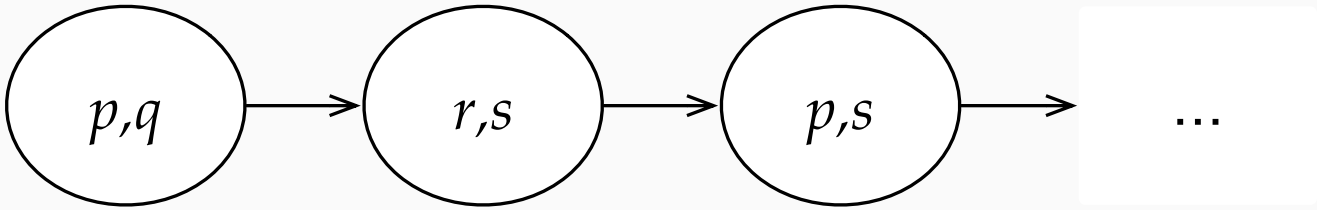
Specification formalism: LTL_3

LTL_3 is a three-valued variety of Linear Temporal Logic (LTL): Same *syntax*, different *semantics*.

LTL: Linear Temporal Logic

- In the context of formal methods, Temporal Logic is used to reason about the evolution of some system over time.
- LTL is based on the notion of time as an infinite sequence of time slices: Time is *linear*.
- The idea of using Linear Temporal Logic for program verification originated with Pnueli [8].

Linear time



LTL Syntax

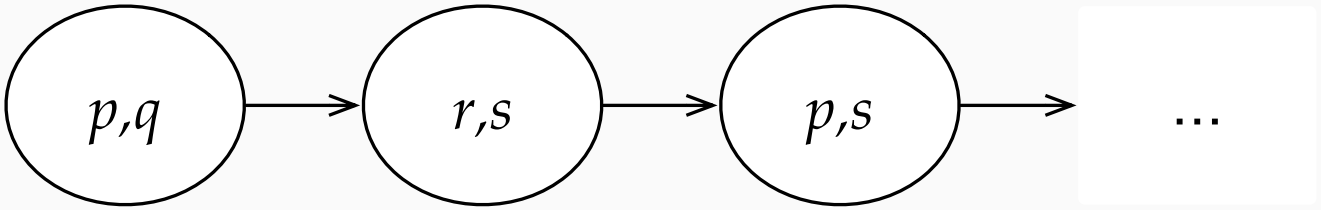
Syntactically, LTL extends the familiar Propositional Logic with temporal operators. The temporal operators are:

- $\Box$ (*always*),
- $\Diamond$ (*eventually*),
- $\bigcirc$ (*next*),
- U (*until*),
- R (*release*),
- W (*weak until/waiting for*).

LTL Semantics

- LTL formulas are interpreted on sequences of states, where each state contains a set of atomic propositions that are true in that state.
- In the standard view, these sequences of states are considered **infinite**, and we call such sequences of states *traces*.

LTL Semantics



Some observations:

- $\sigma \models p \, U \, s$,
- $\sigma \models \bigcirc s$,
- $\sigma \models \Diamond r$,
- $\sigma \not\models \Box p$ and
- $\sigma \models p \, W \, s$.

Key idea of LTL_3: A reasonable way of dealing with *finite* traces.

LTL_3 Semantics

- LTL_3 is defined for *finite* traces, which makes it suitable for runtime verification: We can only observere finite runs.

Key idea of LTL_3: Identify *good* and *bad* prefixes [5].

Good prefix

- A trace fragment u is a good prefix with respect to some property ϕ if ϕ holds in **all** possible futures following u.

Bad prefix

- A trace fragment u is a bad prefix with respect to some property ϕ if ϕ holds in **no** possible futures following u.

LTL_3 Semantics summarized

We can thus state the truth-value of an LTL_3 formula ϕ with respect to a finite trace u as follows:

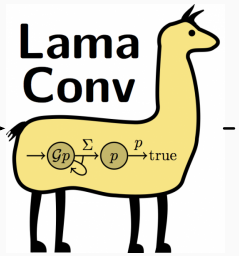
$$u \models_3 \phi = \begin{cases} \top & \text{if } u \text{ is a good prefix wrt. } \phi \\ \bot & \text{if } u \text{ is a bad prefix wrt. } \phi \\ ? & \text{otherwise.} \end{cases}$$

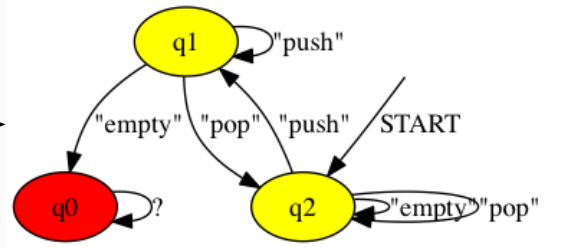
Foundational idea: For an LTL formula ϕ, a corresponding *Büchi automaton* [2] can be derived.

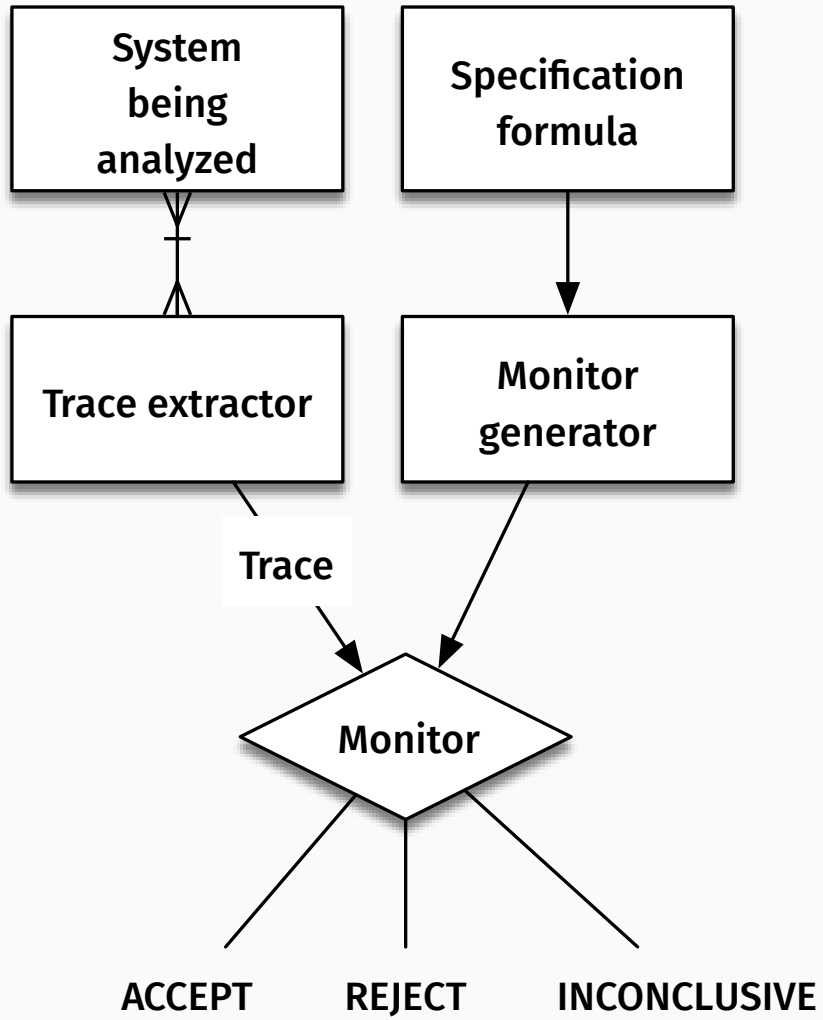
- Büchi Automata are defined on *infinite* traces and accept a trace if and only if the automaton visits some accepting state infinitely often.

- Bauer et al. give an algorithm for creating LTL_3-monitors [1, 14:10-14:13]
- This algorithm is implemented in `LamaConv` [9], which we make use of in the thesis.

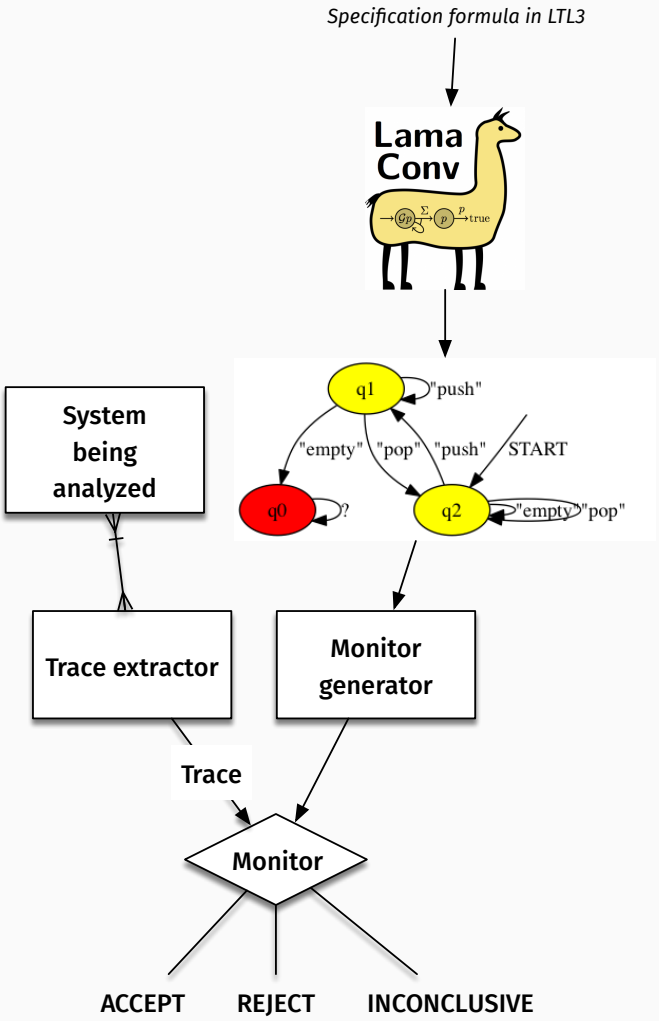
$$\square((\text{push} \wedge \Diamond\text{empty}) \rightarrow (\neg\text{empty}\ U\ \text{pop}))$$






System being analyzed
Specification formula
Trace extractor
Monitor generator
Trace
Monitor
ACCEPT
REJECT
INCONCLUSIVE

$$\square((\text{push} \wedge \Diamond\text{empty}) \rightarrow (\neg\text{empty}\, U\, \text{pop}))$$



DTrace

- DTrace is an operating system technology for monitoring running software systems.
- Originally written by Bryan Cantrill, Adam Leventhal and Mike Shapiro for the Sun Solaris 10 operating system, DTrace is now available for Mac OS X, FreeBSD and other systems [4]
- If a running system has DTrace installed, an administrative user can log into the system, write a DTrace script and get insights about the system without having to reboot, stop or alter the system in any way.

DTrace's two most compelling features

1. DTrace gives a *unified* view of the whole system: Events within the kernel and in userland processes can be analyzed simultaneously.
2. DTrace provides facilities for *dynamic tracing*: Instrumentation which does not rely on static artifacts in the source code.

Using DTrace: The D scripting language

- Users interact with the DTrace framework via a domain-specific AWK-like scripting language called D.
- D is an event-driven programming langauge, where users specify actions that DTrace should take when an event of interest occurs.

Using DTrace: The D scripting language

```
#!/usr/sbin/dtrace -qs
syscall::read:entry
/execname != "dtrace" /
{
    printf("%s\n", execname);
}
```

Main components of D

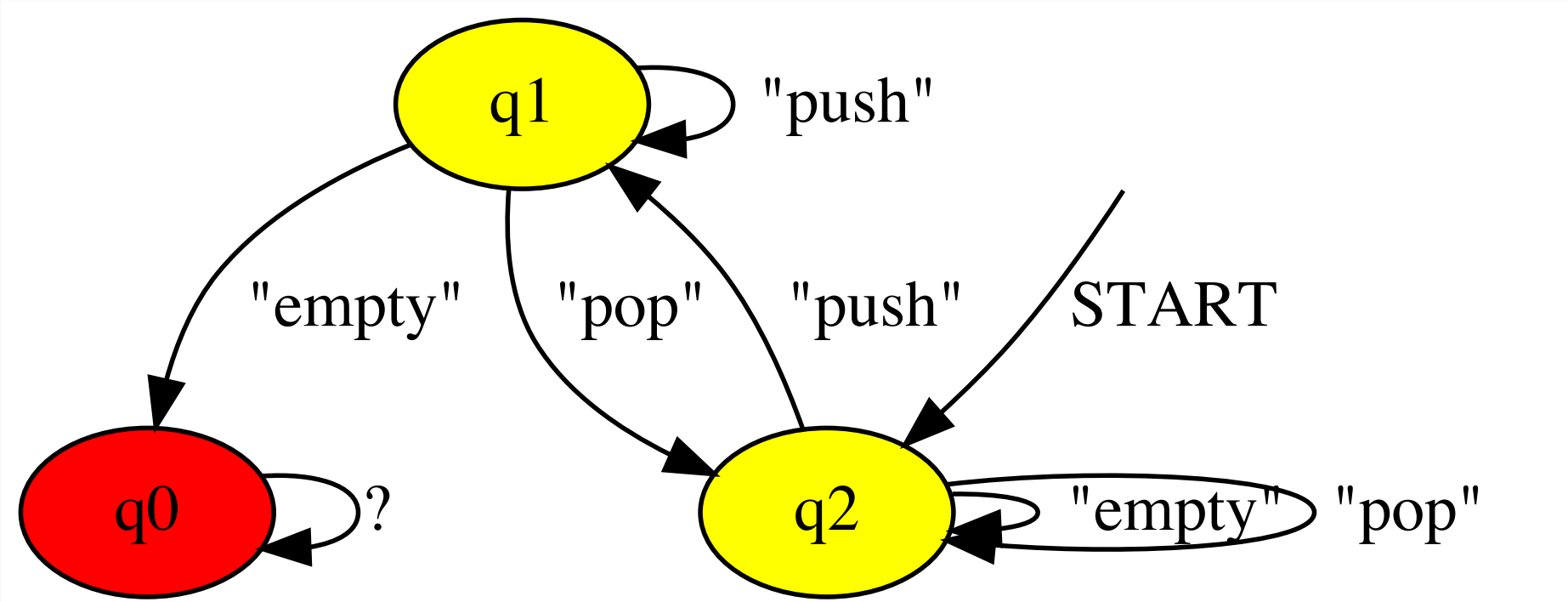
- Probes (4-tuples)
- Action blocks
- Predicates
- Probe clauses

DTrace Probes

- DTrace provides the user with an enormous list of possible instrumentation points representing events of interest. These instrumentation points are called *probes*.
- The available probes reflect aspects of the system that can be monitored at the current point in time.
- Probes are identified by a four-tuple `<provider:module:function:name>`.
- When the event a probe represents occurs, one says that the probe "fires").

Design and Implementation of `graphviz2dtrace`

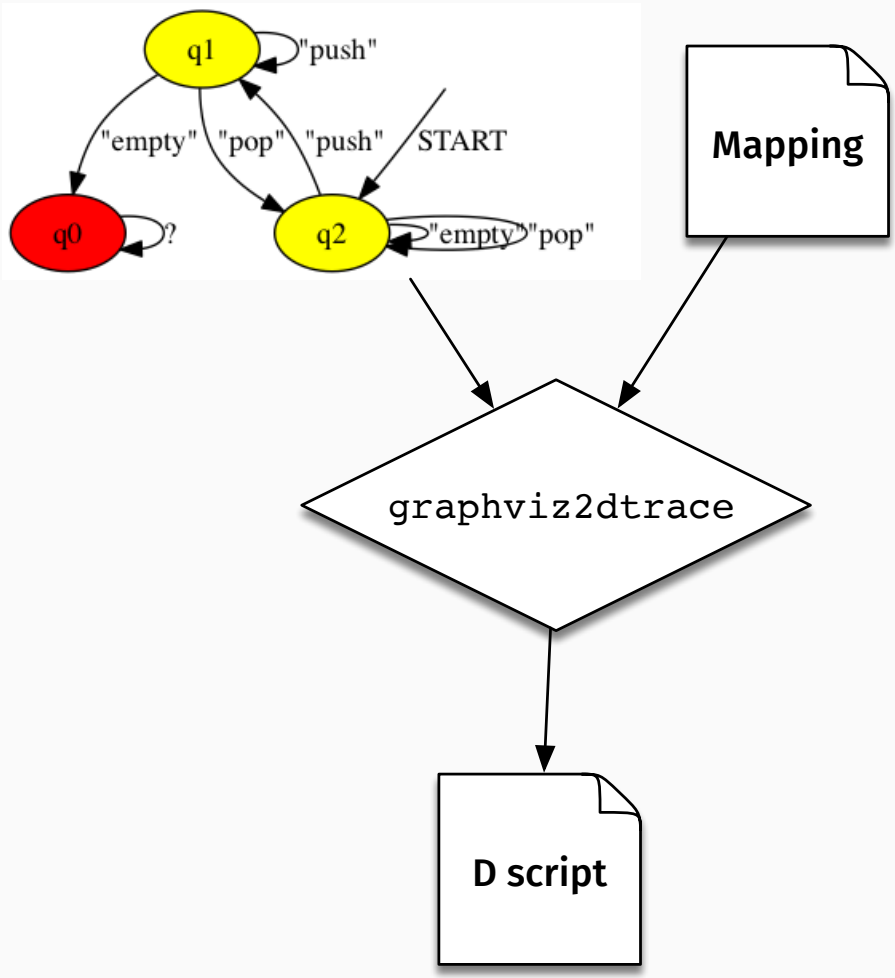
Basic idea: Associate atomic propositions in LTL specifications with DTrace probe specifications (with optional predicates).



push → `pid$target::push:entry`

pop → `pid$target::pop:return`

empty → `pid$target::empty:return/arg1 == 1/`


q1
"push"
"empty"
"pop"
"push"
START
q0
?
q2
"empty"
"pop"
Mapping
graphviz2dtrace
D script

graphviz2dtrace

- In essence, `graphviz2dtrace` is a source-to-source compiler which takes LTL_3-based automata and creates corresponding D scripts.
- The resulting scripts have the automaton's transition function encoded in an array, and the automaton state stored in a variable.
- When an event occurs, the state of the automaton is updated according to the transition function.

Anticipation

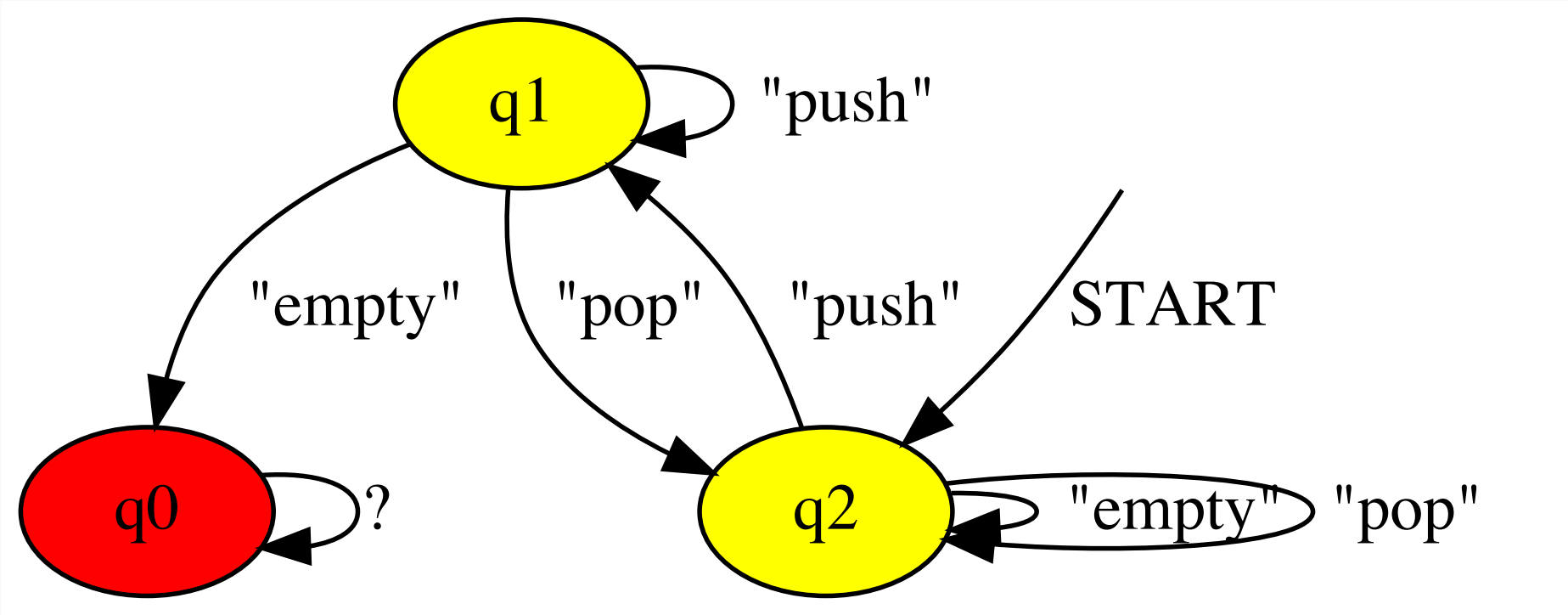
- `graphviz2dtrace` creates monitors that terminate immediately upon finding a good or bad prefix: They are *anticipatory*.
- The scripts achieve this by understanding which state it is *about* to enter.

Anticipation

```
pid$target::empty:return
/ (arg1 == 1) && (state == 1)/
{
        trace("REJECTED");
        HAS_VERDICT = 1;
        exit(0);
}
```

Implementation

- The essential implemenation concern is creating three types of probe clauses: *rejecting*, *accepting* and *neutral* clauses.
- Rejecting clauses are clauses dealing with situations where bad prefixes are found.
- Accepting clauses are clauses dealing with situations where good prefixes are found.
- Neutral clauses simply update the state of the automaton.

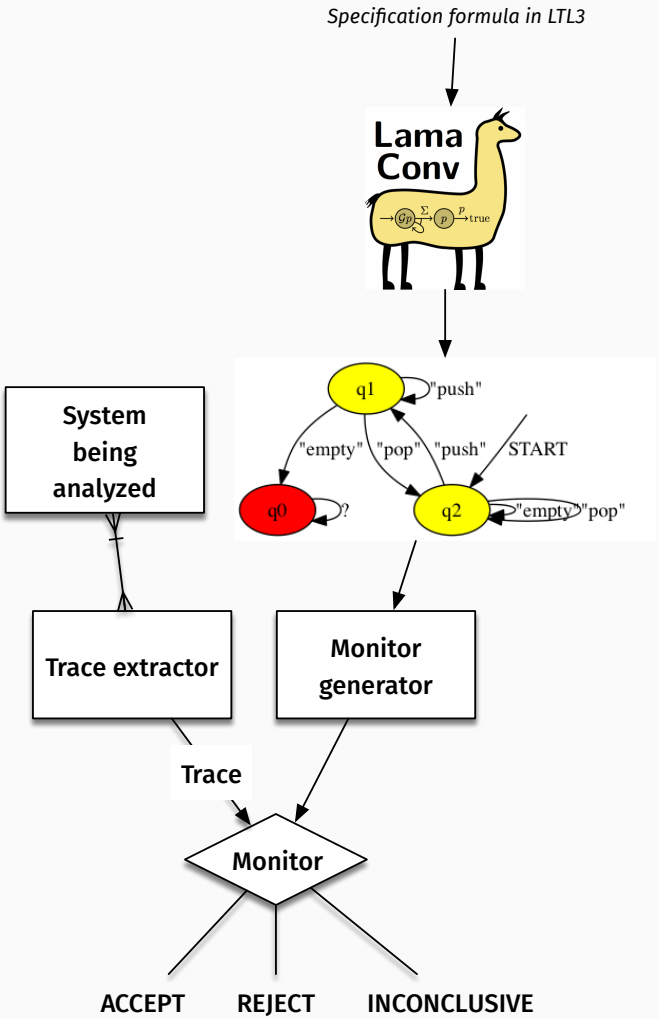

q1
"push"
"empty"
"pop"
"push"
START
q0
?
q2
"empty"
"pop"

Implementation

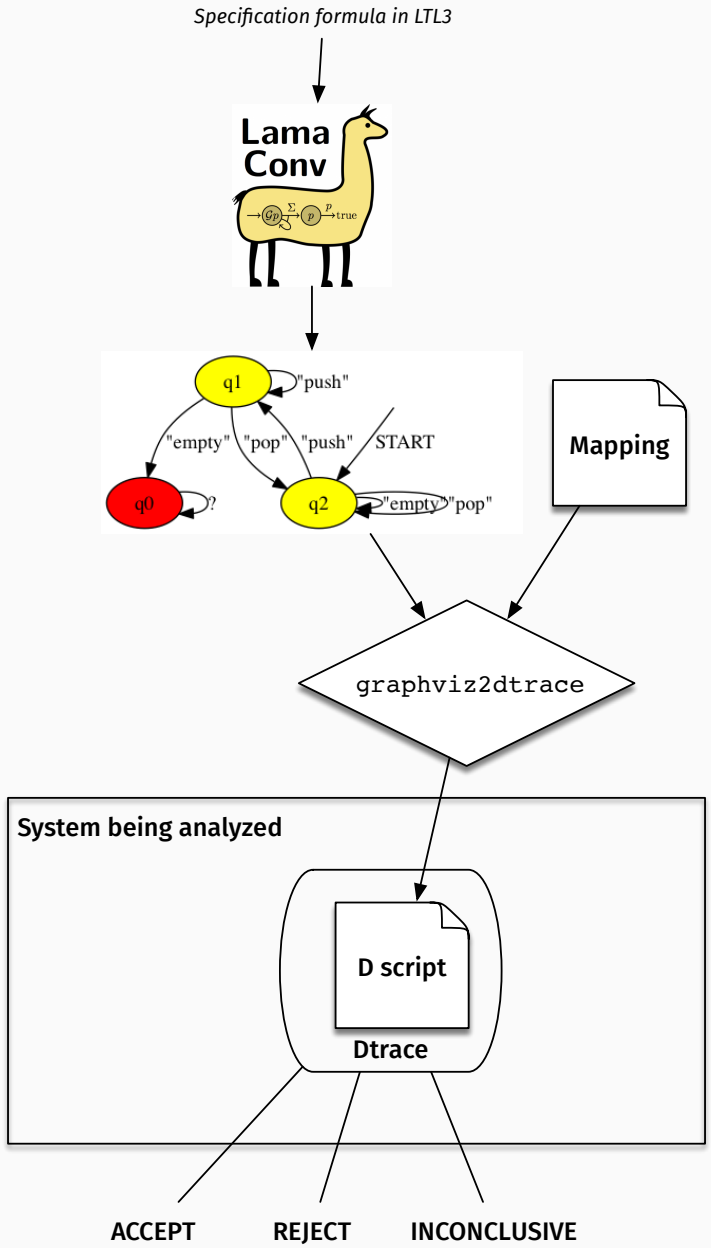
- Since all clauses use predicates to reason about the automaton states, and since probes are processed top to bottom, neutral clauses must be placed last in the script.

Key limitation: Race conditions occur if two neutral probe clauses fire simultaneously.

$$\square((\text{push} \wedge \Diamond\text{empty}) \rightarrow (\neg\text{empty}\, U\, \text{pop}))$$



$$\Box((\text{push} \wedge \Diamond\text{empty}) \rightarrow (\neg\text{empty}\ U\ \text{pop}))$$



Case study 1: The stack

Demo time!

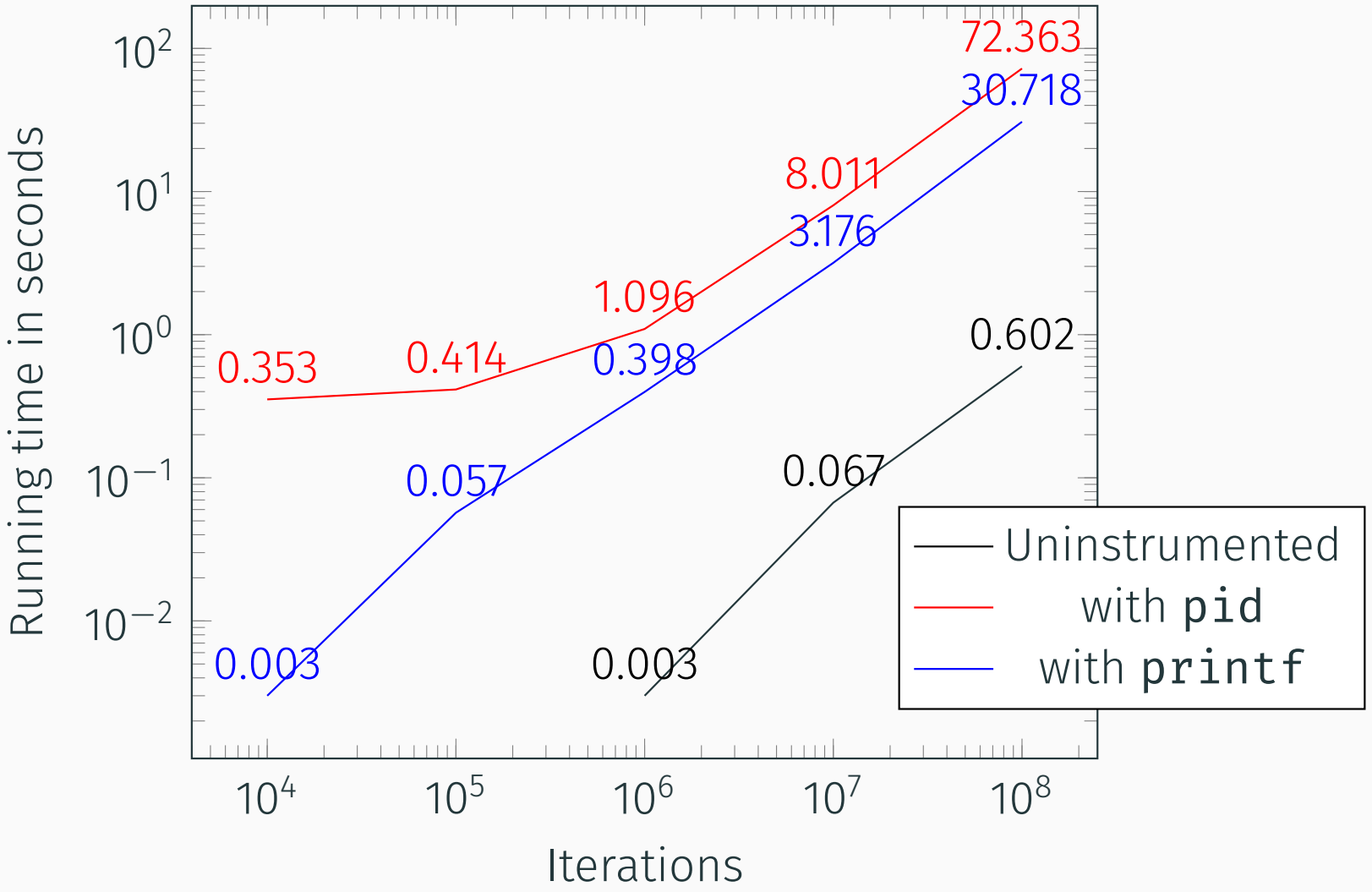
`https://vimeo.com/169470067` (03:00)

Gregg's dictum

Brendan Gregg [10]

- "Don't worry too much about pid provider probe cost at < 1000 events/sec."
- "At > 10,000 events/sec, pid provider probe cost will be noticeable."
- "At > 100,000 events/sec, pid provider probe cost may be painful." [3]

Monitor overhead[1]



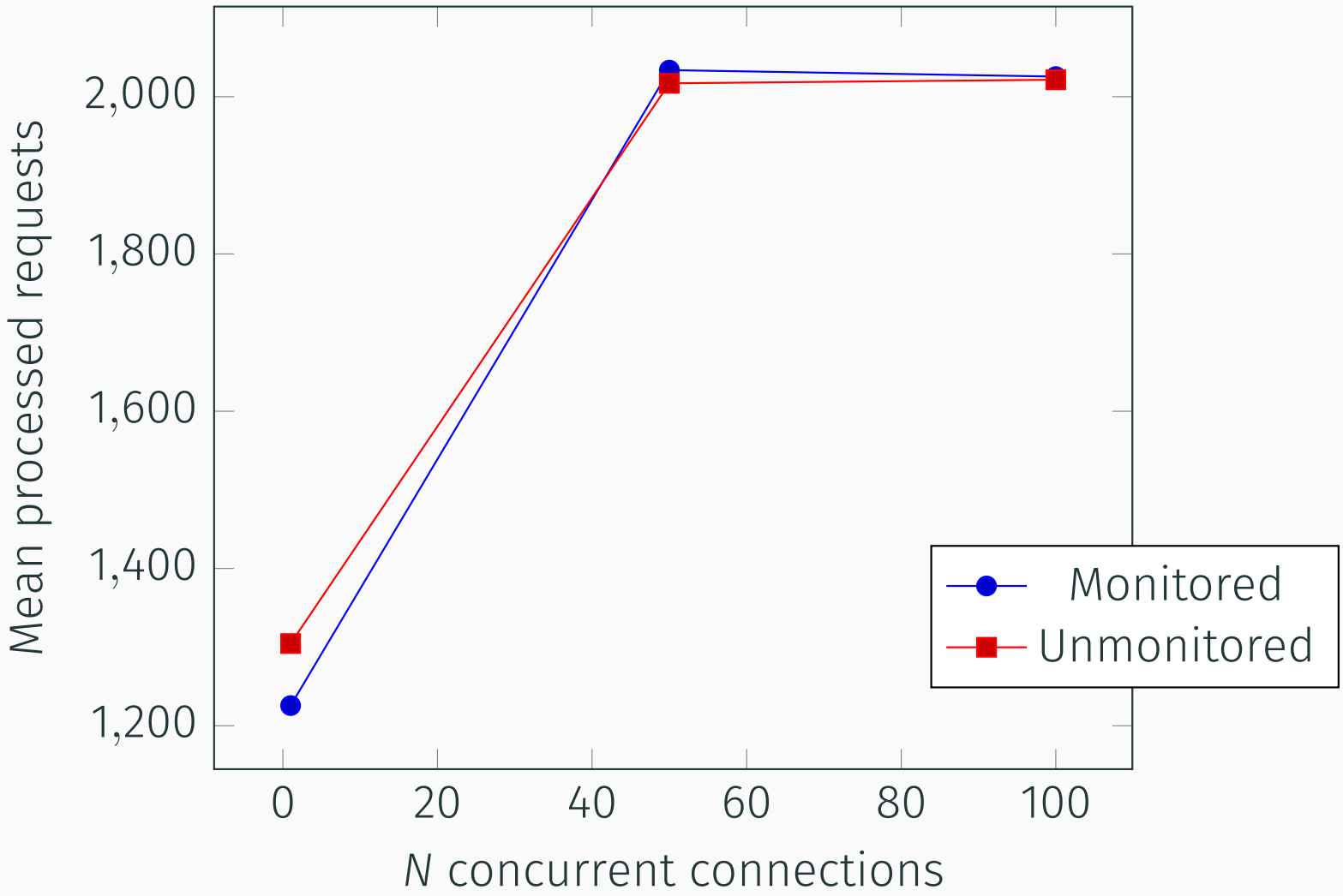
[1]Averaged, measured with `time`, largest of `real` or `user+sys`

Case study 2: Node.js and PostgreSQL

Demo time!

`https://vimeo.com/169585739` (02:55)

Mean processed requests per second at various concurrency levels[2]



[2] Averaged, measured with **ab**

Conclusion and Evaluation

Questions?

References I


A. Bauer, M. Leucker, and C. Schallhart.
Runtime verification for ltl and tltl.
ACM Trans. Softw. Eng. Methodol., 20(4):14:1–14:64, Sept. 2011.


J. R. Büchi.
On a decision method in restricted second order arithmetic.
In *Proc. Internat. Congr. Logic, Method. and Philos. Sci*, pages 1–12, 1960.


B. Gregg.
DTrace pid Provider Overhead.
`http://dtrace.org/blogs/brendan/2011/02/18/dtrace-pid-provider-overhead/`, 2011.

References II


B. Gregg and J. Mauro.
DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD.
Prentice Hall Professional, 2011.


O. Kupferman and M. Y. Vardi.
Model checking of safety properties.
Formal Methods in System Design, 19(3):291–314, 2001.


M. Leucker.
Teaching runtime verification.
In S. Khurshid and K. Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 34–48. Springer Berlin Heidelberg, 2012.

References III

M. Leucker and C. Schallhart.

A brief account of runtime verification.

The Journal of Logic and Algebraic Programming, 78(5):293 – 303, 2009.

The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

A. Pnueli.

The temporal logic of programs.

In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

T. Scheffel and M. S. et al.

Lamaconv—logics and automata converter library.

`http://www.isp.uni-luebeck.de/lamaconv`.

References IV

D. Straughan.
Brendan Gregg speaking at ZFS Day, Oct 2, 2012, San Francisco., 2012.
(Own work) [CC BY-SA 3.0], via Wikimedia Commons.