Leveraging DTrace for runtime verification

Carl Martin Rosenberg [1] Martin Steffen [2] Volker Stolz [2,3]

September 28, 2016

[1]Simula Research Laboratory

[2]Inst. for Informatikk, Universitetet i Oslo

[3]Inst. for Data- og Realfag, Høgskolen i Bergen
Norway

Context: Runtime Verification

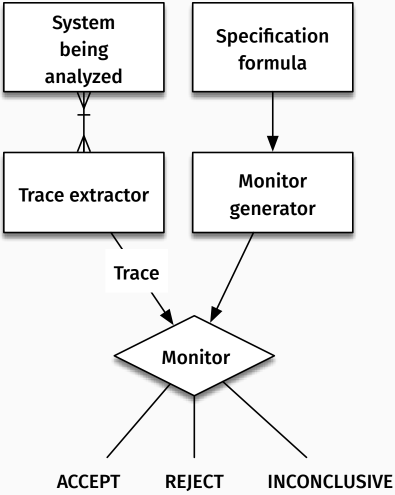
System

Desired properties

“Every request gets an answer”

“Buffers should never overflow”

“Variables should never enter an inconsistent state”


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

Overview

- Goal: Evaluate DTrace's suitability for RV.
- Contribution: `graphviz2dtrace`, a monitor synthesis tool.
- We evaluate the tool on two case studies.

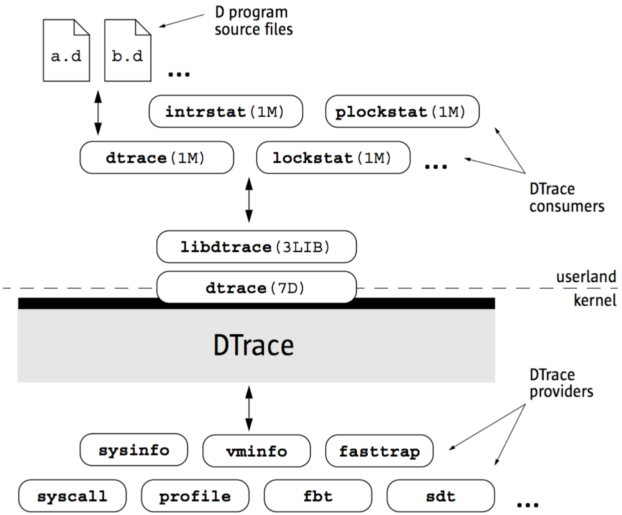
DTrace

- DTrace is a system-wide instrumentation framework.
- Originally written for the Sun Solaris 10 operating system, now available for for Mac OS X, FreeBSD and other systems [Gregg and Mauro, 2011].

DTrace’s two most compelling features

1. DTrace provides facilities for *dynamic tracing*.
2. DTrace gives a *unified* view of the whole system.

DTrace Architecture



From *Solaris Dynamic Tracing Guide*, page 28

Static and Dynamic Instrumentation

- DTrace allows for both static and dynamic instrumentation.
- Dynamic providers: `pid` and `fbt`.
- All other providers rely on static instrumentation artefacts.

Static and Dynamic Instrumentation

- Developers can add their own instrumentation points.
- Many prominent projects have static instrumentation points: PostgreSQL, Node.js, Apache, CPython etc.

Using DTrace: The D scripting language

- Users interact with DTrace via *D*, a DSL.
- 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 /* probe */
/execname != "dtrace" / /* predicate */
{
    printf("%s\n", execname);
} /* action block */
```

D has all the right building blocks for encoding Finite State Automata.

Design and Implementation of `graphviz2dtrace`

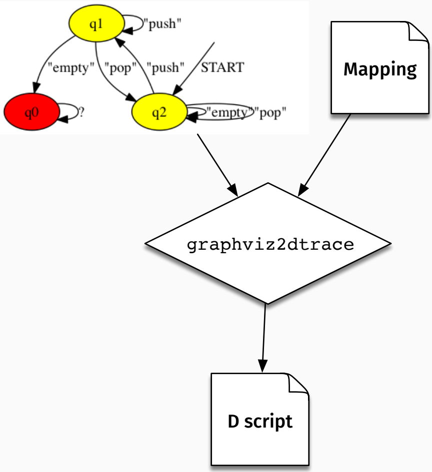
Basic idea 1: Associate atomic propositions in LTL specifications with DTrace probes.

push $\rightarrow$ `pid$target::push:entry`

pop $\rightarrow$ `pid$target::pop:return`

empty $\rightarrow$ `pid$target::empty:return/arg1 == 1/`

Basic idea 2: Use standard techniques to create automata from specification formulas, and encode automata in D.


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

Specification formalism: LTL_3

- LTL_3[Bauer et al., 2006] gives a reasonable way of dealing with *finite* traces.
- LTL_3 is a three-valued variety of Linear Temporal Logic (LTL): Same *syntax*, different *semantics*.
- Key idea of LTL_3: Identify *good* and *bad* prefixes [Kupferman and Vardi, 2001].

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:

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

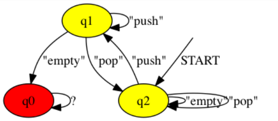
Creating automata: LamaConv

- Bauer et al. give an algorithm for creating LTL_3-monitors [Bauer et al., 2011, 14:10-14:13]
- This algorithm is implemented in `LamaConv`[1], which we make use of.

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

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

Lama Conv



`graphviz2dtrace`

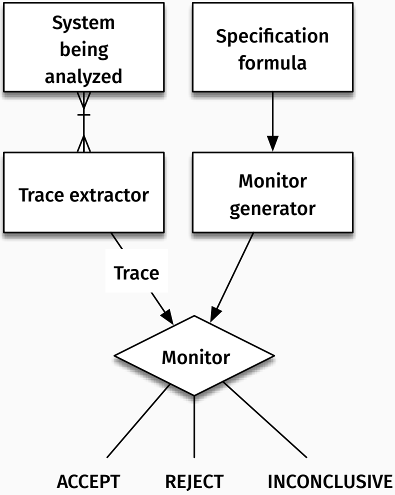
- In essence, `graphviz2dtrace` is compiles from LTL_3-based automata to D scripts.
- The automaton's transition function is encoded in an array, and the state is stored in a variable.
- When an event occurs, the state of the automaton is updated according to the transition function.

Anticipation

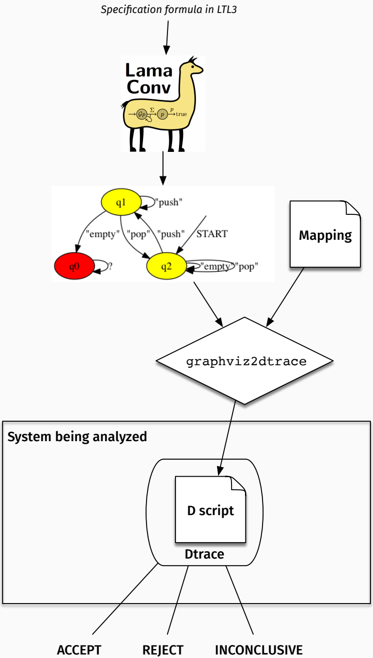
- `graphviz2dtrace` creates *anticipatory* monitors that terminate immediately upon finding a good or bad prefix.
- The scripts achieve this by understanding which state it is *about* to enter.

Anticipation

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


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

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

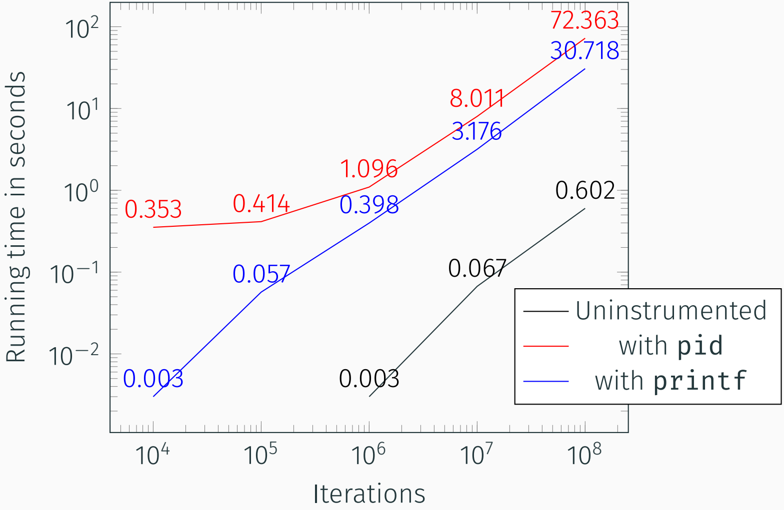


Evaluation

Case Studies

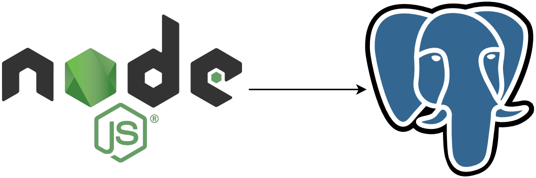
1. We dynamically instrument a faulty stack implementation written in C.
2. We investigate a Node.js web server interacting with a PostgreSQL database.

Monitor overhead in Case 1[2]



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

Case 2



Case 2

We want the following properties to hold:

1. The server should never send a response before the corresponding database query is complete.
2. There should never be an HTTP request for which the corresponding database query and HTTP response never happen.

Hack: Use counters to keep track of queries

Case 2

The server should never send a response before the corresponding database query is complete:

Approximation: Number of sent responses should never exceed number of queries:

$$\Box\neg(\text{nresponses} > \text{nqueries})$$

Case 2

There should never be an HTTP request for which the corresponding database query and HTTP response never happen:

Approximation: There should never be more than 100 pending requests:

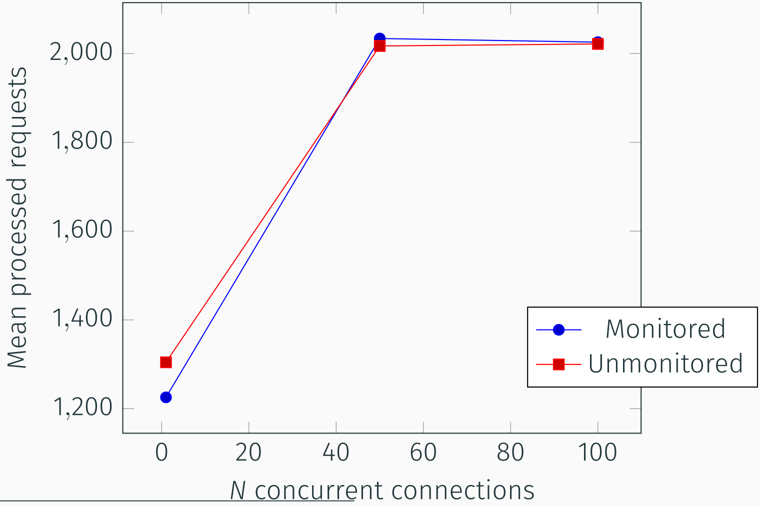
$$\Box\neg(((\text{nrequests} - \text{nresponses}) > 100) \land ((\text{nrequests} - \text{nqueries}) > 100))$$

Case 2: Results

1. Monitor with counters detect violations of both properties.
2. Screencast: `https://vimeo.com/169585739`

Case 2: Performance Evaluation

Mean processed requests per second at various concurrency levels[3]



[3] Averaged, measured with `ab`

Gregg's dictum

Brendan Gregg [Straughan, 2012]

- "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." [Gregg, 2011]

Future Work

- Separate trace-generation from verification: Collect data with DTrace, evaluate with external process.
- Investigate mapping *predicates* rather than *probes*.
- Steering systems can be created by using the `system` function.

Concluding remarks

- Monitoring overhead is negligible when probe firings are below 10 000 per second.
- `graphviz2dtrace` enables cross-process monitoring.
- `graphviz2dtrace`-generated scripts *are* susceptible to race conditions if probe firings may overlap.

References I

Bauer, A., Leucker, M., and Schallhart, C. (2006).
FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science: 26th International Conference, Kolkata, India, December 13-15, 2006. Proceedings, chapter Monitoring of Real-Time Properties, pages 260–272.
Springer Berlin Heidelberg, Berlin, Heidelberg.

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

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

References II

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

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

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