Leveraging DTrace for Runtime Verification[⋆]

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

[1] Inst. for Informatikk, Universitetet i Oslo
[2] Inst. for Data- og Realfag, Høgskolen i Bergen
Norway

Abstract. DTrace, short for "dynamic tracing", is a powerful diagnostic tool and tracing framework. It is invaluable for performance monitoring, tuning, and for getting insights into almost any aspect of a running system. In this paper we investigate how we can leverage the DTrace operating system-level instrumentation framework [9] to conduct runtime verification. To this end, we develop `graphviz2dtrace`, a tool for producing monitor scripts in DTrace's domain-specific scripting language D for specification formulas written in LTL_3, a three-valued variety of the well-known Linear Temporal Logic. We evaluate the tool by analyzing a small stack-implementation and a multi-process system.

1 Introduction

Runtime verification is an emergent field of research in which formal properties of concrete program or system runs are checked in an automatic manner. In order to conduct runtime verification, one must extract relevant information from the running system without harming or degrading the system in the process. We investigate using the DTrace [9] framework for this purpose.

Originally developed for Sun Microsystems, DTrace combines both static and dynamic instrumentation techniques in a unified framework spanning all aspects of a software system, from specific events in userland processes to function calls within the operating system kernel. DTrace exposes instrumentation points representing events of interest, and lets users associate actions that the computer should take when the selected events occur via a domain-specific, AWK-like, programming language, D. We investigate the suitability of DTrace for runtime verification by making the following contributions:

1. We design and implement `graphviz2dtrace`, a tool for generating DTrace-based monitors for properties specified in LTL_3: a three-valued variety of the well-known specification logic Linear Temporal Logic (LTL) [5]. In conjunction with the `LamaConv` automata library [22], `graphviz2dtrace` provides a complete runtime verification platform.

[⋆] This article is based upon work from COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology).

2. We use `graphviz2dtrace`-based monitors to verify two software systems: A simple stack implementation written in C, and a web application consisting of a Node.js [17] web server communicating with a PostgreSQL [19] database. We demonstrate how `graphviz2dtrace`-based monitors can be used to detect property violations and analyze the performance penalty we induce by monitoring the running system.
3. Drawing on the two case studies, we discuss the possibilities and inherent limitations of `graphviz2dtrace`-based monitoring, and suggest directions for future work using DTrace for runtime verification.

The paper is organized as follows. In Section 2, we describe the main components of DTrace: probes, providers, and the D scripting language. We also discuss how dynamic instrumentation is possible with the `pid` and `fbt` providers. Then, we describe how to create a bridge between logical and practical concepts by associating atomic LTL propositions with DTrace *probe specifications*, and how this idea is implemented in `graphviz2dtrace`. Since `graphviz2dtrace` produces standalone scripts in the D programming language, we discuss how `graphviz2dtrace` is constrained by the inherent limitations of D, especially with respect to *concurrency.*

We describe the process of finding and specifying observable events, associating the events to atomic propositions in LTL specification formulas, and using the generated monitors to detect property violations.

We evaluate the tool in two case studies: First, we investigate a faulty stack implementation written in C, demonstrating how we can instrument a program without leaving static artifacts in the source code. Then, we investigate a system composed of a web server written in Node.js and a PostgreSQL database. We specify a safety property concerning the interaction between the web server and the database and demonstrate how to detect a violation by hooking a monitor onto the running processes. We also analyze the performance degradation we induce through monitoring, before evaluating our findings and drawing our conclusions.

Section 5 concludes with related and future work. An extended version and technical annexes can be found in the recently published Master thesis [21] and the accompanying web-page.[3]

2 DTrace

DTrace, short for “dynamic tracing”, is a powerful operating system level diagnostic tool and tracing framework. It can be seen as a major step forward from older tools such as `ptrace` or `strace` in terms of versatility, sophistication, and efficiency. It offers a flexible tool set for performance monitoring, tuning and collecting comprehensive information on the behavior of a running system, from the behavior of a single process to the internals of the operating system kernel.

[3] `http://www.mn.uio.no/ifi/english/research/groups/pma/completedmasters/2016/rosenberg/`

In its most basic form, it gives users a way of specifying events of interest and associate actions that the computer should take when those events occur. With DTrace, a user can make requests like

– *whenever a process opens this file, increment this counter and notify me when the counter exceeds a hundred*, or even something as complex as
– *whenever the Apache web server processes an HTTP request, store the response code in a data structure, and when I say so, show me a statistical distribution of the response codes.*

Requests like these are programmed in a domain-specific scripting language, D, which is heavily inspired by AWK and C. Originally written for the Sun Solaris 10 operating system, DTrace is now available for Mac OS X, FreeBSD, and other systems [13]. With 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 has two main concerns: Firstly, to give users a way of specifying the information they want, and secondly, to acquire the requested information in a safe and efficient manner. While both concerns ultimately must be met, they are treated separately within DTrace: *Producers* are DTrace components that acquire the requested data. Other components post-process the acquired data, presenting it to the user in the manner the user requested: These components are called *consumers*. One purpose of this separation is to ensure safety: producers should only be concerned with acquiring data in a safe and unintrusive way, not with how the acquired data is to be presented or used [8, p. 30-32].

At the kernel level, there are a series of producer components called *providers* that gather data about some aspect of the running system. For example, the `syscall` provider gives data about system calls that are issued to the operating system. The most important consumer is the `dtrace` program, the command-line utility that provides the most common way of interacting with the DTrace framework. This component compiles and executes D-scripts, and calls upon the underlying producers to acquire the requested data.

Specifying events of interest: Probes. First of all, users need a way to specify events of interest. To this end, 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>`. Users use these tuples to select the probes they are interested in, and specify actions to be taken once the associated events occur. In DTrace parlance, when the event a probe represents occurs, one says that the probe “fires”.

It is also possible for application developers to employ so called User Statically Defined Tracing (USDT) to their own programs, by inserting static probes in the application source code. In this way, the application developers can create custom providers for their applications. Many notable software projects have

USDT probes, including the PostgreSQL database management software, that we will visit in our case study later, as well as many programming language runtimes.

Doing things when events occur: Actions. Once users have specified which probes they are interested in, they can associate actions blocks that should be executed when the selected probes fire. Users can store data in variables, collect statistics, spawn other processes, inspect system structure and analyze function parameters, to name just a few of the possibilities. Even though action statements are specified in blocks tying them to specific probes, it is possible to share variables and data structures between action blocks, making it possible to monitor complex interactions between events [13, p. 37–42]. The available action statements will vary between DTrace implementations on different platforms.

Filtering out the noise: Predicates. When a probe fires, an optional predicate determines if the corresponding action block should execute or not. Predicates are written as boolean expressions that can use any D operator and any D data object. A missing predicate is equivalent to the predicate `/true/`, meaning that no filter is present and the action block will be executed unconditionally when the probe fires.

Dynamic Tracing. A foundational concept in DTrace is *dynamic tracing*. Dynamic tracing permits users to instrument programs on the fly, without requiring static artifacts to be present in the software that is being instrumented [8, p. 30]. This makes it possible to analyze systems that provide limited logging capabilities, systems that are distributed in binary form only, and systems that are opaque in other ways.

In DTrace, dynamic tracing is made possible by the `fbt` and `pid` providers [12]. The previously mentioned `fbt` provider makes it possible to instrument all function return values and arguments in the operating system kernel [13, p. 163]. For userland processes, the `pid` provider gives probes that fire when a function is entered or returned from, and can also be used to create probe firings for specific instructions in the function [13, p. 788-791]. In Section 4.1, we use the `pid` provider to dynamically instrument a stack program in C.

The listing in Figure 1 shows a simple D-script which matches on `read`-syscalls into the kernel. It prints the name of the process issuing the call, except of any running instance of the `dtrace`-process itself.

```
syscall::read:entry              /* probe */
/execname != "dtrace"/           /* predicate */
{ printf("%s\n", execname); }    /* action block */
```

Fig. 1. A simple D-script

3 Design of `graphviz2dtrace`

The fundamental idea behind `graphviz2dtrace` is to let users associate the atomic propositions in LTL formulas with DTrace-observable events represented by DTrace probes (with optional predicates). Suppose, for example, that we want to ensure that a program deallocates all memory before exiting, and have expressed this property as $\neg exit\ W\ \ dealloc$ using the precedence pattern identified by Dwyer et al. in [1]. Suppose that we also have produced a corresponding LTL_3 automaton with `LamaConv`. With `graphviz2dtrace`, we can create a concrete monitor program for this property in the following manner: First, we can map *exit* to the DTrace probe `pid$target::main:return`, which fires whenever the `main` function returns. Similarly, we can map *dealloc* to `pid$::dealloc:entry`, which fires whenever the `dealloc` function is entered. Every time DTrace registers one of the specified events, the state of the automaton is updated according to the automaton transition function (encoded in a two-dimensional array). The monitor reports a verdict the moment it detects that the property is either satisfied or violated and terminates itself.

Originally, the idea (and hence the name) of `graphviz2dtrace` was to provide a unified way of producing DTrace monitors from any monitor automaton encoded in Graphviz [11] dot notation[4]. However, we chose to restrict ourselves to monitor automata for LTL_3 since it is well suited for reasoning about *finite* traces.

LTL_3 differs from traditional LTL in its semantics, which is defined for *finite prefixes* of infinite traces. The semantics of LTL_3 is based on the notion of *good and bad prefixes* originally developed in [15]: A good prefix for a formula ϕ is a prefix such that all possible continuations of the prefix make ϕ true. Conversely, a bad prefix for ϕ is a prefix such that all possible continuations of the prefix make ϕ false. Consequently, an LTL_3 monitor is an automaton that *accepts* a trace if it detects a good prefix, *rejects* a trace if it detects a bad prefix or outputs *inconclusive* if the provided trace is neither a good nor a bad prefix [6].

In LTL_3 monitor automata produced by `LamaConv`, all states will be labeled either green, red or yellow. Whenever the automaton enters a red state, the automaton has detected a *bad prefix.* If the automaton enters a green state, a *good prefix* has been found. If the trace (ie. the input to the automaton) is terminated while the automaton is in a yellow state, the verdict is inconclusive. In `graphviz2dtrace`-produced scripts, this is reflected in three types of probe clauses: As soon as the automaton detects that it is *about* to enter an accepting or rejecting state, the script outputs the corresponding verdict and stops itself. If the script is terminated while in a yellow state, the script outputs `INCONCLUSIVE`.

Concurrency-related limitations The most important *limitation* with DTrace is that *there is no way to have a globally accessible yet synchronized state variable in D*: This introduces the possibility of race conditions if two or more probe

[4] The graphviz dot notation was chosen because `LamaConv` can produce it, for its ubiquity, and for the ease with which automata can be visualized.

clauses attempt to update the state variable of the automaton at the same time. A possible mitigation would be to use a thread-local rather than a global state variable, but that would make it impossible to reason about probes that are not associated with the same thread. We elected to make `graphviz2dtrace` agnostic about the provided probes: Users are responsible for preventing race conditions. While this severely restricts the properties that one can safely monitor with `graphviz2dtrace`-based scripts, we show an example that works around this limitation in section 4.

4 Case studies

In this section we use `graphviz2dtrace` to analyze simple properties in two different setups: in the first case study, we observe function calls in a simple C program that implements a stack-API. In the second, we show how our DTrace-based approach can be used to cover properties that span different operating-system level processes. Lastly, we discuss the performance penalties incurred through DTrace.

4.1 Verifying a single process program

To demonstrate `graphviz2dtrace` in practice, we start by investigating a naïve implementation of the classic stack data structure, supporting the operations *push*, *pop* and *empty*. The *push* function adds an element to the top of the stack, *pop* removes the topmost element on the stack and returns the element to the user, and *empty* says whether the stack is empty or not. We will consider the following property:

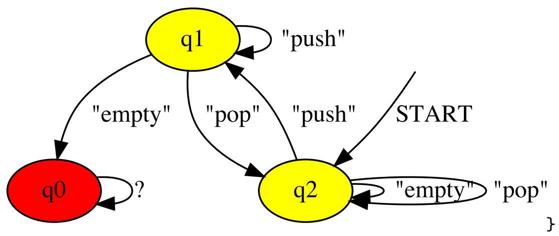
$$\Box((\text{push} \wedge \Diamond\text{empty}) \rightarrow (\neg\text{empty}\; U\; \text{pop}))$$

This property is chosen among the properties which Bauer et al. determined to be LTL_3-monitorable [6] and can be understood as saying that *for any stack that has been pushed to and is eventually found empty, a pop event must have occurred before the empty event.*

Obtaining the automaton. First, we must obtain an automaton by using `LamaConv`. We use the following invocation to generate an automaton encoded in the Graphviz dot language:

```
rltlconv "LTL=[]((push && <>empty) -> (!empty U pop)),
  ALPHABET=[push,pop,empty]" --formula --moore --min --dot
```

The resulting automaton and corresponding dot code are shown in Figure 2. We observe that the resulting automaton has two yellow states and one red state. If the input to the automaton ends while the automaton is in any of the yellow states, the verdict is *inconclusive*. If the automaton is in the red state, it means that it has detected a violation of the property.



```
digraph G {
  q0 -> q0 [label="?"];
  q1 -> q2 [label="\"pop\""];
  q1 -> q1 [label="\"push\""];
  q1 -> q0 [label="\"empty\""];
  q2 -> q2 [label="\"empty\""];
  q2 -> q2 [label="\"pop\""];
  q2 -> q1 [label="\"push\""];
  start [shape=none, style=invis];
  start -> q2 [label="START"];
  q2 [style=filled, fillcolor=yellow]
  q1 [style=filled, fillcolor=yellow]
  q0 [style=filled, fillcolor=red]
}
```

Fig. 2. Automaton (left) and dot script (right) for the formula $\Box((\text{push} \wedge \Diamond\text{empty}) \rightarrow (\neg\text{empty}\ U\ \text{pop}))$

Mapping atomic propositions to DTrace probe and predicate expressions. With the automaton in hand, we map the atomic propositions in the LTL formula (*push*, *pop* and *empty*) to DTrace probe and predicate expressions. We use the `pid` provider to detect function calls within the program, which lets us detect when a function is being called and when a function is returned from. In this way, we can inspect both function arguments and return values. We create the following mapping in JSON as `mapping.json`:

$$\begin{aligned} \text{push} &\rightarrow \texttt{pid\$target::push:entry} \\ \text{pop} &\rightarrow \texttt{pid\$target::pop:return} \\ \text{empty} &\rightarrow \texttt{pid\$target::empty:return/arg1 == 1/} \end{aligned}$$

Anytime the stack program enters the `push` function, our monitor script registers this as a *push* event and updates the internal automaton state accordingly. Similarly, whenever the stack program returns from the `pop` function, the monitor registers this as a *pop* event.

The `empty` function reports whether the stack is empty or not. It returns either 1 or 0, meaning *true* or *false*. Since we are interested in the event "the stack is empty" rather than "the `empty` function is being called", we must check the return value of `empty`. We use a predicate expression for this. The predicate checks that the return value of the function, which the `pid` provider binds to `arg1`, is 1.

We now have all the necessary ingredients. To obtain our monitor, we use the following `graphviz2dtrace` invocation:

```
$ ./graphviz2dtrace.py --mapping mapping.json automaton.dot
```

The listing in Figure 3 shows the salient parts of the generated script, eliding generated comments, and parts of the transition table.

Detecting a violation. To experiment with the monitor, we introduce a fault into the stack implementation. The `push` function does not increment the buffer

```
int HAS_VERDICT;
int state;
int tf[3][3];

dtrace:::BEGIN
{
        tf[0][0] = 0;
        /* ... */
        tf[2][2] = 0;
        HAS_VERDICT = 0;
        state = ($1 ? $1: 0);
}

pid$target::empty:return
/ (arg1 == 1) && (state == 2)/
{
        trace("REJECTED");
        HAS_VERDICT = 1;
        exit(0);
}
```

```
pid$target::push:entry
/state == 2 || state == 0/
{
     state = tf[state][1];
}

pid$target::empty:return
/ (arg1 == 1) && (state == 0)/
{
    state = tf[state][0];
}

pid$target::pop:return
/state == 2 || state == 0/
{
    state = tf[state][2];
}

dtrace:::END / !HAS_VERDICT /
{ trace("INCONCLUSIVE"); }
```

Fig. 3. Generated script

index after pushing a new element onto the stack, ie. the `empty` operation will yield 1 (ie. true) even though elements have been pushed onto the stack:

```
void push(int number, int* i) { buffer[*i] = number; }
```

We demonstrate this by feeding the program a test case via the standard input. Notice that the monitor is called with the `-c` parameter, which tells the monitoring script that it should start the provided program and trace until the target program finishes running. We run the program with `sudo`, as DTrace requires special privileges to run, regardless of the privilege level of the programs being monitored.

```
$ sudo ./monitor.d -c ./stack < incite_error.in
PUSHED 3
PUSHED 4
PUSHED 5
YES
REJECTED
```

Indeed, we see that the last line is `REJECTED`. To ensure against a false positive, we fix the stack implementation to increment on `push`, which should make the monitor output `INCONCLUSIVE`, we recompile the program and run the test case again:

```
$ sudo ./monitor.d -c ./stack-wpushfix < incite_error.in
PUSHED 3
PUSHED 4
PUSHED 5
NO
INCONCLUSIVE
```

As expected, the verdict is `INCONCLUSIVE`, since we have reached the end of the trace and stopped in neither an accepting, nor rejecting, state.

4.2 Verifying interactions between programs

The previous case study concerned a single-process program. What if the system we want to analyze is realized by more than one process? To illustrate how `graphviz2dtrace` can create monitors suitable for these occasions, we will now analyze a simple system consisting of a web server written in Node.js [17] talking to a PostgreSQL [19] database. The point of this case study is not to illuminate some complex system—in fact, the system is made deliberately simplistic to emphasize how the system is instrumented—but rather to discuss what it is like to use `graphviz2dtrace` in practice on a deployed system.

The web server listens to incoming HTTP requests and stores the user-agent strings of the incoming requests in a PostgreSQL database. When the server starts up, it reports its process ID and the process ID of the attached PostgreSQL client to the terminal. Suppose we wanted to ensure that whenever the web server receiving a request, the database completes the corresponding insertion query successfully before the web server sends a response to the client. How could we do that?

In the following, we go through the process of selecting relevant probes corresponding to the events we want to study, specifying the property in LTL, creating the corresponding monitor, attaching it to the running system and detecting violations.

Both Node.js and PostgreSQL have tailor-made static probes that we can make use of. For PostgreSQL, we consulted the listing of available static probes in [20] and used DTrace to find a single probe to associate with the event that a specific PostgreSQL client is done executing a query:

`postgresql$$1:postgres:PortalRun:query-execute-done`.

The `$$1` lets us target a specific PostgreSQL client instance by providing the corresponding PID to the monitoring script via the command line. Furthermore, Node.js has static probes for incoming HTTP requests and responses, which can be tied to a specific Node.js instance as in these probe specifications:

`node$target:node::http-server-request` and
`node$target:node::http-server-response`.

Here, we use the `$target` macro variable to specify the PID of the relevant Node.js instance via the command line.

By not supporting parameterized properties, `graphviz2dtrace` makes it hard to reason about *distinct* events of the same type. However, the predicate mechanism in DTrace is quite expressive. Let us see if we can use the DTrace predicate mechanism to express the property as something that either *should* happen or *should never* happen, and see if we can get closer to our intended meaning.

We wanted to ensure that the server never sends a response to the client before the database management system has completed the corresponding query. Let us rephrase this property in terms of what should never happen:

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.

Suppose we kept three running counters: One for registered requests, another for completed queries, and a third for completed responses. We can achieve this in a D script by adding one probe clause for each event that increments the corresponding counter:

```
int nrequests, nresponses, nqueries;

node$target:node::http-server-request
  { nrequests++; }

node$target:node::http-server-response
  { nresponses++; }

postgresql$$1:postgres:PortalRun:query-execute-done
  { nqueries++; }
```

If we want to add this to a `graphviz2dtrace`-generated script, we must place these probe clauses *before* the clauses related to the automaton logic to get the intended result, since probe clauses associated with the same probe are processed in order. If we place them below the automaton-related clauses, the counters would be incremented *after* we check if the property is violated. Note also that global variables are initialized to 0 by default in D script.

With the counters in place, we can then express the first property as

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

What about the response property? We suggest the following: Define a tolerance level for how big the difference can be between registered requests on the one hand and registered responses and queries on the other. As a starting point, let us arbitrarily specify the tolerance level by saying that this difference should never exceed 100:

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

Having decided on these properties, we need to find a way of associating the atomic propositions of these properties with probe firings so we can detect violations. We can associate the atomic proposition in the precedence property with the `http-server-response` probe:

```
node$target:node::http-server-response/nresponses > nqueries/
```

The response property is not as obvious, but we we can use the special `tick` provider to inspect the state of the monitoring script at a given interval. The `tick` provider fires at a fixed interval on one CPU [18, p. 177]. By associating a suitable predicate with a `tick` event, we can check if the difference between registered requests and registered queries and responses is too large. If we check the property 10 times a second, the probe and predicate specification becomes:

```
tick-10hz/((req - res) > 100) || ((req - queries) > 100)/
```

We then go on to constructing an appropriate automaton. First, we create some aliases. We call the event related to the precedence property *mismatch* and the event associated with the response property *unresponsive*. We then define our specification formula as the following conjunction:

$$(\Box\neg\text{mismatch}) \wedge (\Box\neg\text{unresponsive})$$

We then use `LamaConv` to create the automaton, and `graphviz2dtrace` to create the corresponding script. We also add the counter logic mentioned above to get the counters to work. Finally, to make the verdicts more informative, we also add print statements helping us distinguish between when the property is violated due to the *mismatch* event and when the violation is caused by the *unresponsive* event.

Detecting a violation. With the monitoring script in hand, let us proceed to verifying the system under scrutiny. Again, we have introduced an artificial problem in our code to give our monitor something interesting to observe. We use the following fragment in the web server source code to handle a request:

```
client.query('INSERT INTO entries(entry) VALUES($1)',
        [req.headers['user-agent']],
        function (error, result){
            if(error){
                res.end('Query failed\n');
            }
        });
res.end('Accepted entry\n');
```

Once the database finishes, the runtime executes the code in the anonymous callback function. In the meantime, the webserver can go on processing other events. However, the statement `res.end('Accepted entry\n');` which closes the HTTP response, is *outside* of the callback which fires when the database is done. Therefore it is possible that the statement above is executed *before* the database is done completing the query.

On startup, the Node.js server prints all the information that we need to subsequently attach DTrace:

```
$ node server.js
Server running at 127.0.0.1, port 1337
```

```
Node.js PID is 11509
PostgreSQL client PID is 11510
$ sudo ./monitor.d -p 11509 11510
```

The `p` flag binds `11509` to the `$target` macro variable. Similarly, `11510` will be bound to `$1`. With the monitor attached, we use the Apache Benchmark [2] tool `ab` to send the server a series of requests, and quickly trigger the monitor:

```
$ ab -n 10000 http://127.0.0.1:1337/
...
REJECTED DUE TO MISMATCH
```

We can detect a violation of the response property, too. Running a new benchmark on the server after fixing the callback error above, this time with a high number of concurrent connections via the command line option `-c 200`, we get immediately: `REJECTED DUE TO UNRESPONSIVENESS`.

The tolerance gap of 100 requests in the property was chosen arbitrarily, so this does not have to mean that there is any grave error with the software system as such. Nevertheless, we have seen that the monitor detects a violation.

On concurrency Since the Node.js web server and the PostgreSQL database run as separate processes on a multi-core machine it is both possible and desirable that they do tasks in parallel: Generally, this can also mean that we get two simultaneous probe firings that create a race condition on the monitor's `state` variable. In this specific case, we are in the clear: The clauses in the generated monitor never update the `state` variable, since as soon as either the *mismatch* or the *irresponsive* event is detected a bad prefix has been found and monitoring is terminated.

4.3 Performance

Finally, we would like to observe and discuss the performance of DTrace-based runtime verification. Gregg [12] analyzes the overhead of the `pid` provider and states the following principle about the performance overhead induced by DTrace:

> "The running overhead is proportional to the rate of probes – the more they fire per second, the higher the overhead."

He then formulates the following rules of thumb:

- "Don't worry too much about pid provider probe cost at < 1 000 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."

Paraphrasing our performance evaluation detailed in [21], we see that in the case of the web-server, in our benchmark the system processes roughly 2 000 requests per second, and with three probe firings associated with each request (one for the request, one for the query and one for the response), this only adds up to 6 000 probe firings per second, which is well below this threshold.

In the case of the stack example, we also compared with `printf`-based events (essentially how logging would be implemented). We observe that although this method of event-generation has only half the runtime overhead of using DTrace probes from the `pid` provider, it is a static instrumentation which gives less flexibility, especially in the case where the application does not need to be monitored. In that case, DTrace would have virtually no overhead, whereas the program with the `printf`-statements would have to be recompiled without them.

This shows that we can instrument a running system with DTrace without adversary performance effects, so long as we limit the number of possible probe firings per second to a reasonable level.

5 Conclusion and Future Work

DTrace offers a unique insight into running programs with little overhead. Its main design goal is unobtrusiveness, i.e., apart from a usually minor performance impact, DTrace cannot affect the program execution in any way. Here, we have used DTrace scripts to monitor events provided by the operating system runtime (function calls via C-style libraries or syscalls into the kernel), and through User Statically Defined Tracing, where developers deliberately expose relevant probes to DTrace.

As the scripts run inline with the actual program, we have chosen to encode the transition function of three-valued Linear Time Logic LTL_3 directly through a two-dimensional array in DTrace. The three-valued logic gives us the possibility to yield a verdict on an accepting/rejecting a run as soon as possible.

A major advantage of this approach is that we can associate events from different (operating system-level) processes, possibly even implemented in different languages. We have illustrated the usefulness in two small cases studies, and reflected on performance impacts.

Related Work. The main challenge in applied runtime verification is how to observe a program. Approaches can be divided into those that require access to the source code, and those that do not.

The former rely on recompilation (or byte-code transformation in case of interpreted languages) to be able to intercept relevant events at runtime. Programs are either recompiled with manual annotations or instrumentations, or through a more declarative approach like aspect-oriented programming [16]. The notions of capturing function entries and exits with the possibility to bind values in AspectJ have already been used previously together with temporal logics [24] and trace-based interface specifications [7].

In the second category of tools that can work with a binary representation only of the software, we have log-based tools [14], or those that work on a lower level, e.g. by using advanced emulation or virtualization techniques like Intel's

SAE technology [10]. The latter works on the instruction-level and requires reconstruction of higher-level actions of the program from sequences of assembly instructions.

Another dimension of classification is online versus offline monitoring. In online monitoring, properties are checked in lock-step with the program execution. This also allows the monitor to interrupt, or otherwise interact with the program as soon as a violation is detected. The ability of a system to reason and reflect about its own operating modes and overall system state at runtime has also been termed *runtime reflection* in [4]. In offline monitoring, runtime verification techniques are only used to record a trace, which is then processed later, e.g. for post-mortem analyses. Our DTrace approach realises online monitoring, as scripts are executed inline, yet we have not made use of a feedback mechanism to realise reflection. However, a feedback mechanism could be achieved by connecting the output of the DTrace script to the input of the program, or making use of DTrace's so-called *destructive actions* [18, p. 114], which, among other things, offer direct manipulation of memory contents. This requires active cooperation from the program, in the sense that a developer has to program the application to respond to monitor verdicts.

In future work, we plan to extend the framework with parametrized propositions [23], or quantified event automata [3]. This will allows us to instantiate properties with events that carry values, e.g. to match corresponding identifiers in requests and responses. As we have seen in the web-server example, this is a limiting factor which can only be partially remediated through counters.

It is not clear how concurrent programs can effectively be monitored without race conditions in the action blocks. An obvious, though less elegant, solution would be to use DTrace to only *collect* the trace data, and produce a single stream of interleaved events that is *processed outside* of DTrace.

All source code to the example programs, monitors, and detailed instructions are available in [21] and the accompanying web-page `http://www.mn.uio.no/ifi/english/research/groups/pma/completedmasters/2016/rosenberg/`.

References

1. H. Aalav, G. Avrunin, J. Corbett, L. Dillon, M. Dwyer, and C. Pasareanu. Specification patterns. `http://patterns.projects.cis.ksu.edu/`. Accessed: 2015-08-13.
2. Apache Software Foundation. ab – Apache HTTP server benchmarking tool. `https://httpd.apache.org/docs/2.4/programs/ab.html`.
3. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In D. Giannakopoulou and D. Méry, editors, *18th Intl. Symp. Formal Methods (FM 2012)*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012.
4. A. Bauer, M. Leucker, and C. Schallhart. Model-based runtime analysis of distributed reactive systems. In *17th Australian Software Engineering Conference (ASWEC 2006).* IEEE Computer Society, 2006.

5. A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and N. Garg, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2006)*, pages 260–272. Springer, 2006.
6. 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.
7. E. Bodden and V. Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In W. Löwe and M. Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.
8. B. Cantrill. Hidden in plain sight. *ACM Queue*, 4(1):26–36, Feb. 2006.
9. B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *ATEC '04 Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX, 2004.
10. N. Chachmon, D. Richins, M. Christensson, R. Cohn, W. Cui, and V. Janapa Reddi. Simulation and analysis engine for scale-out workloads. In *Proceedings of the 30th ACM on Intl. Conf. on Supercomputing.* ACM, 2016.
11. J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphviz—open source graph drawing tools. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 483–484. Springer, 2002.
12. B. Gregg. DTrace pid Provider Overhead. `http://dtrace.org/blogs/brendan/2011/02/18/dtrace-pid-provider-overhead/`, 2011.
13. B. Gregg and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD.* Prentice Hall Professional, 2011.
14. K. Havelund and R. Joshi. Experience with rule-based analysis of spacecraft logs. In C. Artho and C. P. Ölveczky, editors, *Formal Techniques for Safety-Critical Systems*, volume 476 of *Communications in Computer and Information Science*, pages 1–16. Springer, 2015.
15. O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
16. R. Laddad. *AspectJ in Action.* Manning Publications, second edition, 2009.
17. Node.js Foundation. Node.js. `https://nodejs.org/en/`.
18. Oracle Corporation. *DTrace Guide for Oracle Solaris 11.* Oracle Corporation, 2012.
19. PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org/`.
20. PostgreSQL Global Development Group. PostgreSQL Documentation: Dynamic Tracing. `http://www.postgresql.org/docs/current/static/dynamic-trace.html`.
21. C. M. Rosenberg. *Leveraging DTrace for Runtime Verification.* Master thesis, Dept. of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, May 2016.
22. T. Scheffel, M. Schmitz, et al. LamaConv—logics and automata converter library. `http://www.isp.uni-luebeck.de/lamaconv`.
23. V. Stolz. Temporal assertions with parametrized propositions. *J. Log. Comput.*, 20(3):743–757, 2010.
24. V. Stolz and E. Bodden. Temporal assertions using AspectJ. *Electr. Notes Theor. Comput. Sci.*, 144(4):109–124, 2006.