UiO : Department of Informatics
University of Oslo

Leveraging DTrace for Runtime Verification

Carl Martin Rosenberg

Master's Thesis Spring 2016

Leveraging DTrace for Runtime Verification

Carl Martin Rosenberg

May 2016

Acknowledgements

I would like to give special thanks to my supervisors Volker Stolz and Martin Steffen for offering their guidance, and Lars Tveito, Kristian Saksvik Munkvold, Axel Rosenberg (my brother) and Lars Kristian Maron Telle for detailed feedback and proofreading.

Also, to Lars (again), Tore Norderud and Sigurd Kittilsen: Thanks for all the fun! Even though our pursuits never seem to be on the critical path[1], I will dearly miss the time we had together on the eighth floor.

Last, but not least, I want to thank my family and friends for their unconditional support.

[1] Pursuits like using model checking to analyze the evolution of awkward seating arrangements on the tram.

Abstract

As we create increasingly complex software systems, we need better tools and concepts to analyze and understand them. On the practical level, we need instrumentation tools that give insights into running systems in a safe and efficient manner. On the analytical level, we need better ways of reasoning about systems as their state and behavior change over time. *Runtime verification* is an emergent field of research that studies how to rigorously specify properties about software systems and how corresponding *monitors* – programs that check if the system satisfies these properties – can be built. In this thesis, we investigate how we can leverage the DTrace instrumentation framework [8] to conduct runtime verification. To this end, we develop `graphviz2dtrace`, a tool for producing monitor scripts in DTrace's domain-specific scripting language D from specification formulas written in LTL_3, a three-valued variety of the well-known Linear Temporal Logic. We evaluate the tool by analyzing both single- and multi process systems.

Contents

List of Figures

List of Tables

Chapter 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. In this thesis, we investigate using the DTrace [8] framework for this purpose.

Originally developed by Bryan Cantrill, Adam Leventhal and Mike Shapiro for Sun Microsystems, DTrace lets a user instrument practically all aspects of a running system. Crucially, DTrace has mechanisms for instrumenting the system in a *dynamic* manner, providing insights into programs without requiring pre-compiled static artifacts in the program source code. The user simply specifies events of interest, and associates actions that DTrace should take every time the event occurs. These actions are specified in an AWK-like scripting language called D, which provides several convenient language constructs for data aggregation and presentation.

When we analyze programs that we do not have the source code for, or if the pre-programmed logging mechanisms cannot answer our questions, DTrace's dynamic tracing capabilities enable us to get insights we otherwise could not get. This makes Runtime Verification possible in places where it previously was not.

In this thesis, 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 common specification logic Linear Temporal Logic (LTL). When used in conjunction with the `LamaConv` automata library, `graphviz2dtrace` provides a runtime verification platform.

2. We use `graphviz2dtrace`-based monitors to verify two software systems: A simple stack implementation written in C, and a web appli-

cation consisting of a Node.js [14] web server communicating with a PostgreSQL [18] 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.

1.1 Chapter Overview

We set the stage in the next chapter by introducing runtime verification, focusing on what separates runtime verification from *model checking*.

In chapter 3 we familiarize the reader with LTL_3, a three-valued extension to Linear Temporal Logic (LTL) and the main verification formalism in this thesis. We introduce LTL_3 in the context of classic LTL, and introduce the distinguishing three-valued semantics of LTL_3 which is based on the concepts of *good and bad prefixes*. Finally, we describe the procedure given in [4] for generating LTL_3-based monitors.

Having set the theoretical stage, we turn our attention to the DTrace instrumentation framework in chapter 4. We describe the main components: Probes, providers and the D scripting language. We also discuss the precise meaning of *dynamic instrumentation* in the context of DTrace.

With the practical and theoretical foundations in place, we move on to describing the design and implementation of `graphviz2dtrace` in chapter 5. We describe how we 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 limited by the inherent limitations of D, especially with respect to concurrency.

Having presented the design and implementation of `graphviz2dtrace`, we use the tool in two case studies. In the first case study in chapter 6, we analyze a simple implementation of the classic stack data structure written in C. We demonstrate how to detect a violation of a property and how to hook a monitoring script onto a running process. We also analyze the performance degradation we induce through the act of monitoring. In the second case study in chapter 7, we analyze a system composed of a web server written in Node.js and a PostgreSQL database. We analyze the process of finding observable events, associating the found events to atomic propositions in LTL specification formulas, and using the generated monitors to detect property violations. As in the first case study, we investigate the performance penalty induced by monitoring.

We evaluate our findings in chapter 8, and draw our conclusion in chapter 9.

1.2 Project website

All code and data written and collected as a part of this thesis is available on http://www.mn.uio.no/ifi/english/research/groups/pma/completedmasters/2016/rosenberg.

Chapter 2

Runtime Verification

Informally, *runtime verification* is a branch of computer science concerned with verifying the behavior of programs by analyzing concrete runs of said programs. It is related to the field of Model checking, which it borrows many concepts from. We will therefore present Runtime Verification by comparing and contrasting it to traditional Model Checking.

2.1 Model Checking

A program can behave in bewilderingly different ways, and can assume an astronomical number of unique states. To see this, it is enough to consider how many possible states we introduce when we permit the program to input a single integer. An integer can take up to 2^{64} different values on a modern system, leading to potentially 2^{64} different outcomes. Even though most programs only consider some select cases, this indicates how quickly the number of possible program states grows–and this is just when considering the program as a closed system. In order to be useful, most programs must interact with other programs and the underlying operating system. As a result, the number of possible states increases exponentially. Ultimately, even if we had the omniscience required to understand every factor that influences a program's state, *the halting problem* establishes some absolute limits on the possibilities of software verification: The general software verification problem is *undecidable* [30].

Nevertheless, there are meaningful forms of software verification that can be done within the bounds set by the halting problem. One approach is to accept that while we cannot understand real software completely, we can make simplified *models* of the software and reason about these models instead. *Model checking*[1] exemplifies this approach. It is a form of automated software verification that proceeds in three steps [9, p. 4]:

[1] For a general introduction to Model checking, see [9] or [2].

The Modeling Step Derive a formal model of the program in question (typically, some Finite State Automaton is used).

The Specification Step Specify a property ϕ that the model should satisfy using an appropriate notation. For this purpose, one typically uses a temporal logic like Linear Temporal Logic (LTL).

The Verification Step Do an exhaustive search for a computation[2] that falsifies ϕ. If no such computation exist, then the model satisfies ϕ, which we write as $M \models \phi$.

Model checking has been very successful in verifying finite state systems like hardware controllers and communication protocols [9, p. 4]. However, Model checking suffers from some inherent limitations that motivate the need for a complementary verification technique. First of all, Model checking is plagued by the *state explosion problem*[3], which we implicitly discussed above: As the number of possible states a program can enter increases exponentially, it becomes harder and harder to exhaustively check a corresponding model in reasonable time. This limits what you can analyze with Model checking. Secondly, as Leucker points out in [24, p. 295]: As model checking requires a concrete model of the program in question, one cannot apply Model checking to *black box systems*, i.e. systems where one knows very little or nothing about the internal workings of the system. In the next section, we will see how Runtime verification approaches some of these limitations.

2.2 Runtime Verification

As a starting point, let us use the definition of Runtime verification given by Martin Leucker [23, p. 36]:

> 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*.

As is clear from this definition, Runtime Verification differs from traditional Model checking by analyzing *runs* of programs rather than *models* of the program. The most important implication of this is that Runtime Verification can only say something about one particular run: It cannot purport to say something about all possible program runs, as in traditional Model checking. However, where the legitimacy of traditional Model checking depends on how well the model represents the system, Runtime Verification bases itself on data obtained in the real world.

[2] In this context, a computation is the same as a run of a program, represented as a history the program's states.

[3] See for instance the discussion in [9].

In the previous section, we saw that Model checking proceeds in three steps: A modeling step, a specification step and a verification step. Runtime Verification is similar to Model checking in the *Specification Step*, but differs from traditional Model checking in the *Modeling* and *Verification* step.

In the *specification step*, properties are specified using an appropriate Logic, typically some variant of Linear Temporal Logic. Once the property is specified, one needs to derive a representation of the property that can be used to analyze program traces. For this purpose, one typically derives a *monitor* – a program that consumes data from the program 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 [24, p. 294–295].

In practice, such monitors are automata-based and typically automatically created from some specification formula. We discuss this in detail in chapter 3.

Returning to the comparison to Model checking: We see that the *Verification step* is carried out by having the derived monitor consume a trace of the run and give a verdict. If the trace cannot provide sufficient information for a verdict, the monitor should report "inconclusive". This should also be reflected in the specification logic. A specification logic that incorporates this is LTL_3, which is a three-valued Linear Temporal Logic [24, p. 296–298]. We take a deeper dive into LTL_3 in the next chapter.

As Martin Leucker states in [23, p. 39], "runtime verification is mainly concerned with the synthesis of efficiently operating monitors." To see why this problem is essential, consider the following: Firstly, the monitor must properly represent the specified property, i.e. the specification logic must be faithfully translated into monitor code. Secondly, since the monitor typically runs on the same system as the program under scrutiny, one must ensure that the monitor does not interfere with the program being monitored, and that the monitor does not take up excessive computational resources.

Often, Runtime Verification aims to analyze systems while the systems are still running. This form of Runtime Verification is called *Online* Runtime Verification, and stands in contrast to *Offline* Runtime Verification, in which the program runs are analyzed in a postmortem fashion. One particularly exciting aspect of *Online* Runtime Verification is that it makes it possible to *act* upon incoming information. This can be used to create so-called "steering systems" that automatically alter a running system in response to some event. For example, one could make the steering system terminate a program if the program started to violate a safety property [24].

In this thesis, we present a tool for creating runtime verification monitors implemented as DTrace scripts. The resulting monitors are made for *online* runtime verification, and supports instrumenting both "black box" systems and systems which we know the internals of. Before we can describe the details of `graphviz2dtrace`, we will present the theoretical foundations we make use of in `graphviz2dtrace` as well as the DTrace instrumentation tool.

Chapter 3

Theoretical Foundations

As we saw in the previous chapter, Runtime Verification is in large part concerned with creating *monitors*-programs that check if a system satisfies some property by evaluating runtime data about the system. In order to automatically and correctly create such monitors, runtime verification exploits insights from Logic and Automata Theory: Practitioners of RV use Logic to specify properties in an unequivocal manner, and exploit connections between Logic and Automata Theory to automatically create monitors for these properties. In this chapter we explore the key formal pieces needed to understand how we do Runtime verification in this thesis: How we can use Linear Temporal Logic (LTL) to specify properties of programs, and how we can exploit the connection between *Büchi Automata* [6] and LTL formulas to create monitors.

In essence, given a description of a program's behavior and state over time – what we call a *trace* – LTL checks whether a property is true for the trace. The "classic" LTL is concerned with *infinite* traces. In practice, we can only extract finite traces from our running programs. By adding the concept of an *inconclusive* verdict to LTL, we get the three-valued logic LTL_3, which gives a meaningful way of dealing with finite traces[1]. For this reason, `graphviz2dtrace` uses LTL_3 as a logical foundation: Users express properties in standard LTL syntax, and `graphviz2dtrace` produces monitors adhering to LTL_3 semantics. We therefore present LTL_3 semantics, which is based on the concepts of good and bad prefixes introduced in [22].

3.1 Linear Temporal Logic

A temporal logic is a logic dealing with *time*. In the context of formal methods, Temporal Logic is used to reason about the evolution of some system over time. Different notions of time give rise to different temporal

[1] LTL_3 was first introduced in [3]. In this chapter we follow the presentation given in [4].

logics, and the best logic to use in a given context will depend on which notion of time best captures the phenomenon one wishes to study. Computation Tree Logic[2], for example, represents time as a series of decision points: A decision is made, and time diverges into separate paths representing what the world becomes if the choice was A, if the choice was B, etc. Linear Temporal Logic (LTL), on the other hand, treats time as a sequence of states, where each state represents a "snapshot" of the world at a given unit of time. This logic is *linear* in the sense that the sequence of states forms a straight line.

The idea of using Linear Temporal Logic for program verification originated with Pnueli [27][3]. In this chapter, we follow the exposition on LTL given in [31].

3.1.1 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*).

We call the set of atomic propositions AP and recursively define the set of well-formed LTL formulas LTL as follows:

Definition 3.1.1 (Set of well-formed LTL formulas)**.** Let LTL be the minimal set such that $\phi \in LTL$ if and only if one of the following holds, where $\psi, \gamma \in LTL$:

- $\phi \in AP$.
- $\phi = (\psi \wedge \gamma)$.
- $\phi = (\psi \vee \gamma)$.
- $\phi = (\psi \rightarrow \gamma)$.
- $\phi = \neg\psi$.
- $\phi = \Box\psi$.
- $\phi = \Diamond\psi$.

[2]For an extensive treatment of Computation Tree Logic, see [2, p. 313 - 433].

[3]For a broader treatment of the origins and evolution of temporal logic, see the discussion in [25, p. 268-273].

- $\phi = \bigcirc\psi$.
- $\phi = (\psi \, U \, \gamma)$.
- $\phi = (\psi \, R \, \gamma)$.
- $\phi = (\psi \, W \, \gamma)$.

3.1.2 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*[4] . In later sections, we will see how this notion of a sequence of states can correspond to a run of an automaton. This section follows the presentation of LTL semantics given in [31], but out of preference we adopt a slightly different notation[5].

Let

1. $\sigma(k)$ denote the k-th state in the trace σ (counting from 0), and let
2. σ^k denote the trace starting in and including $\sigma(k)$, i.e. $\sigma(k), \sigma(k+1), \sigma(k+2) \ldots$

On this basis, we say that a trace σ models ϕ, or that ϕ is *true* relative to a trace σ, denoted $\sigma \models \phi$, subject to the following conditions:

$\sigma \models \phi$ iff $\sigma(0) \models \phi$, when ϕ is atomic
$\sigma \models (\phi \wedge \psi)$ iff $\sigma \models \phi$ and $\sigma \models \psi$
$\sigma \models (\phi \vee \psi)$ iff $\sigma \models \phi$ or $\sigma \models \psi$
$\sigma \models (\phi \rightarrow \psi)$ iff $\sigma \not\models \phi$ or $\sigma \models \psi$
$\sigma \models \neg\phi$ iff $\sigma \not\models \phi$
$\sigma \models \Box\phi$ iff $\sigma^k \models \phi$ for all $k \geq 0$
$\sigma \models \Diamond\phi$ iff $\sigma^k \models \phi$ for some $k \geq 0$
$\sigma \models \bigcirc\phi$ iff $\sigma^1 \models \phi$
$\sigma \models (\phi \, U \, \psi)$ iff $\sigma^k \models \psi$ for some $k \geq 0$, and $\sigma^i \models \phi$ for every i such that $0 \leq i < k$
$\sigma \models (\phi \, R \, \psi)$ iff for every $j \geq 0$, if $\sigma^i \not\models \phi$ for every $i < j$ then $\sigma^j \models \psi$
$\sigma \models (\phi \, W \, \psi)$ iff $\sigma \models \phi \, U \, \psi$ or $\sigma \models \Box\phi$.

In the case that σ is not a model for ϕ, we say that ϕ is *false* relative to σ, and we denote this with $\sigma \not\models \phi$.

[4]Nomenclature varies: Some prefer to to use *paths*, other use *words* to emphasize the connection to formal language theory. We choose to use traces, as it makes it easier to understand the connection between the theoretical and practical concept as we move on.

[5]Specifically, we use $\sigma(k)$ instead of σ_k to denote the state at position k.

Example

Consider the fragment of trace σ shown in Figure 3.1:

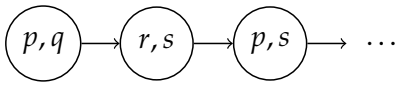

Figure 3.1: A trace σ

Some observations:

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

3.2 LTL$_3$

While standard LTL semantics show us how to evaluate whether a trace σ models a formula ϕ, the semantics is defined for *infinite* traces. In practice, however, we can only produce finite traces. While there exists semantics for LTL based on finite traces with the usual two truth-values, Bauer et al. argue persuasively that one should employ a three-valued semantics for dealing with Finite Traces–using a third *inconclusive* option to account for the cases in which the finite trace is insufficient to draw a verdict [4, p. 14:4]. The resulting three-valued logic is called LTL$_3$.

3.2.1 Semantics of LTL$_3$

The semantics of LTL$_3$ are based on the concepts of *good and bad prefixes*, originally developed by Kupferman and Vardi in [22]. Where Kupferman and Vardi (and following them, Bauer et al. in [4]) defined these concepts in terms of formal language membership, we define them in terms of traces to make the connection to our discussion of LTL semantics clearer:

Definition 3.2.1 (Good prefix wrt. ϕ)**.** Let $\sigma = u\omega$ be an infinite trace consisting of a finite trace u concatenated with an infinite trace ω. Then u is a *good prefix* wrt. an LTL formula ϕ if and only if $u\omega \models \phi$ for all ω.

Definition 3.2.2 (Bad prefix wrt. ϕ)**.** Let $\sigma = u\omega$ be an infinite trace consisting of a finite trace u concatenated with an infinite trace ω. Then u is a *bad prefix* wrt. an LTL formula ϕ if and only if $u\omega \not\models \phi$ for all ω.

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

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

Note that we use $\models_3$ rather than $\models$ to differentiate the notion of semantic logical consequence in LTL_3 from the standard notion in LTL.

On reflection, we see that the semantics of LTL_3 imply that good or bad prefixes may not exist for certain properties: Consider the property $\Box\Diamond p$ ("always eventually p"): We can never obtain a finite trace that gives a verdict, because there is always a possible future in which the property is either violated or satisfied [4]. If no verdict can be obtained for a property, we call that property *nonmonitorable*[6]:

Definition 3.2.3 (Nonmonitorable property)**.** A property ϕ is *nonmonitorable* if there exists no u such that u is either a good or a bad prefix for ϕ.

3.3 Creating LTL_3 monitors

Manually evaluating properties on traces is error-prone and practically impossible to do in most cases. By leveraging a formal connection between Logic and Automata theory we can derive procedures for automatically generating monitors for logical formulas. Conceptually, this resembles the way in which Finite Automata can be derived from Regular Expressions.

Drawing on the foundational work done by Büchi in [6], there are ways of deriving a *Büchi Automata* from an LTL expression. A Büchi Automaton is similar to the well known Finite State Automata, but differ in the acceptance condition: Where a finite automaton accepts an input trace if and only if the automaton is in an accepting state when the trace is fully consumed, Büchi Automata are defined on *infinite* traces and accept a trace if and only if the automaton visits some accepting state infinitely often.

Equipped with a method of deriving Büchi Automata for LTL properties, Bauer et al. give an algorithm for creating LTL_3-monitors [4, 14:10-14:13], which we state in outline and comment on:

1. For an LTL formula ϕ, derive the corresponding Nondeterministic Büchi Automaton NBA_ϕ.

[6] This notion of nonmonitorability was first developed in [28]. We follow the exposition given in [4].

2. Derive $NBA_{\neg\phi}$ for $\neg\phi$, the negation of ϕ.
3. Find the maximal *Strongly Connected Components*[7] (SCCs) of NBA_ϕ, and use these to derive a function $F_\phi : q \to \{\top, \bot\}$, where q is any state in NBA_ϕ. F_ϕ is defined as follows:

$$F_\phi(q) = \begin{cases} \top \text{ if } q \text{ can reach an SCC with } \textit{at least one} \text{ accepting state} \\ \bot \text{ otherwise} \end{cases}$$

.
4. Derive $F_{\neg\phi} : q \to \{\top, \bot\}$ for $\neg\phi$ using the procedure in step 3.
5. Using F_ϕ, derive a Nondeterministic Finite State Automaton NFA_ϕ for ϕ having the same states, alphabet, starting state and transition function as NBA_ϕ but where the set of accepting states Q_a^ϕ is defined as $\{q \in Q_a^\phi | F_\phi(q) = \top\}$.
6. Derive $NFA_{\neg\phi}$ such that the set of accepting states $Q_a^{\neg\phi}$ is such that $\{q \in Q_a^{\neg\phi} | F_{\neg\phi}(q) = \top\}$.
7. Find DFA_ϕ, the Deterministic Finite State Automaton corresponding to NFA_ϕ.
8. Derive $DFA_{\neg\phi}$ corresponding to $NFA_{\neg\phi}$.
9. Create the product automaton $PROD_\phi = DFA_\phi \times DFA_{\neg\phi}$, in which each state $(q_i^\phi, q_j^{\neg\phi})$ is labeled according to the function λ defined by

$$\lambda((q_i^\phi, q_j^{\neg\phi})) = \begin{cases} \top & \text{if } q_i^\phi \notin Q_a^{\neg\phi} \\ \bot & \text{if } q_j^{\neg\phi} \notin Q_a^\phi \\ ? & \text{otherwise.} \end{cases}$$

10. Minimize $PROD_\phi$ to obtain $MONITOR_\phi$.

To derive a Büchi Automaton which gives verdicts according to an LTL formula ϕ as in step 1, one usually employs a construction developed by Vardi and Wolper in [33]. A very accessible explanation of the procedure can be found in [34].

Furthermore, as Bauer et al. make clear [4, p. 14:10], the purpose of the functions F_ϕ and $F_{\neg\phi}$ is to determine whether the language of the Büchi Automaton *starting in* state q is nonempty–which it is if it is possible to feed the sub-automaton a trace which could get accepted. Note that one cannot simply ask if the state is *in* an SCC with an accepting state: When starting in state p it might be possible to reach the accepting state q residing in a different component, even though there is no path back from q to p.

[7] A strongly connected component of a graph G is a subgraph of G in which for every pair of nodes n, m in the graph there is a path from n to m. For a detailed exposition of Strongly Connected Components, see [11, p. 615-620 and p. 1170-1171.]

The deterministic counterparts of NFA_ϕ and $NFA_{\neg\phi}$ are obtained using a standard technique relying on the equivalence of NFAs and DFAs, as shown in [30, p. 54-56].

The purpose of the product automaton is to have a device that simulates what happens if you feed the input trace to both DFA_ϕ and $DFA_{\neg\phi}$ simultaneously. Consequently, the states in the product automaton correspond to possible combinations of states from DFA_ϕ and $DFA_{\neg\phi}$, respectively. We use the notation $(q_i^\phi, q_j^{\neg\phi})$ to indicate that the product automaton state we are in simulates that DFA_ϕ is in state q_i^ϕ while $DFA_{\not\phi}$ is in $q_j^{\neg\phi}$.

When $MONITOR_\phi$ consumes a trace, the verdict the monitor gives is determined by the labeling function: If the monitor is in in a state labeled $\top$ when the trace is consumed, the verdict is *true*; if the label is $\bot$, it is *false*; if the label is ? the verdict is *inconclusive*.

Chapter 4

DTrace

DTrace is an operating system technology for monitoring running software systems. 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. DTrace can give insights into almost any aspect of a running system, from the behavior of a single process to the internals of the operating system kernel. 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 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 [17]. 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.

4.1 Crucial aspects of the DTrace architecture

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: Some DTrace components acquire the requested data: These are called *producers*. Other components post-processes 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: Producer components 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. In userland, there are a series of consumer components. 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 parses and executes D-scripts, and calls upon the underlying producers to acquire the requested data.

4.2 Using DTrace: The D scripting language

We mentioned previously that DTrace provides a way of describing events of interest and specifying what should happen when these events occur. To achieve this, users create scripts using the D scripting language.

To make the following discussion concrete, Figure 4.1 shows a simple, commented D script.

4.2.1 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 time. The DTrace producer components, which expose the probes to the consumer components, will only expose probes that are safe to use [7, p. 31].

Each probe is uniquely identified by the four-tuples on this form:

```
<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"). We briefly discuss what each component of this tuple refers to.

Providers A provider is a library of probes, organized by a common theme. Most DTrace installations have the following providers [17, p. 11–12]:

`dtrace` This is a special provider for probes related to the DTrace framework itself. Users can utilize these probes to specify what should happen when a given D script starts, ends or throws an error.

```
#!/usr/sbin/dtrace -s

/* This is a comment. Comments in D are inspired by
 * the C programming language. The statement at the
 * top of the file makes it easier to execute the
 * script from the command line.
 *
 * (Atribution: This script is adapted from
 * (Gregg, 2011, p. 7)).
 *
 * The first line after this comment specifies a
 * probe. This specific probe fires whenever a
 * process issues the 'read' system call.
 * The probe is identified in the following manner:
 *
 *      Provider: syscall
 *      Module: wildcard. Matches any applicable module.
 *      Function: read
 *      Name: entry (fires upon function entry)
 *
 * The second line is a predicate: It asks DTrace to
 * disregard every call to 'read' done by the DTrace
 * program itself. Consequently, when Dtrace itself
 * issues a 'read' system call, the action block is not
 * executed.
 * */

syscall::read:entry
/execname != "dtrace"/
{
    /* This (the contents surrounded by the curly
     * braces) is an action block. The statements
     * within will be executed when the probe fires. */

    /* This command will print the name of the process
     * issuing the 'read' system call */
    printf("%s\n", execname);
}
```

Figure 4.1: A simple D-script

`syscall` This provider catalogs all probes related to the system call API, which processes use to interact with the operating system kernel.

`proc` This instruments events related to processes and threads, such as calls to `fork` and `exec`.

`profile` This is a collection of probes for collecting timed data that can be used in performance analysis.

`fbt` An acronym for *function boundary tracing*, this provider manages probes for functions in the operating system kernel.

`lockstat` This provider instruments synchronization events in the kernel, notably the acquiring and releasing of locks.

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 as well as many programming language runtimes[1].

Modules For kernel probes, the module field indicates which kernel module the probe belongs to. For example, on FreeBSD installations that have the ZFS file system, the probes that instrument this file system can be addressed by specifying `ZFS` in the module field. For userland probes, this field may indicate a shared library that organizes the probes. On Mac OS X El Capitan, the instrumentation points for the Python runtime are organized in a `python` module. However, most probes are not assigned to a specific module.

The 'function' field If the probe is attached to an actual function, the name of this function will be in the 'function' field. For example, if we are monitoring a C program, we might be interested in the `malloc` function [17, p. 10].

The 'name' field The name field is the last part of the namespace, and serves as a "meaningful name" for the probe. The best examples are the names `entry` and `return`, that correspond to when a given function is entered into and returned from, respectively [17, p. 10].

[1] See pages 1051 to 1063 in in [17] for an extended introduction to USDT Probes. The same source also covers PostgreSQL probes on pages 851–858, as well probes for C++, Java, Javascript, Perl, PHP, Python, Ruby and Tcl.

4.2.2 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 [17, p. 37–42]. The available action statements will vary between DTrace implementations. For a comprehensive overview of the actions available on Solaris, see appendix D in [17, p. 1011–1024].

4.2.3 Filtering out the noise: Predicates

When a probe fires, users can use predicates to determine if a certain action block should occur or not; in essence, the predicate serves as a filter. Predicates are written as boolean expressions that can use any D operator and any D data object. The predicate expression must evaluate to some integer or pointer that can be interpreted as true or false in line with the C convention: 0 or NULL is treated as false, all other values are treated as true. The user can choose not to specify a predicate. This is equivalent to specify the predicate `/true/`, meaning that no filter is present and the action block will be executed unconditionally.

4.3 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 [7, 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 [16]. The previously mentioned `fbt` provider makes it possible to instrument all function return values and arguments in the operating system kernel source code [17, 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 [17, p. 788-791]. In chapter 6, we use the `pid` provider to dynamically instrument a stack program in C.

Chapter 5

Design and Implementation of `graphviz2dtrace`

Let us take stock: We have learned how we can use Linear Temporal Logic to specify properties of programs and how to evaluate whether a *trace* makes the property *true*, *false* or *inconclusive* if the trace is not enough to draw a verdict. We have also had a brief encounter with DTrace and how we can use it to harvest information from running programs. It remains to be seen how we can combine DTrace and LTL_3-based automata into a complete Runtime Verification platform. In this chapter, we describe `graphviz2dtrace`, a tool which produces monitoring scripts in D adhering to LTL_3 semantics.

At a high level, we would like to accomplish the following:

1. Gather data about the running system with DTrace.
2. Feed the obtained data as a trace to a monitor.
3. Make the monitor output a verdict.

There are several ways to go about doing this, and we will now explain the most important decisions we have made in designing and developing `graphviz2dtrace`. The `graphviz2dtrace` source code is listed in Appendix D.

5.1 Main ideas

Given that our view of the software being verified is limited to what DTrace can observe, how do we make a meaningful connection between the things that DTrace can observe and what we reason about in our LTL formulas? Concretely, what should the atomic propositions in our LTL specifications stand for? Our solution to this question is the following:

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

To exemplify this principle, suppose we had a C program implementing a simple stack program with functions `push`, `pop` and `empty` for checking if the stack is empty (we investigate such a program in chapter 6). We can then associate the atomic proposition *push* with the probe specification `pid$target::push:entry`. Then every time DTrace detects that the process in question enters the `push` function, we formally treat this as if *push* is true. Furthermore, the atomic proposition *empty* can be associated with `pid$target::empty:return/ arg1 == 1/`. In this way, the atomic proposition *empty* becomes true whenever DTrace detects that the process in has called the `empty` function and that the `empty`-function is returning 1 (ie. 'true' in C semantics). Notice how we use both a probe specification (`pid$target::empty:return`) and a predicate (`/ arg1 == 1/`).

In effect, we let users analyze systems by specifying events of interest and reasoning about these events using temporal logic. An event of interest is specified as a DTrace probe description with an optional predicate and nothing else. While DTrace can provide an enormous variety of probe specifications, the downside of this approach is that phenomena which cannot be expressed as a firing probe cannot be analyzed. A consequence of this is that there is not a one-to-one relationship between what users can reason about in properties and what information it is technically possible for DTrace to obtain.

Having decided on the conceptual bridge between LTL and DTrace concepts, the second design decision concerns implementing the monitor. For creating automata from LTL formulas, we use the `LamaConv` [29] automata library. However, it remains to be seen how we turn this automaton specification into a concrete program.

Conceptually, we use DTrace to produce a trace, which is fed to a monitor which in turn decides if the trace is a good, bad or inconclusive prefix. Interpreting this schema in a concrete way, it might make sense to create the monitor as a program separate from the D script we use to obtain the trace. However, we also know that the D scripting language has the necessary building blocks for embedding automata: The automaton's transition function can be encoded into a two-dimensional array, and a variable can be used to keep track of the state the automaton is in. As probes fire, the action block associated with the probe can update the state variable according to the transition function matrix. In this respect, we made the following decision:

Encode both the trace generation and monitor logic in one script.

We made this decision early on in the project to explore the possibilities and limitations of the D language. At the outset, this seems like an attractive option: Since both trace generation and trace validation is happening in the same program, there is no involved overhead in communicating between the trace-producer and the monitor. In theory, this should make for a faster

and less intrusive solution than the alternative. It is also easier to build a tool with one rather than two target languages.

However, by making this decision you are accepting not only the possibilities but also the *limitations* of the D scripting language. The most important limitation you will have to address is that *there is no way to have a globally accessible yet synchronized state variable in D*: You can have a global state variable, but then a race condition occurs if two probe firings overlap. You can have a thread-local state variable, but then you are either forced to assume that all specified events register to the same kernel thread, which severely limits the kinds of properties you can express, or you will have to do some form of synchronized post-processing to generalize a global result from the states of each kernel thread state. In this regard, we made the following decision:

> *Use a global variable to keep track of the automaton state, even though this means that you cannot safely verify properties dealing with potentially overlapping events.*

In chapter 8, we evaluate how good these decisions were in comparison to other options.

5.2 Implementation

Having described the main ideas underlying the design of `graphviz2dtrace`, let us now turn to the implementation details.

In essence, `graphviz2dtrace` is a source-to-source compiler taking LTL_3-based automata and creating corresponding D scripts. It proceeds in phases, and we describe each phase as we follow the control flow through the program. The control flow is summarized in Figure 5.1.

In the simplest case, `graphviz2dtrace` is invoked with one parameter referencing a graphviz dot file. We also allow the user to specify a mapping between atomic propositions and DTrace probes in the form of a JSON file. We use `docopt` [21] to parse the command-line arguments and to automatically generate a command-line interface which should feel familiar for most UNIX users.

After we have parsed the command line options, the submitted automaton must be parsed into a usable data structure. For this purpose, we use `PyGraphviz` [20], which parses a graphviz dot file into a graph data structure with convenient methods for accessing and manipulating the automaton. With `PyGraphviz`, we can easily iterate through the states and edges in the automaton. We assume that the automaton is encoded in the same way as `LamaConv` produces its graphviz output. Specifically, we assume that the automaton has `fillcolor=red` on rejecting states, `fillcolor=green` on accepting states and that there is a hidden, 'artificial state' from which there is an edge labeled 'START'.

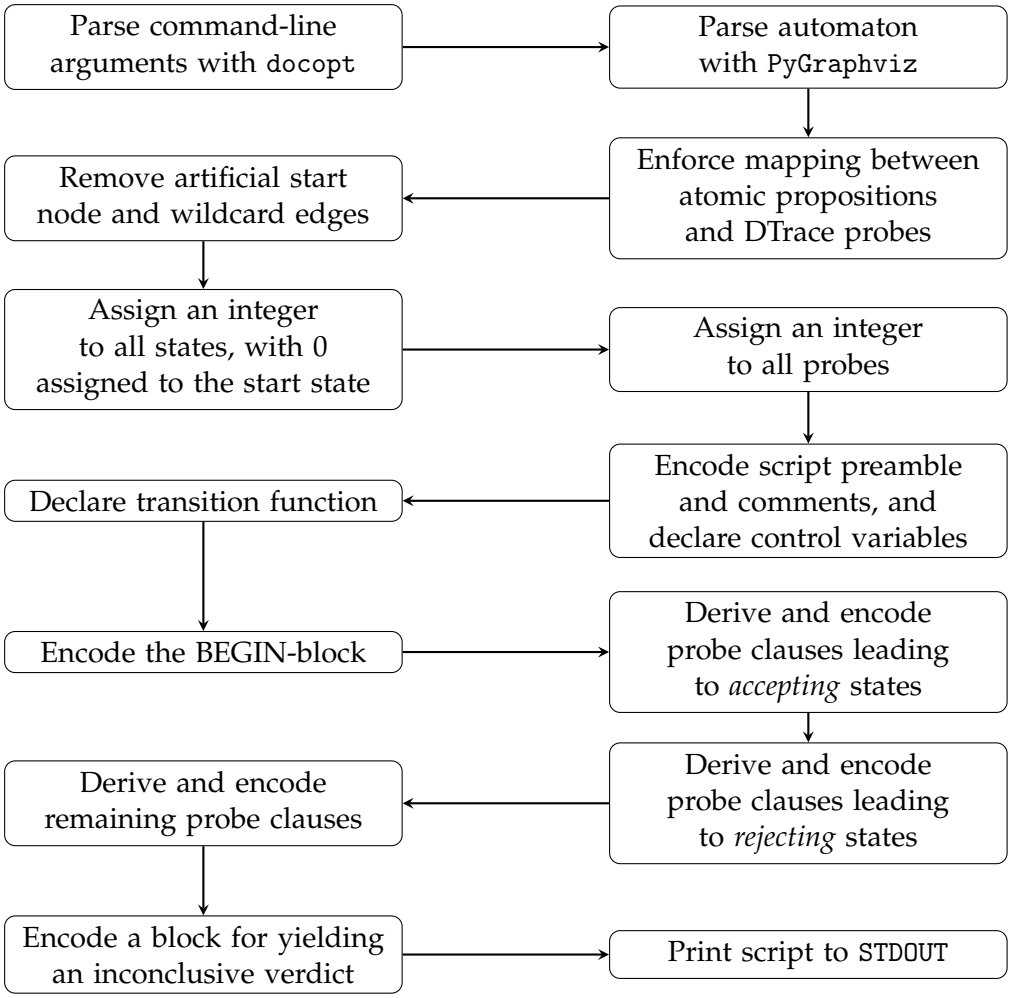

Figure 5.1: Control flow in `graphviz2dtrace`

In the simplest case in which `graphviz2dtrace` is invoked with only one parameter referencing a dot file, `graphviz2dtrace` assumes that the edge labels in the automaton are valid probe specifications. In practice, however, it is so inconvenient to use fully qualified DTrace probe names as variables in LTL_3 expressions that users will typically always use a mapping. In fact, using probe names in `LamaConv` invocations causes syntax errors. Therefore we let users submit a separate JSON file with atomic propositions as keys and probe names as values. We parse this JSON file and replace the atomic propositions in the automaton graph with proper probe names.

We now have a data structure representing the automaton and we have inserted the correct probe references into the automaton. Before we start to encode the final script, we do two more transformations on the automaton. The first transformation removes some artifacts in the automaton that only serve to visually indicate where the start node is. Recall that `graphviz2dtrace` assumes that the provided automaton has been generated by `LamaConv`. A standard artifact of automata created by

`LamaConv` is a hidden node. By making the 'START' edge an edge between the hidden node and a real node, the visual effect is that the 'START' edge indicates where the start node is (see Figure 5.2). After using the 'START' edge to find the start node, we remove it, along with the hidden node.

The second transformation concerns what we will call *wildcard edges*, meaning edges labeled with '?'. In `LamaConv` usage, a '?' indicates that the edge can be followed regardless of what event occurred. In most (but not all) specifications these wildcard edges will always be edges from a node to itself, indicating that the state is a *sink state*: If the automaton reaches such a state, it will stay in that state regardless of future events. If we on the other hand discover that a wildcard edge is used between two nodes, `graphviz2dtrace` currently terminates with an assertion error: We discuss some ideas for how `graphviz2dtrace` could handle such events in the future in section **??**.

Having removed redundant edges and nodes, we start figuring out how to represent the automaton in a D script. Our basic strategy is to use a two-dimensional integer array, in which the rows represent the state the automaton is in, the columns represent which probe is firing, and the value stored in the table cell indicates which state the automaton should go to. To do this, we have to associate every state with a number, and every probe with a number.

Recall that when removing the hidden node and the 'START' edge, we also determined the start node in the automaton. We associate the starting node with 0, since then we can always initialize the script with the state variable set to 0. For the remaining states we give a unique number, and we keep this mapping between states and integers in a dictionary. Similarly, we give each probe-predicate expression a unique integer, and store this in another dictionary.

We now have the required ingredients to start creating the resulting script. `graphviz2dtrace` represents the resulting script as a list of strings, where each string represents some associated set of statements. When all the components of the script have been gathered, one final string is created from the list by concatenating the strings in the list, inserting a newline between each string.

The resulting script starts with a standard DTrace script header, a signature comment saying that the script is produced with `graphviz2dtrace` as well as comments detailing which integers the probes and states have been mapped to. We also declare the state variable as well as the `HAS_VERDICT` control variable[1]. After that, we use the dictionaries obtained above to infer the dimensions of the transition function table, and with the dimensions in hand we declare the transition function table.

Having declared the transition function table, the state and the `HAS_VERDICT`

[1] We will soon explain what `HAS_VERDICT` is used for.

variables, we define the `dtrace:::BEGIN` block, the probe clause which we use to initialize the declared components. Recall that D-script is programmed by attaching statements to probe firing events. `dtrace:::BEGIN` is a special DTrace probe which fires when the D-script starts. The statements in the BEGIN block will be carried out before any other probe firings are handled.

Before we can initialize the transition function in the BEGIN-block, we must determine which values to put in each cell of the transition function table. We do this by iterating through each node in the automaton and looking at its outbound edges. For every outbound edge, we determine the associated probe, and we set the value of the transition function to the destination state of the outbound edge. For every probe that has no associated outbound edge from the current node, we set the value of the transition function to be the current state, meaning that the automaton stays in the same state.

Having initialized the transition function table, we initialize `HAS_VERDICT` to zero, and determine how to initialize the state variable[2]. Although we initially wanted to always initialize the state variable to 0, we have made it possible for users to set a specific starting state via the command line. This feature makes it easier to monitor a running system when the user knows something about the past behavior of the system before the monitoring script was started. The user-submitted starting state is represented in the script as a DTrace macro variable, for example `$2`: We set the relevant macro variable using the first available macro variable number when taking into account the macro variables used in probe specifications. We demonstrate this technique in chapter 6.

Having encoded the BEGIN-block, we need to encode probe clauses that update the state of the automaton when the associated probe fires. At first thought, it seems that we could simply write a probe clause for each probe-predicate expression that updates the state according to the transition function table, and then have three probe clauses associated with the `dtrace:::END` probe: One with a predicate checking if you are in an accepting state, one checking if you are in a rejecting state and a third for the inconclusive option. However, this would not make our monitor output a verdict as soon as possible: A verdict would only be given when DTrace is forced to quit or terminates because the program in question terminates.

To ensure that a verdict is given as soon as possible, we encode probe clauses that anticipate which state the monitor is *about to enter*. To implement this, we create three kinds of probe clauses: *accepting*, *rejecting* and *neutral* clauses. If the script enters an accepting clause, the script outputs `"ACCEPTED"` and terminates itself. Similarly, if the script enters a rejecting clause, the script outputs `"REJECTED"` and terminates itself. The

[2]We are aware that DTrace will always initialize global variables to 0, but we prefer to be explicit rather than implicit.

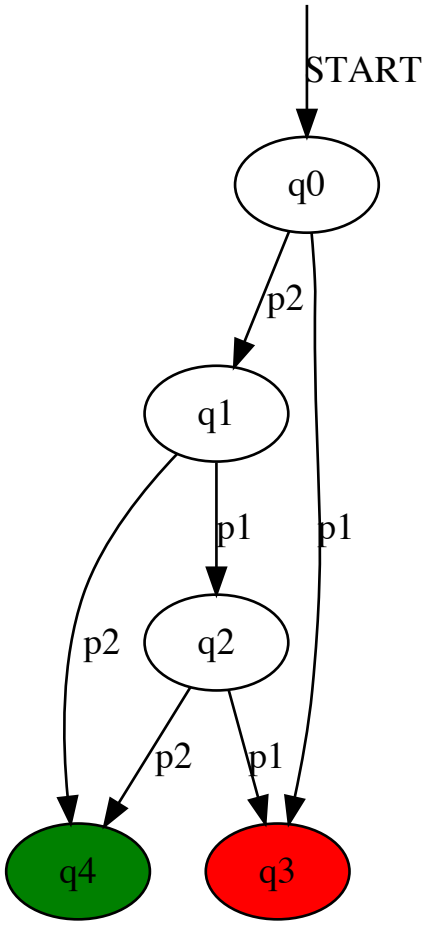

Figure 5.2: An example automaton. $p1$ and $p2$ are arbitrary DTrace probe specifications.

neutral clauses simply update the state variable according to the transition function table.

To create the accepting clauses, we need to know which state/probe combinations lead to accepting states. Similarly, we create the rejecting clauses by determining which state/probe combinations lead to rejecting states. We will explain the procedure we use to determine these clauses by way of an example.

Consider the automaton shown Figure 5.2. There is only one accepting state, $q4$, and we see that the automaton may enter $q4$ if the automaton is in $q2$ and the $p2$ event fires or if the automaton is in $q1$ and the $p2$ event fires. So in this case, $p2$ is the only probe firing that can take the automaton into an accepting state, and this happens if the automaton is in state $q2$ or $q1$. So for the automaton in Figure 5.2 we need to create one accepting clause on

the following general form:

```
p2
/((state == q2) || (state == q1))/
{
  printf("ACCEPTED");
  exit(0);
}
```

That is, if DTrace detects that $p2$ fires, it should check if the state is $q2$ or $q1$, and if so, it output `"ACCEPTED"` and stop the monitoring.

How do we create these clauses programmatically? We start by finding all accepting states. With `PyGraphviz`, we do this by iterating through the list of nodes in the graph and selecting those that have `fillcolor=green`. Then, we find all transitions leading into these accepting states. We do this by using the `in_edges` method provided by `PyGrahpviz`. We then look at the probe firings associated with these transitions: For every distinct probe, we create a list of source states from which the automaton would enter an accepting state if the probe in question fired. We store this probe to states-mapping in a dictionary, and use this dictionary to create accepting clauses: One for each distinct probe.

We carry out the exact same procedure for rejecting clauses, by finding all rejecting states (states with `fillcolor=red`), finding all transitions leading into these states and associating each distinct probe with a list of source states.

When we create the predicate that checks the current state of the automaton, we must also check if the user submitted some predicate as part of the probe specification. For example, if the user mapped $p2$ to `pid$target::empty:return/ arg1 == 1/`, we need to create a predicate which is a conjunction of the user-submitted predicate and the predicate we create to check the automaton's current state. To do this, we use a regular expression to detect if the user added a predicate to the current probe, extract the predicate and create the conjunction. Returning to the automaton shown in Figure 5.2, this would give us an accepting block on the following form:

```
pid$target::empty:return
/((arg1 == 1) &&
(state == 1 ||
state == 3))/
{
    trace("ACCEPTED");
    exit(0);
}
```

Obviously, not all transitions lead to accepting or rejecting states. We need some probe clauses for triggering these transitions, too. We do this by finding all states that are neither accepting nor rejecting, and carrying

```
#!/usr/sbin/dtrace -qs

dtrace:::BEGIN
{
    state = 0;
}

pid$target::p:entry
{
    state = 1;
}

pid$target::p:entry
/ state == 1/
{
    trace("F");
}
```

Figure 5.3: Even though the p function fires only once, the first probe clause will update the state variable, satisfying the predicate in the second probe clause before the second probe clause is processed.

out the same dictionary-building procedure as for accepting and rejecting clauses. These clauses will simply update the state of the automaton by doing a lookup in the transition function table.

We now have three clause groups: accepting, rejecting and neutral clauses. Due to a particularity in how DTrace processes incoming probe firings, it is essential that we place the accepting and rejecting clauses *before* the neutral clauses which update the state of the automaton. To see this, consider the D script shown in Figure 5.3. When DTrace detects that the `f` function is entered, it processes all the probe clauses associated with this event in the order the blocks occur in the script, from top to bottom. The first block will be executed, setting the state variable to 1. Then, the next block is executed. DTrace evaluates the predicate to true since the state variable is 1. However, if we intended the predicate to mean the state in which some automaton was in *before* the event occurred, we have created a misleading result.

Recall that the accepting and rejecting clauses always terminate the monitoring script. By entering an accepting or rejecting state, we have found a good or bad prefix, and we do not need any more information to make a verdict. Therefore, it doesn't matter if we put the accepting clauses before the rejecting clauses, or the other way around. What *does* matter is that we put the neutral clauses after the rejecting and accepting clauses.

If the monitoring script stops before it enters an accepting or rejecting state, the verdict should be `INCONCLUSIVE`. We enable this by adding one last

clause to the script: A clause listening to the `dtrace:::END` probe. This probe fires whenever DTrace stops. When this event occurs, we print `INCONCLUSIVE` before the script stops. However, there is one catch: When we forcefully quit DTrace in an accepting or rejecting clause, we trigger the `dtrace:::END` event. To avoid always printing `INCONCLUSIVE` when DTrace exits, we use the previously mentioned `HAS_VERDICT` variable as a guard. Before issuing `exit(0)`, an accepting or rejecting clause sets `HAS_VERDICT` to 1. By adding a predicate that checks if `HAS_VERDICT` is set to 1 before entering the inconclusive block, we ensure that `INCONCLUSIVE` is only printed on the right occasions.

Having assembled the complete script, we concatenate the pieces and print the script to `STDOUT`. A complete script for the automaton shown in Figure 5.2 is shown in appendix A.1, with extensive comments.

Chapter 6

Case study: 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* functions 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. The stack implementation is listed in its entirety in appendix B.1.

We have three goals with this case study:

1. To demonstrate how we can use the DTrace `pid` provider in conjunction with a `graphviz2dtrace`-generated D script to verify a program without leaving static artifacts in the code.
2. To demonstrate how we can attach the generated monitor to an already running process and set the monitor in a state based on what we know about the past behavior of the program.
3. To analyze the performance degradation we induce by attaching a DTrace-based monitor to the running system.

6.1 Creating the monitor

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 in [4, p.14:54] 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*.

In order to make a `graphviz2dtrace`-generated monitor for verifying that our stack program fulfills this property, we need to

1. Generate an automaton from the formula with `LamaConv`
2. Create a mapping between the atomic propositions in the formula (*push*, *empty* and *pop*) and DTrace probe and predicate expressions.
3. Feed the automaton and mapping to `graphviz2dtrace` to acquire the final monitor.

6.1.1 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 [5]:

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

The resulting automaton is illustrated in Figure 6.1, and the corresponding dot code is shown in Figure 6.2. 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.

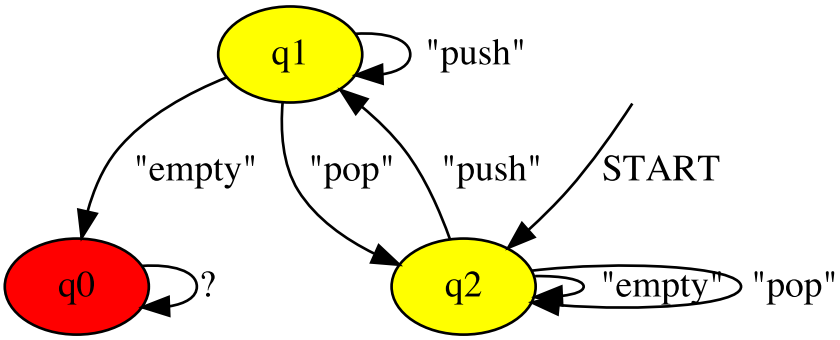

Figure 6.1: An automaton for the property $\square((\text{push} \wedge \Diamond\text{empty}) \rightarrow (\neg\text{empty}\ U\ \text{pop}))$

6.1.2 Mapping atomic propositions to DTrace probe and predicate expressions

With the automaton in hand, we map the atomic propositions the LTL formula (*push*, *pop* and *empty*) to DTrace probe and predicate expressions. In this investigation we want to use the `pid` provider (discussed in section **??**) to detect function calls within the program. The `pid` provider lets us

```
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]
}
```

Figure 6.2: The dot script for encoding $\square((\text{push} \wedge \Diamond\text{empty}) \rightarrow (\neg\text{empty}\ U\ \text{pop}))$

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[1]. We create the following mapping:

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

In this manner, 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.

For the *empty* event, we get a bit more sophisticated. The `empty` function is used in the stack program to determine if 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 stack 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 program our mapping in a JSON file like so:

```
{
    "empty": "pid$target::empty:return/arg1 == 1/",
    "pop": "pid$target::pop:return",
```

[1] In fact, the `pid` provider can be used to instrument "any instruction as specified by an absolute address or function offset" [12, p. 222] but in this thesis we restrict ourselves to using the `entry` and `return` probes.

```
    "push":"pid$target::push:entry"
}
```

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 complete monitor script is listed in appendix B.2. With the monitor in hand, we can now go on to the actual analysis.

6.2 Detecting a violation

Let us take a closer look at the implementation of the `push` function:

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

We observe that the push function does not increment the buffer index after pushing a new element onto the stack. In other words, the `empty` operation will yield 1 (ie. true) even though elements have been pushed onto the stack. We demonstrate this by feeding the program the following test case, which we store in the file `incite_error.in` and feed to the program:

```
PUSH 3
PUSH 4
PUSH 5
EMPTY
```

When running the program against this test case, we get the following interaction:

```
$ ./stack < incite_error.in
PUSHED 3
PUSHED 4
PUSHED 5
YES
```

Let us see if our generated script detects this violation. 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[2]

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

[2]Notice also that we run the program with `sudo`, as DTrace requires administrative privileges to run, regardless of the privilege level of the programs being monitored.

```
YES
REJECTED
```

Indeed, we see that the last line is `REJECTED`. To ensure against a false positive, we modify the stack implementation to increment on `push`, which should make the monitor output `INCONCLUSIVE`:

```
void push(int number, int * i)
{
    buffer[*i] = number;
    *i += 1;
}
```

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`.

6.3 Starting the monitor in a known state

In the previous section, we used the monitor to start the program being monitored by passing the `-c` flag to the monitoring script. What if the program we want to verify is already running?

Suppose that our stack program is already running and that we know that the stack has been pushed to before we decide to verify the program:

```
$ ./stack
PUSH 5
```

If we would start the monitor in its starting state, we would get misleading results. Looking back at the automaton in Figure 6.1, we would want the monitor to start in state *q1*, not *q2*.

To alleviate situations like these, `graphviz2dtrace` creates scripts that allow you to force the automaton into a given state by providing an extra integer argument on the command line. On initialization, DTrace checks if an extra argument is provided on the command line, and if so, it sets the state variable to the value of the command line argument. So to attach our monitor to the running program, we need two things:

1. The process ID of the process we want to monitor.
2. The number used to encode the specific state we want to start the monitor in.

We find the process ID of the running stack via the standard `ps` UNIX utility, and find the number used to encode the desired start state by inspecting the monitor script: Every `graphviz2dtrace`-generated monitor script comes with a comment showing how the state names used in the original automaton specification are mapped to integers in the transition function table in the monitor. In our current monitor (which is listed in its entirety in appendix B.2) it looks like this:

```
/*
q0 mapped to 1
q1 mapped to 2
q2 mapped to 0
*/
```

In a different terminal session from the one running the stack, we attach the monitor using the `-p` flag to the set `$target` macro variable in the script and pass the desired state (which is 2) as another command line argument:

```
$ ps ax | grep stack
15110 s005  S+     0:00.00 ./stack
15153 s006  U+     0:00.00 grep stack
$ sudo ./monitor.d -p 15110 2
```

Back in the terminal session where the stack program is running, we send the `EMPTY` command to the stack followed by an `EOF` signal to get a verdict:

```
$ ./stack
PUSH 5
PUSHED 5
EMPTY
YES
```

Immediately in the other terminal we get a verdict:

```
$ sudo ./monitor.d -p 15110 2
REJECTED
```

6.4 Performance evaluation

Having shown how to use the monitor, let us now evaluate the performance degradation we induce by attaching a monitor to the running system. In [16], DTrace expert Brendan Gregg 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:

	1k	10k	100k	1m	10m	100m
Uninstrumented						
1	0	0	0	0	0.07	0.6
2	0	0	0	0.01	0.07	0.6
3	0	0	0	0	0.06	0.59
4	0	0	0	0.01	0.07	0.6
5	0	0	0	0	0.06	0.6
6	0	0	0	0	0.06	0.6
7	0	0	0	0	0.07	0.61
8	0	0	0	0	0.07	0.62
9	0	0	0	0	0.07	0.61
10	0	0	0	0.01	0.07	0.59
Monitored with `pid`						
1	0.35	0.39	0.41	1.16	7.85	72.8
2	0.34	0.34	0.44	1.1	7.95	72.46
3	0.36	0.36	0.42	1.09	7.96	72.58
4	0.35	0.35	0.41	1.07	8.03	72.61
5	0.33	0.35	0.41	1.06	7.98	71.89
6	0.34	0.35	0.41	1.11	8.17	72.59
7	0.33	0.35	0.39	1.11	8.11	71.74
8	0.35	0.34	0.43	1.12	7.93	72.08
9	0.35	0.35	0.42	1.06	8.08	72.17
10	0.34	0.35	0.4	1.08	8.05	72.71

Table 6.1: Running times of stack program in seconds, as measured with `time` (using the `real` metric). A value of 0 means that the actual running time was too short for `time` to report.

- "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."

To investigate Gregg's claims, we follow his strategy of inducing as many probe firings per second as possible and measuring the increase in running time. In order to create as many probe event firings as possible, we replace the I/O loop in the stack with a loop that runs for a set number of iterations[3]. The new stack implementation is shown in its entirety in appendix B.3. We run the program with the UNIX `time` utility to check the running time. With the monitor attached, an invocation looks like this:

```
$ sudo time -lp ./monitor.d -c "./stack_xiter 1000" >> monitored_1k.txt
```

Even though `time` is too crude to give us a good picture of the running

[3]Note that `stack_xiter`, as all C programs in this thesis, is compiled with optimizations turned off.

	1k	10k	100k	1m	10m	100m
Uninstrumented						
1	0	0	0	0	0.07	0.6
2	0	0	0	0.01	0.07	0.6
3	0	0	0	0	0.06	0.59
4	0	0	0	0.01	0.07	0.6
5	0	0	0	0	0.06	0.6
6	0	0	0	0	0.06	0.6
7	0	0	0	0	0.07	0.61
8	0	0	0	0	0.07	0.62
9	0	0	0	0	0.07	0.61
10	0	0	0	0.01	0.07	0.59
Printf (user+sys)						
1	0	0	0.06	0.39	3.09	32.63
2	0	0	0.05	0.38	3.23	30.2
3	0	0	0.06	0.43	3.12	30.44
4	0	0.01	0.06	0.38	3.17	30.53
5	0	0.01	0.05	0.41	3.1	31.3
6	0	0	0.06	0.43	3.27	30.44
7	0	0	0.06	0.39	3.16	30.32
8	0	0	0.06	0.37	3.25	30.45
9	0	0	0.05	0.41	3.15	30.42
10	0	0.01	0.06	0.39	3.22	30.45

Table 6.2: Running times of stack with and without `printf` statements communicating to an external monitor, as measured with `time` (largest value of `real`,(`user+sys`)

times for iterations below a million, we see that the mere act of monitoring adds a constant upstart cost of about 350 milliseconds. As we increase the number of iterations to one million and beyond, we see that monitoring induces a slowdown factor of about one hundred.

Let us put these numbers in perspective: What if we instead of using our DTrace-based monitor inserted traditional print-statements to `STDERR` into the stack program, and piped the output to an external monitor written in C? Using the version of the stack program listed in appendix B.4 and the C monitor listed in appendix B.4.1, we get the results listed in Table 6.2.

As we can see, traditional printf-based instrumentation induces a slowdown factor of about a fifty: Half as bad as when monitoring using the `pid`-provider. However, we must remember two things: First of all, we have deliberately designed this test to induce as many events per second as possible to measure the *worst case* performance degradation induced by DTrace. It is not representative for the average or best case scenarios. Secondly, when using printf-based monitoring, we have to know which information to report a priori. With the `pid` provider, however, we always have

a way to investigate the running program, and we can turn off the monitor at any time to regain performance: The `printf` statements, however, must remain in the code and cause a permanent slowdown. Although DTrace costs twice as much as `printf`, the flexibility it gives is arguably worth the price.

As a final perspective, let us check if there is a noticeable difference between instrumenting the program using the `pid` provider and instrumenting the program by inserting static probes[4] that communicate with the DTrace framework into the stack program. The results are listed in Table 6.3. As is clear from the table, there is no enormous gap between using the `pid` provider and using static probes, although static probes are faster.

[4]The precise way of adding static probes to a program varies on different operating systems. A detailed example for Solaris-based systems is given in [17, p. 1051-1059]. On Mac OS X, the DTrace `man` page (`man dtrace`) gives a thorough example. For completeness, we have listed the stack program with static probes in appendix B.5.1, the probe specification script in B.5.2 and the generated header file in B.5.3.

	1k	10k	100k	1m	10m	100m
Uninstrumented						
1	0	0	0	0	0.07	0.6
2	0	0	0	0.01	0.07	0.6
3	0	0	0	0	0.06	0.59
4	0	0	0	0.01	0.07	0.6
5	0	0	0	0	0.06	0.6
6	0	0	0	0	0.06	0.6
7	0	0	0	0	0.07	0.61
8	0	0	0	0	0.07	0.62
9	0	0	0	0	0.07	0.61
10	0	0	0	0.01	0.07	0.59
Monitored with `pid`						
1	0.35	0.39	0.41	1.16	7.85	72.8
2	0.34	0.34	0.44	1.1	7.95	72.46
3	0.36	0.36	0.42	1.09	7.96	72.58
4	0.35	0.35	0.41	1.07	8.03	72.61
5	0.33	0.35	0.41	1.06	7.98	71.89
6	0.34	0.35	0.41	1.11	8.17	72.59
7	0.33	0.35	0.39	1.11	8.11	71.74
8	0.35	0.34	0.43	1.12	7.93	72.08
9	0.35	0.35	0.42	1.06	8.08	72.17
10	0.34	0.35	0.4	1.08	8.05	72.71
Monitored with static probes						
1	0.34	0.34	0.38	1.02	7.42	66.1
2	0.34	0.34	0.39	1.01	7.35	66.47
3	0.34	0.35	0.38	1.03	7.23	66.25
4	0.32	0.34	0.38	1.02	7.25	66.33
5	0.34	0.33	0.4	1.01	7.39	66.95
6	0.33	0.34	0.4	1.03	7.37	66.27
7	0.32	0.36	0.38	1.02	7.35	65.51
8	0.34	0.34	0.38	1.01	7.37	67.06
9	0.33	0.34	0.37	1	7.35	65.12
10	0.33	0.34	0.38	1.03	7.45	66.41

Table 6.3: Running times of stack with various instrumentation techniques as measured with `time` (largest value of `real`,(`user+sys`)

Chapter 7

Case study: 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 [14] talking to a PostgreSQL [18] 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 upon 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 finding relevant probes corresponding to the phenomena we want to study, specifying the property in LTL, attaching the monitor to the running system and using the monitor to derive a verdict.

In addition to demonstrating how we can use `graphviz2dtrace` to verify a deployed system, we use this case study to discuss two important limitations: How certain properties are very hard or impossible to verify with `graphviz2dtrace`, and how `graphviz2dtrace` based monitors are susceptible to race conditions.

The complete source code for the web server is listed in appendix C.1.2,

and the database schema used in listed in appendix C.2.

7.1 Finding relevant probes

Let us start by finding some relevant DTrace probes. Node.js comes with some useful static probes which it exposes via a dedicated `node` provider[1]. We can find the available probes by running

```
dtrace -lP "node*"
```

in the terminal while a node process is running. On our installation (the details of which are described in Appendix **??**) we find `http-server-request` and `http-server-response` (among others) which is relevant for our case. Furthermore, each node process will have its own provider, on the form `node<pid>`, much like every process on the system has its own `pid` provider. To restrict ourselves to probes associated with the web server in question, we can use the `$target` DTrace macro variable in the same in the same way we did in the previous investigation. In other words, for detecting when the server sends a response, we can use the probe specifier `node$target:::http-server-response`, and for detecting when the server receives an incoming request we can use `node$target:::http-server-request`.

The PostgreSQL Database Management System can also be compiled with a series of static DTrace probes. The full list of available probes is listed in [19]. Although most of these static probes are clearly aimed at PostgreSQL developers, some probes are useful for the end user. One such probe is `query-execute-done`, which fires whenever PostgreSQL is finished executing a query. We start our search for the specific probe to listen to by issuing

```
dtrace -ln "query-execute-done"
```

which lists all probes whose *name* component is `query-execute-done`. However, since PostgreSQL runs as a cluster of communicating processes, it is not immediately clear which probe to target, even if the database we target is the only one deployed on the computer: On our system, which has only one database, the command above yields fourteen distinct probes.

To find the correct probe, we run the following DTrace script to determine which PostgreSQL probe to listen to:

```
#!/usr/sbin/dtrace -qs

postgresql*:::*:query-parse-start
{
```

[1]The Node.js documentation is very sparse on this topic, but Node.js seems to compile with DTrace probes included by default.

```
    printf("%s:%s:%s:%s: fired\n",
            probeprov, probemod, probefunc, probename);

    printf("pid %d , tid %d, query: %s, \n",
            pid, tid, copyinstr(arg0));
}
postgresql*::*:query-execute-done
{
    printf("%s:%s:%s:%s: fired\n",
            probeprov, probemod, probefunc, probename);

    printf("pid %d , tid %d, is done executing query.\n", pid, tid);
}
```

In this script, we use the `query-parse-start`, a static PostgreSQL probe which fires whenever a query is being parsed and exposes the incoming query to DTrace as a probe argument [19]. We use the built-in special variables , `pid` and `tid` to print exactly which process and thread is carrying out the query, and the DTrace keywords `probeprov`, `probemod`, `probefunc`, and `probename` give us the exact probe `<provder:module:function:name>` specification to use for specifying this event.

In order for DTrace to read the query string we use the built-in function `copyinstr` that copies strings from userland and into the DTrace framework in the kernel. Under the assumption that there is only one database on the system, we can determine the relevant probe by sending an HTTP request to the web server while this script is running.

We send an HTTP request to the server and the script outputs the following:

```
postgresql19053:postgres:pg_parse_query:query-parse-start: fired
pid 19053 , tid 277582, query: INSERT INTO entries(entry) VALUES($1),
postgresql19053:postgres:PortalRunMulti:query-execute-done: fired
pid 19053 , tid 277582, is done executing query.
postgresql19053:postgres:PortalRun:query-execute-done: fired
pid 19053 , tid 277582, is done executing query.
```

We recognize the query being sent to PostgreSQL's parsing module as the same query used in the web server, and since the process and thread IDs are the same for all the firing probes, we can safely assume that we have found the right process. However, we also observe that the `query-execute-done` event has been triggered by two separate functions. Since the web server is only running on our local machine, we know that it parsed only one request and we will need to decide on one of the functions to listen to: If not we will have two query probe firings for every corresponding request-response event pair. It turns out that in the PostgreSQL source code, `PortalRun` calls `PortalRunMulti` to process the query[2]. We therefore specify that we are

[2]We determined this by inspecting the PostgreSQL source code listing on http://doxygen.postgresql.org/pquery_8c_source.html#l00813.

interested in the case where `PortalRun` reports `query-execute-done`.

To make our script general, we would like to be able to provide the PostgreSQL process ID to the script on the command line. To this end, we deploy a macro variable. The resulting three probes are then

1. `node$target:node::http-server-request`,
2. `node$target:node::http-server-response` and
3. `postgresql$$1:postgres:PortalRun:query-execute-done`.

7.2 Specifying the property

Having verified that we have relevant probe events to reason about, we can turn to specifying the property. For simplicity, we will use the symbols `req`, `res` and `query` instead of the probes found in the previous section, with the understanding that the we will substitute these symbols for actual probes when we produce the monitor script with `graphviz2dtrace`.

We started out by wanting to check that for every incoming request to the web server, the database completes the corresponding query before the web server sends a response. We start by making the requirements more specific:

1. If a request event happens, eventually a response event must happen.
2. If a request happens, eventually a query event must happen.
3. The query event must come before the response event.

We recognize the first and second as *response*-properties, and the third as a *precedence* property, and derive the three corresponding LTL formulas using the well-known specification patterns in [1]:

1. $\text{req} \rightarrow \Diamond \text{res}$
2. $\text{req} \rightarrow \Diamond \text{query}$
3. $\neg \text{res}\ W\ \text{query}$

When we take the conjunction of these three formulas, `LamaConv` produces the automaton shown in Figure 7.1.

We quickly discover that something is odd with the generated automaton. First of all, we observe that the automaton will go into an accepting state if it sees a query event *before* a request event has happened. We want the query event to happen after a request. We therefore need to strengthen our formula to ensure that the events in question happen in the desired order. We add the following constraints:

4. $\neg \text{query}\ W\ \text{req}$
5. $\neg \text{res}\ W\ \text{req}$

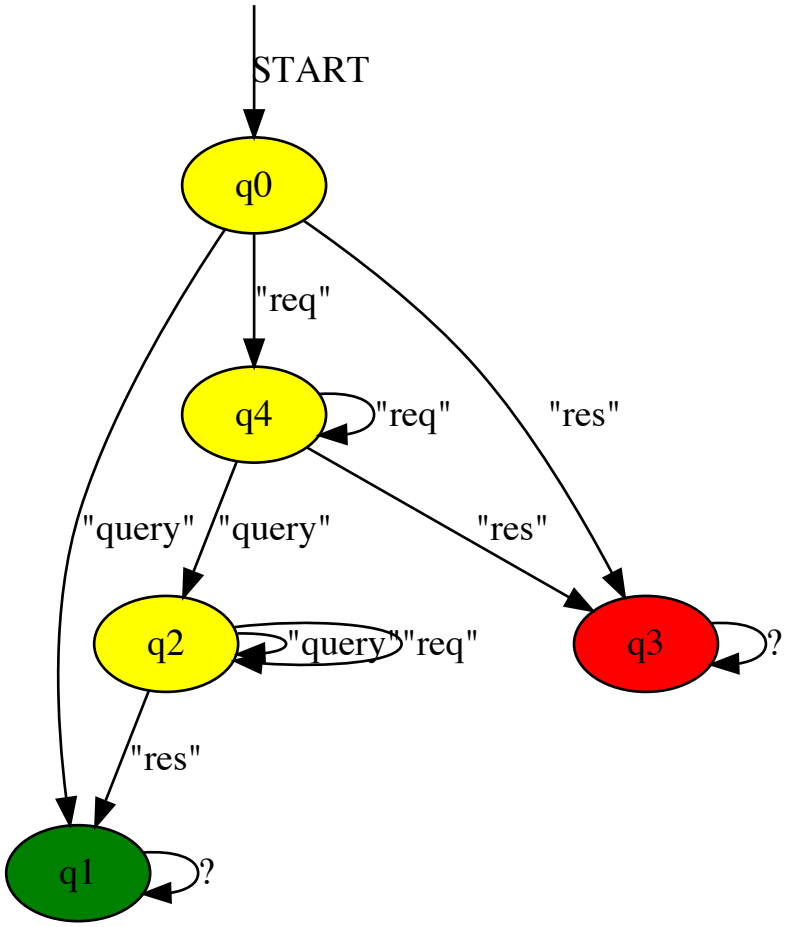

Figure 7.1: An automaton for $(\text{req} \rightarrow \Diamond\text{res}) \wedge (\text{req} \rightarrow \Diamond\text{query}) \wedge (\neg\text{res}\ W\ \text{query})$

We now have proper transitivity on the events. We find the conjunction of the five criteria and deduce the monitor shown in Figure 7.2.

While this is certainly an improvement, it does not fully capture our intended meaning. First of all, on reflection, it becomes clear that we do not want to say that the property is satisfied as soon as we have seen *one* valid sequence containing a request, query and response: The monitor accepts strange traces such as

$$\texttt{req} \rightarrow \texttt{req} \rightarrow \texttt{req} \rightarrow \texttt{query} \rightarrow \texttt{res}$$

The fundamental problem is that we would like to have a way of reasoning about *distinct* query, request and response events. While there exists forms of *parametric* Linear Temporal Logic (for example in [32]) to express such properties, we cannot use `graphviz2dtrace` to produce monitors that faithfully represent these. In the next section, we investigate how we can approach the problem within the limits of `graphviz2dtrace`.

7.2.1 Using counters and DTrace predicates

As we described in the previous section, `graphviz2dtrace` is limited by not supporting parameterized properties, which makes it hard to talk 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. We reasoned that this property is a combination of response and precedence properties. 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, like so:

```
#!/usr/sbin/dtrace -qs

int nrequests, nresponses, nqueries;

node$target:node::http-server-request
{
```

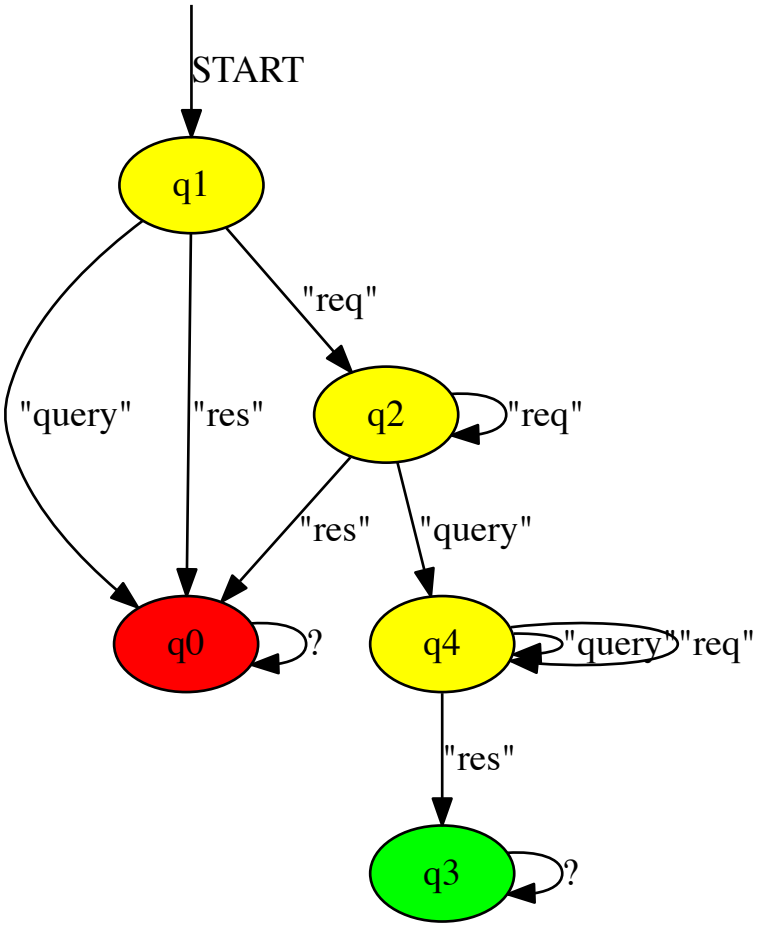

Figure 7.2: An automaton for $(\text{req} \rightarrow \Diamond\text{res}) \wedge (\text{req} \rightarrow \Diamond\text{query}) \wedge (\neg\text{res}\ W\ \text{query}) \wedge (\neg\text{query}\ W\ \text{req}) \wedge (\neg\text{res}\ W\ \text{req})$

```
    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, it is very important that we place these probe clauses *before* the probe clauses related to the automaton logic to get the intended result, for the reasons we discussed in section 5.2. Note also that global variables like the ones we use here are initialized to 0 by default in D script [12, p. 53].

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:

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

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 [12, 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 decide to check the property 10 times a second, the probe and predicate specification becomes:

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

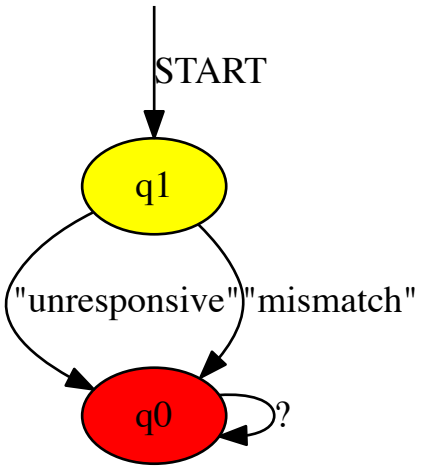

Figure 7.3: An automaton for $(\Box\neg\text{mismatch}) \wedge (\Box\neg\text{unresponsive})$

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 shown in Figure 7.3[3]. We use `graphviz2dtrace` to create the corresponding script, with a few manual amendments to get the counters to work. The complete monitor is shown in appendix C.3.

On concurrency

Before we move on to using the monitor, we must address the proverbial elephant in the room: Concurrency. Since the Node.js web server and the

[3]To get this automaton from `LamaConv`, we had to set the alphabet to `[mismatch, unresponsive, other]`. If we had not added the `other` symbol, an automaton with a single wildcard edge from a start state to a rejecting state would have been generated, which `graphviz2dtrace` cannot understand. We therefore generate the automaton with `other` in the alphabet and remove the resulting `other` looping edge on the start node to obtain the final automata.

PostgreSQL database run as separate processes on a multi-core machine it is both possible and desirable that they do tasks in parallel: This can also mean that we get two simultaneous probe firings that create a race condition on the monitor's `state` variable. This is the most fundamental limitation in `graphviz2dtrace`-generated scripts, and we must always ask ourselves if the `state` variable can be corrupted in our derived monitors.

In this specific case, we are in the clear: The clauses in the program never update the `state` variable, so no race condition occurs. This is an exception to the rule: In general, `graphviz2dtrace` cannot guarantee against race conditions when using probes that can fire simultaneously.

We return to the question of concurrency in the section 8.3, where we indicate some ways one can use DTrace to verify concurrent programs.

7.3 Detecting a violation

With the monitoring script in hand, let us proceed to verifying the system under scrutiny. The Node.js server prints the following on startup:

```
$ node server.js
Server running at 127.0.0.1, port 1337
Node.js PID is 11509
PostgreSQL client PID is 11510
```

Since the Node.js server reports its process ID and the process ID of the attached PostgreSQL client on startup, we can attach our monitor to the system under scrutiny by sending the process IDs via the command line. A representative invocation looks like the following:

```
$ 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 [15] tool `ab` to send the server a series of requests:

```
$ ab -n 10000 http://127.0.0.1:1337/
```

Almost immediately, the monitor responds with

```
REJECTED DUE TO MISMATCH
```

To understand why this error happens, we must take a closer look at the web server source code. Consider the following fragment:

```
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');
```

In this code, the method `client.query` is called with three parameters: An SQL query string, a set of parameters to populate the SQL query and anonymous callback function which will be called when the results from the database are made available to the web server. Used correctly, this programming pattern makes the webserver more efficient, as it does not have to busily wait for the database to finish: Once the database finishes, the runtime executes the code in the anonymous callback function[4]. In the meantime, the webserver can go on processing other events.

The error we observe is due to the fact that 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.

We fix the error by always closing the HTTP response in the callback associated with the PostgreSQL query:

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

When we restart the web server and run the monitor again on the same benchmark, no violation is reported.

Let us see if we can detect a violation of the response property, too. We run a new benchmark on the server after fixing the callback error discovered above, this time with the number of concurrent connections set to 200:

```
$ ab -n 10000 -c 200 http://127.0.0.1:1337/
```

Immediately, we get

```
REJECTED DUE TO UNRESPONSIVENESS
```

The tolerance gap of 100 requests was chosen arbitrarily, so this does not have to mean that there is any grave error with the software system as such.

[4] A full treatise on the concurrency and I/O model underlying Node.js and JavaScript is beyond the scope of this thesis. For the interested reader, a good resource on the JavaScript Event Loop is [26]. For Node.js specifically, a good starting point is [13].

		C=1	C=50	C=100
Standard				
	1	1237.75	1976.26	2040.91
	2	1303.61	2008.46	2052.25
	3	1310.54	2058.02	2054.21
	4	1306.25	1957.03	2035.86
	5	1319.98	2017.99	2088.38
	6	1324.14	2016.91	2089.08
	7	1284.68	2004.85	2077.76
	8	1327.21	2045.01	2076.72
	9	1314.30	2047.57	2084.79
	10	1316.82	2038.44	1906.94
Monitored				
	1	1165.37	2082.67	1989.40
	2	1184.10	2103.73	2063.09
	3	1221.96	2078.91	1994.72
	4	1243.81	2061.58	2003.81
	5	1235.49	1924.42	2038.09
	6	1240.24	2168.59	2051.90
	7	1242.31	2059.75	2038.34
	8	1249.34	2033.45	2033.45
	9	1231.39	1756.83	2040.44
	10	1242.06	2068.34	2002.91

Table 7.1: Mean processed requests per second for `C` concurrent requests as measured by `ab`.

Nevertheless, we have seen that the monitor detects a violation.

7.4 Performance evaluation

How much does the web server degrade in performance when the monitor is attached? To investigate this, we use the Apache Benchmark tool `ab` [15] to run a series of performance tests. For both the case where the web server is being monitored and for the case where it is not, we run the `ab` command ten times, with an interval of a minute between each invocation. We make `ab` send 10 000 requests, and check three different concurrency levels: 1, 50 and 100 concurrent requests (if we go up to 200, the monitor will be violated, as we saw in the previous section). The results are shown in table 7.1. Keep in mind that all software runs on the same computer, so the fast response times reflect the fact that the queries do not go over the network.

The first thing we observe is that there is no substantial difference between the standard and monitored version of the software when the server is taking in more than one connection at a time. For these runs, there is no

clearly observable degradation. If we recall Brendan Gregg's principles from the last performance investigation, this should not surprise us: We expect to only notice a significant degradation after 10 000 events per second. With the system processing 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. 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.

However, this raises the question: How can we know ahead of time how big the probe firing rate is? We conclude this chapter by demonstrating a DTrace script for sampling the probe firing rate.

7.4.1 Sampling the probe firing rate

We can use a script like the following to sample the probe firing rate:

```
#!/usr/sbin/dtrace -qs

node$target:node::http-server-request,
node$target:node::http-server-response,
postgresql$$1:postgres:PortalRun:query-execute-done
{
    @agg["firings"] = count()
}

tick-1
{
    printa(@agg);
    exit(0);
}
```

In this script, we have listed the relevant probes we want to listen to, and we use a DTrace *aggregation* [12, p. 89-104] to count the number of probe firings. The aggregation stores information in per CPU-buffers. When an aggregation is printed, the count from the CPU-specific buffers are coalesced into a final count. Furthermore, we us the `tick` provider to print the count and terminate the script after one second.

When attaching this script to the web server and database system while running the `ab` tool with 10 000 requests at 100 concurrent connections, we get

```
$ sudo ./check_firingrate.d -p 8379 8380

  firings                                          5632
```

Armed with scripts like this and the insight that a noticeable performance degradation typically occurs when the probe firing rate exceeds 10 000, we can make an informed judgement about the cost of instrumentation before we attach a monitor. This is especially helpful if we are not too familiar with the probes we have chosen.

Chapter 8

Evaluation

8.1 Revisiting the design decisions

Having used `graphviz2dtrace` to verify two separate software systems, let us now look back on the design decisions we made when developing `graphviz2dtrace`:

1. *Associate atomic propositions in LTL specifications with DTrace probe specifications* (with optional predicates).
2. *Encode both the trace generation and monitor logic in one script.*
3. *Use a global variable to keep track of the automaton state, even though this means that you cannot safely verify properties dealing with potentially overlapping events.*

In the following, we evaluate these design decisions in turn based on what we observed and learned in the case studies.

Let us start with the decision of associating atomic LTL propositions with DTrace probe specifications. In the first case study (the one involving the stack), it was fairly straightforward to associate the atomic propositions in our specification formula with corresponding DTrace probe specifications. However, in the second case study we quickly saw that we needed to get quite sophisticated in our DTrace predicates to express our intended property, and we had to make manual amendments to the `graphviz2dtrace`-generated script to initialize and update the variables used in the predicate expressions: So while the *idea* of associating atomic propositions with probe specifications is promising, the *implementation* of this idea in `graphviz2dtrace` leaves a bit to be desired, as the scripts must be manually amended to get the intended effect.

For the cases where the predicates reason about counters (as we did in section 7.2.1), a valuable improvement in this regard would be to let users specify a mapping between counter variable names and DTrace probes, indicating that the counter variable in question should be incremented when

the corresponding probe fires. Given such a mapping, `graphviz2dtrace` could insert the necessary variable initialization statements and probe clauses to generate scripts like the one we manually created in 7.2.1. On the other hand, one might argue that since DTrace predicate expression can be arbitrarily complex, it is better to let the end user make the necessary amendments to the script and make `graphviz2dtrace` responsible for one thing only: Encoding the automaton.

It is not entirely obvious how the case studies illuminate the design decision of *encoding both the trace generation and monitor logic in one script*. To clarify, the design decision concerns whether we use DTrace to produce a trace which we send to an *external* program for monitoring, or whether *one* program collects data and makes a verdict about the gathered data. Our tool, `graphviz2dtrace`, creates scripts that handle *both* of these concerns in the same script. The advantage of this is that you avoid the performance overhead of having to send data from DTrace to another program: The disadvantage is that you constrain yourself to the limitations of the D programming language – notably the fact that there is no way of implementing a globally synchronized state variable for the automaton. Our third decision – *using a global variable to track the automaton state* – was to accept this fact. In the first case study, this posed no problem for us as we analyzed a single-threaded, single-process system where the events were guaranteed not to overlap. In the second case study, however, we had to address this problem by ensuring that we used an automaton for which no race condition on the state variable could occur.

This raises a dilemma of where to draw the line between the responsibilities of the *tool* and the responsibilities of the *user*. Arguably, there is great value in being able to freely choose among the thousands of unique probes made available by DTrace. Also, as we saw in the second case study where we used DTrace to distinguish between probe firings from `PortalRun` and `PortalRunMulti` within PostgreSQL, we always have to be very careful in specifying exactly the probe firings we want. Is it then too much to ask that users also understand which probes can fire in parallel?

Nevertheless, the possible race condition on the state variable is the most significant limitation of `graphviz2dtrace`. In section 8.3, we discuss other ways doing Runtime Verification with DTrace that avoids this problem.

8.2 Performance Impact

We started out by investigating a principle about DTrace overhead suggested by DTrace expert Brendan Gregg in [16]: That overhead is proportional to the rate at which probes fire. We found this principle to be correct in the first case study, where we deliberately tried to induce as many probe firings per second as possible. However, we also put this overhead into perspective by running a similar test for a program that uses

verification via traditional print statements, and found that DTrace was only twice as bad. So the tremendous increase in running time that we observed has more to do with the practical limits of software observability than DTrace in particular.

In the second case study, where we subjected a software system to varying degrees of load using Apache Benchmark, we saw that the act of monitoring induced no significant effect. Our monitor produced roughly 6000 probe firings per second, which is well below what Gregg suggests is the threshold of noticeable degradation: 10 000 events per second [16]. We also demonstrated how to write a DTrace script for sampling the probe firing rate on a given system, which provides a way of understanding the possible performance penalty before attaching the monitor.

8.3 Suggestion for future work: Separate trace generation from monitoring

Even though `graphviz2dtrace` cannot produce monitors that are safe against race conditions, there is an alternative way of using DTrace for Runtime Verification which avoids these problems. The trick is to separate the act of *producing a trace* from the act of *verifying if the trace satisfies a property*, and only use DTrace for the former. To demonstrate, consider the following DTrace script for instrumenting the web server and database in case study two:

```
#!/usr/sbin/dtrace -qs

#pragma D option switchrate=10hz
#pragma D option temporal

node$target:node::http-server-request
{
    printf("(%u,%d,%d,%d): %s\n", walltimestamp, cpu, pid, tid,
            "node$target:node::http-server-request");
}

node$target:node::http-server-response
{
    printf("(%u,%d,%d,%d): %s\n", walltimestamp, cpu, pid, tid,
            "node$target:node::http-server-response");
}

postgresql$$1:postgres:PortalRun:query-execute-done
{
    printf("(%u,%d,%d,%d): %s\n", walltimestamp, cpu, pid, tid,
            "postgresql$$1:postgres:PortalRun:query-execute-done");
}
```

The overall strategy in this script is to produce a series of print statements that indicate what probe firing happened, at what precise time, on which CPU and in which process and thread. The first statement tells DTrace to copy data from its in-kernel per-CPU buffers up to the DTrace consumer at a 10 hertz (10 times per second). The standard option is to copy the buffered data once per second: This settings ensures that the monitor which receives the incoming trace gets the trace faster.

The second statement, `pragma D option temporal`, ensures that DTrace sorts its data in chronological order before uploading it to the userland consumer. Since DTrace keeps separate buffers *per* CPU [12, p. 127-133], the standard behavior is to simply upload the buffer contents associated with each CPU, one CPU at a time. This means that you can get traces that are not sorted chronologically. The `temporal` option makes DTrace present the buffer contents from the different CPUs in chronological order. While we could let the monitor sort the trace by using the timestamps provided in the trace, this option makes it easier to create incremental monitors: Monitors that update their state on every incoming event.

Using a script like this, one can use a UNIX pipe to send the data to an external monitor which makes a verdict from the trace. Since the monitor is a separate process that receives the incoming trace line by line, there is no danger of a race condition. The monitor can also be a lot more sophisticated since it can be written in a general-purpose programming language rather than D. The downside, however, is that the communication between DTrace and the monitor will create more overhead than in a solution like `graphviz2dtrace`.

Chapter 9

Conclusion

In this thesis, we have presented `graphviz2dtrace`, a tool for creating DTrace-based verification scripts based on LTL_3 monitor automata encoded in graphviz dot notation. We used it in conjunction with `LamaConv` to verify two software systems at runtime: A single process stack implementation written in C, and a web server written in Node.js communicating with a PostgreSQL database. We demonstrated how to detect property violations in both systems, and analyzed the performance penalty induced by the act of monitoring. For the case concerning the stack implementation, we confirmed a set of principles laid out by DTrace expert Brendan Gregg in [16]: While a noticeable performance degradation occurs when the rate of DTrace probe firings exceed 10 000 per second, one can meaningfully instrument many systems before exceeding this threshold.

Furthermore, we looked at two inherent limitations with `graphviz2dtrace`-based runtime verification: That the resulting monitor scripts cannot guarantee against race conditions on the embedded automaton's state, and that it is very difficult to reason about *distinct* events of the *same* type when the foundational idea in `graphviz2dtrace` is to map atomic propositions in LTL formulas to DTrace probe specifications. For the case study concerning the web server and database, we demonstrated a strategy for coping with these limitations using counter variables and DTrace predicates. For future work, we also suggested a technique for safely instrumenting concurrent systems: By using DTrace to only produce a trace for an *external* monitor to verify, the inherent concurrency limitations of the D programming language can be avoided.

Appendix A

Example scripts

A.1 A generic `graphviz2dtrace`-generated script

```
#!/usr/sbin/dtrace -qs

/* The line above is the standard shebang used for creating DTrace scripts,
 * with one added option: We add the -q flag to make DTrace only print what we
 * explicitly print in this script */

/* We use this option to automatically intialize all macro
 * variables to zero. We need all macro variables to be initialized
 * to make the trick where we initialize the state variable with a ternary
 * operator work.
 */
#pragma D option defaultargs

/* What follows is an automatically generated script signarture... */

/*
This script was automatically generated by graphviz2dtrace version 0.1
on 2016-05-10 09:51:37.112061 UTC. To run it, make it executable with

chmod +ux

and then run it with root privileges. Example:

sudo myscript.d -c program-to-be-monitored
*/

/*
q0 mapped to 0
q1 mapped to 1
q2 mapped to 3
```

```
q3 mapped to 2
q4 mapped to 4
*/

/*
p1 mapped to 1
p2 mapped to 0
*/

int HAS_VERDICT;
int state;
int tf[5][2];

dtrace:::BEGIN
{
        tf[0][0] = 1;
        tf[0][1] = 2;
        tf[1][0] = 4;
        tf[1][1] = 3;
        tf[2][0] = 2;
        tf[2][1] = 2;
        tf[3][0] = 4;
        tf[3][1] = 2;
        tf[4][0] = 4;
        tf[4][1] = 4;
        HAS_VERDICT = 0;

    /* Since none of the probes in this script use macro variables,
     * the first available macro variable is $1. We check if $1
     * is set: If so we set state to the user-provided value.
     * If not, we set it to zero (ie. the start state).
     */
        state = ($1 ? $1: 0);
}

/* This is the first and only 'accepting block' in this script */
p2
/state == 1 ||
state == 3/
{
        trace("ACCEPTED");
        HAS_VERDICT = 1;
        exit(0);
}

/* This is the first and only 'rejecting block' in this script */
p1
/state == 0 ||
```

```
state == 3/
{
        trace("REJECTED");
        HAS_VERDICT = 1;
        exit(0);
}

/* following two blocks are 'neutral' blocks which simply update the state of the automat
p2
/state == 0/
{
        state = tf[state][0];
}

p1
/state == 1/
{
        state = tf[state][1];
}

/* Finally, we have a block for detecting an inconclusive trace.
 * We check that the dtrace:::END event was not caused by an accepting
 * or rejecting block, and if so, we print out "INCONCLUSIVE".
 * */
dtrace:::END
/ !HAS_VERDICT /
{
        trace("INCONCLUSIVE");
}
```

Appendix B

Code for the stack case study

B.1 The faulty stack implementation

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #define COMMAND_BUF_LEN 16
5  #define STACK_BUFF_LEN 100
6
7  int buffer[STACK_BUFF_LEN];
8
9  void push(int number, int * i)
10 {
11     buffer[*i] = number;
12 }
13
14 int pop(int * i)
15 {
16     int result = buffer[*i];
17     *i -= 1;
18     return result;
19 }
20
21 int empty(int * i)
22 {
23     return *i == 0;
24 }
25
26 int full(int * i)
27 {
28     return *i == 100;
29 }
30
```

```
31 int main(void)
32 {
33     int index, number, retval;
34     char command[COMMAND_BUF_LEN];
35 
36     retval = scanf("%s %d\n", command, &number);
37 
38     index = 0;
39 
40     while(retval != EOF){
41 
42         if(strncmp(command, "PUSH", COMMAND_BUF_LEN) == 0){
43             push(number, &index);
44             printf("PUSHED %d\n", number);
45         }
46 
47         if(strncmp(command, "POP", COMMAND_BUF_LEN) == 0)
48             printf("%d\n", pop(&index));
49 
50         if(strncmp(command, "EMPTY", COMMAND_BUF_LEN) == 0)
51             printf("%s\n", empty(&index) ? "YES" : "NO");
52 
53         if(strncmp(command, "FULL", COMMAND_BUF_LEN) == 0)
54             printf("%s\n", full(&index) ? "YES" : "NO");
55 
56         retval = scanf("%s %d\n", command, &number);
57     }
58 
59     return 0;
60 }
```

B.2 The generated monitor

```
#!/usr/sbin/dtrace -qs

#pragma D option defaultargs

/*
This script was automatically generated by graphviz2dtrace version 0.1
on 2016-05-01 10:22:16.599448 UTC. To run it, make it executable with

chmod +ux

and then run it with root privileges. Example:

sudo myscript.d -c program-to-be-monitored
```

```
*/

/*
q0 mapped to 1
q1 mapped to 2
q2 mapped to 0
*/

/*
pid$target::empty:return/arg1 == 1/ mapped to 0
pid$target::pop:return mapped to 2
pid$target::push:entry mapped to 1
*/

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

dtrace:::BEGIN
{
        tf[0][0] = 0;
        tf[0][1] = 2;
        tf[0][2] = 0;
        tf[1][0] = 1;
        tf[1][1] = 1;
        tf[1][2] = 1;
        tf[2][0] = 1;
        tf[2][1] = 2;
        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");
}
```

B.3 Stack running x iterations

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STACK_BUFF_LEN 100

int buffer[STACK_BUFF_LEN];

void push(int number, unsigned int * i)
{
    buffer[*i] = number;
    *i += 1;
}

int pop(unsigned int * i)
{
    int result = buffer[*i];
    *i -= 1;
    return result;
}

int empty(unsigned int * i)
{
    return *i == 0;
}
```

```
int full(unsigned int * i)
{
    return *i == 100;
}

int main(int argc, char *argv[])
{
    unsigned int i, index, iterations, pushes, pops, result;

    if(argc < 2){
        iterations = 100;
    } else {
        iterations = atoi(argv[1]);
    }

    index = 0;
    pushes = 0;
    pops = 0;
    result = 0;

    for(i = 0; i < iterations; i++){
        if(i % 2 == 0){
            push(1, &index);
            pushes++;
        }
        else{
            result += (pop(&index) * i % 7);
            pops++;
        }
    }

    printf("pushes: %u, pops: %u, result: %d\n", pushes, pops, result);
    return 0;
}
```

B.4 Stack with `printf` statements

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STACK_BUFF_LEN 100

int buffer[STACK_BUFF_LEN];
```

```
void push(int number, unsigned int * i)
{
    printf("PUSH %d\n", number);
    buffer[*i] = number;
    *i += 1;
}

int pop(unsigned int * i)
{
    printf("POP %d\n", i);
    int result = buffer[*i];
    *i -= 1;
    return result;
}

int empty(unsigned int * i)
{
    printf("EMPTY %d\n", (*i == 0));
    return *i == 0;
}

int full(unsigned int * i)
{
    return *i == 100;
}

int main(int argc, char *argv[])
{
    unsigned int i, index, iterations, pushes, pops, result;

    if(argc < 2){
        iterations = 100;
    } else {
        iterations = atoi(argv[1]);
    }

    index = 0;
    pushes = 0;
    pops = 0;
    result = 0;

    for(i = 0; i < iterations; i++){
        if(i % 2 == 0){
            push(1, &index);
            pushes++;
        }
        else{
            result += (pop(&index) * i % 7);
```

```
            pops++;
        }
    }

    fprintf(stderr, "pushes: %u, pops: %u, result: %d\n", pushes, pops, result);
    return 0;
}
```

B.4.1 Monitor for stack with `printf` statements

```
#include <stdio.h>
#include <string.h>

#define PUSH 0
#define POP 1
#define EMPTY 2

#define COMMAND_BUF_LEN 16

int main(void)
{
    int tf[3][3];
    char command[COMMAND_BUF_LEN];

    tf[0][PUSH] = 0;
    tf[0][POP] = 0;
    tf[0][EMPTY] = 0;

    tf[1][PUSH] = 1;
    tf[1][POP] = 2;
    tf[1][EMPTY] = 0;

    tf[2][PUSH] = 1;
    tf[2][POP] = 2;
    tf[2][EMPTY] = 2;


    int arg, retval, state = 2;

    retval = scanf("%s %d\n", command, &arg);

    while(retval != EOF){

        if(strncmp(command, "POP", COMMAND_BUF_LEN) == 0){
            state = tf[state][POP];
        }
```

```
        if(strncmp(command, "PUSH", COMMAND_BUF_LEN) == 0){
            state = tf[state][PUSH];
        }

        if((strncmp(command, "EMPTY", COMMAND_BUF_LEN) == 0) && arg == 1){
            state = tf[state][EMPTY];
        }

        retval = scanf("%s %d\n", command, &arg);
    }

    switch(state){
        case 0:
            printf("REJECTED\n");
            break;
        default:
            printf("INCONCLUSIVE\n");
            break;
    }

    return 0;
}
```

B.5 Stack with static probes

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/sdt.h>
#include "probes.h"

#define STACK_BUFF_LEN 100

int buffer[STACK_BUFF_LEN];

void push(int number, unsigned int * i)
{
    /* static probe firing */
    STACK_PUSH(number, i);
    buffer[*i] = number;
    *i += 1;
}

int pop(unsigned int * i)
{
    /* static probe firing */
```

```
    STACK_POP(i);
    int result = buffer[*i];
    *i -= 1;
    return result;
}

int empty(unsigned int * i)
{
    /* static probe firing */
    STACK_EMPTY(*i == 0);
    return *i == 0;
}

int full(unsigned int * i)
{
    /* static probe firing */
    STACK_FULL(i);
    return *i == 100;
}

int main(int argc, char *argv[])
{
    unsigned int i, index, iterations, pushes, pops, result;

    if(argc < 2){
        iterations = 100;
    } else {
        iterations = atoi(argv[1]);
    }

    index = 0;
    pushes = 0;
    pops = 0;
    result = 0;

    for(i = 0; i < iterations; i++){
        if(i % 2 == 0){
            push(1, &index);
            pushes++;
        }
        else{
            result += (pop(&index) * i % 7);
            pops++;
        }
    }

    printf("pushes: %u, pops: %u, result: %d\n", pushes, pops, result);
    return 0;
```

```
}
```

B.5.1 Monitor for stack with static probes

```
#!/usr/sbin/dtrace -qs

#pragma D option defaultargs

/*
This script was automatically generated by graphviz2dtrace version 0.1
on 2016-05-01 23:17:26.517546 UTC. To run it, make it executable with

chmod +ux

and then run it with root privileges. Example:

sudo myscript.d -c program-to-be-monitored
*/

/*
q0 mapped to 1
q1 mapped to 2
q2 mapped to 0
*/

/*
stack$target:::empty/arg0 == 1/ mapped to 0
stack$target:::pop mapped to 1
stack$target:::push mapped to 2
*/

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

dtrace:::BEGIN
{
        tf[0][0] = 0;
        tf[0][1] = 0;
        tf[0][2] = 2;
        tf[1][0] = 1;
        tf[1][1] = 1;
        tf[1][2] = 1;
        tf[2][0] = 1;
        tf[2][1] = 0;
        tf[2][2] = 2;
        HAS_VERDICT = 0;
```

```
        state = ($1 ? $1: 0);
}

stack$target:::empty
/ (arg0 == 1) && (state == 2)/
{
        trace("REJECTED");
        HAS_VERDICT = 1;
        exit(0);
}

stack$target:::empty
/ (arg0 == 1) && (state == 0)/
{
        state = tf[state][0];
}

stack$target:::pop
/state == 2 ||
state == 0/
{
        state = tf[state][1];
}

stack$target:::push
/state == 2 ||
state == 0/
{
        state = tf[state][2];
}

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

B.5.2 Probe specification script for stack with static probes

```
provider stack {
    probe push(int, unsigned int*);
    probe pop(unsigned int*);
    probe empty(unsigned int);
    probe full(unsigned int*);
};

#pragma D attributes Evolving/Evolving/Common provider stack provider
```

```
#pragma D attributes Private/Private/Common provider stack module
#pragma D attributes Private/Private/Common provider stack function
#pragma D attributes Evolving/Evolving/Common provider stack name
#pragma D attributes Evolving/Evolving/Common provider stack args
```

B.5.3 Generated probe header file for stack with static probes

```
/*
 * Generated by dtrace(1M).
 */

#ifndef _PROBES_H
#define _PROBES_H

#include <unistd.h>

#ifdef __cplusplus
extern "C" {
#endif

#define STACK_STABILITY "___dtrace_stability$stack$v1$5_5_5_1_1_5_1_1_5_5_5_5_5_5_

#define STACK_TYPEDEFS "___dtrace_typedefs$stack$v2"

#if !defined(DTRACE_PROBES_DISABLED) || !DTRACE_PROBES_DISABLED

#define STACK_EMPTY(arg0) \
do { \
        __asm__ volatile(".reference " STACK_TYPEDEFS); \
        __dtrace_probe$stack$empty$v1$756e7369676e656420696e74(arg0); \
        __asm__ volatile(".reference " STACK_STABILITY); \
} while (0)
#define STACK_EMPTY_ENABLED() \
        ({ int _r = __dtrace_isenabled$stack$empty$v1(); \
                __asm__ volatile(""); \
                _r; })
#define STACK_FULL(arg0) \
do { \
        __asm__ volatile(".reference " STACK_TYPEDEFS); \
        __dtrace_probe$stack$full$v1$756e7369676e656420696e74202a(arg0); \
        __asm__ volatile(".reference " STACK_STABILITY); \
} while (0)
#define STACK_FULL_ENABLED() \
        ({ int _r = __dtrace_isenabled$stack$full$v1(); \
                __asm__ volatile(""); \
                _r; })
#define STACK_POP(arg0) \
```

```
do { \
        __asm__ volatile(".reference " STACK_TYPEDEFS); \
        __dtrace_probe$stack$pop$v1$756e7369676e656420696e74202a(arg0); \
        __asm__ volatile(".reference " STACK_STABILITY); \
} while (0)
#define        STACK_POP_ENABLED() \
        ({ int _r = __dtrace_isenabled$stack$pop$v1(); \
                __asm__ volatile(""); \
                _r; })
#define        STACK_PUSH(arg0, arg1) \
do { \
        __asm__ volatile(".reference " STACK_TYPEDEFS); \
        __dtrace_probe$stack$push$v1$696e74$756e7369676e656420696e74202a(arg0, arg1); \
        __asm__ volatile(".reference " STACK_STABILITY); \
} while (0)
#define        STACK_PUSH_ENABLED() \
        ({ int _r = __dtrace_isenabled$stack$push$v1(); \
                __asm__ volatile(""); \
                _r; })


extern void __dtrace_probe$stack$empty$v1$756e7369676e656420696e74(unsigned int);
extern int __dtrace_isenabled$stack$empty$v1(void);
extern void __dtrace_probe$stack$full$v1$756e7369676e656420696e74202a(const unsigned int
extern int __dtrace_isenabled$stack$full$v1(void);
extern void __dtrace_probe$stack$pop$v1$756e7369676e656420696e74202a(const unsigned int *
extern int __dtrace_isenabled$stack$pop$v1(void);
extern void __dtrace_probe$stack$push$v1$696e74$756e7369676e656420696e74202a(int, const u
extern int __dtrace_isenabled$stack$push$v1(void);

#else

#define        STACK_EMPTY(arg0) \
do { \
        } while (0)
#define        STACK_EMPTY_ENABLED() (0)
#define        STACK_FULL(arg0) \
do { \
        } while (0)
#define        STACK_FULL_ENABLED() (0)
#define        STACK_POP(arg0) \
do { \
        } while (0)
#define        STACK_POP_ENABLED() (0)
#define        STACK_PUSH(arg0, arg1) \
do { \
        } while (0)
#define        STACK_PUSH_ENABLED() (0)
```

```
#endif /* !defined(DTRACE_PROBES_DISABLED) || !DTRACE_PROBES_DISABLED */


#ifdef __cplusplus
}
#endif

#endif /* _PROBES_H */
```

Appendix C

Code for the web server/database case study

C.1 The Node.js web server

C.1.1 Version and installation

The Node.js version used in this thesis is `5.5.0` compiled by Homebrew [10] on Mac OS X El Capitan.

C.1.2 Code listing

```
const http = require('http');
const pg = require('pg');

const conString = "postgres://cmr:@localhost/basic";

const client = new pg.Client(conString);

client.connect(function(error){
    if(error){
        console.log("could not connect to postgresql");
    }
});

client.query('SELECT pg_backend_pid() as PID',
        function (error, result){

            if(error){
                console.log(error);
                return;
            }
```

```
            console.log("Node.js PID is " + process.pid);
            console.log("PostgreSQL client PID is " +
                    result.rows[0]["pid"]);
        }
    );

const hostname = '127.0.0.1';
const port = 1337;

http.createServer((req, res) => {

    res.writeHead(200, { 'Content-Type': 'text/plain' });

    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');

}).listen(port, hostname, () => {
    console.log('Server running at ' + hostname + ', port ' + port);
});
```

C.2 The database schema

```
CREATE TABLE entries (
 id     serial primary key,
 entry text NOT NULL,
 timestamp      timestamptz
);
```

C.3 Generated monitor using counters

```
#!/usr/sbin/dtrace -qs

#pragma D option defaultargs

/*
q0 mapped to 1
q1 mapped to 0
*/
```

```
/*
node$target:node::http-server-response/ (nresponses > nqueries)/ mapped to 0
tick-10hz/((nrequests - nresponses) > 100) || ((nrequests - nqueries) > 100)/ mapped to 1
*/

int HAS_VERDICT;
int state;
int tf[2][2];

/* Counter initialization manually added */
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++;
}

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

node$target:node::http-server-response
/ ( (nresponses > nqueries)) && (state == 0)/
{
        trace("REJECTED DUE TO MISMATCH");
        HAS_VERDICT = 1;
        exit(0);
}

tick-10hz
```

```
/ (((nrequests - nresponses) > 100) || ((nrequests - nqueries) > 100)) && (state
{
        trace("REJECTED DUE TO UNRESPONSIVENESS");
        HAS_VERDICT = 1;
        exit(0);
}

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

Appendix D

graphviz2dtrace source code

```
#!/usr/local/bin/python
"""graphviz2dtrace.py

Usage:
    graphviz2dtrace.py FILE
    graphviz2dtrace.py --version
    graphviz2dtrace.py (-m | --mapping) <map> FILE
    graphviz2dtrace.py (-h | --help)

Options:
    -h --help               Show this screen.
    -v --version            Show version.
    -m --mapping            Provide JSON mapping file for atomic variables
"""

from docopt import docopt
from itertools import count
import pygraphviz as pgv
import json
import re
from datetime import datetime
from sys import exit

VERSION = "0.1"

def dtrace_preamble():

    return "#!/usr/sbin/dtrace -qs\n\n#pragma D option defaultargs"

def script_signature():

    signature = []
```

```
    signature.append("This script was automatically generated by " +
                     "graphviz2dtrace version " + VERSION)

    signature.append("on " + str(datetime.utcnow()) +
                     " UTC. To run it, make it executable with")

    signature.append("\nchmod +ux\n\n" +
                     "and then run it with root privileges. Example:")

    signature.append("\nsudo myscript.d -c program-to-be-monitored");

    return "\n/*\n" + "\n".join(signature) + "\n*/\n"

def create_node_ordering(graph, start_node):

    ordering = {}
    ordering[start_node] = 0

    counter = count(start=1)

    for node in graph.nodes():
        if node not in ordering:
            ordering[node] = counter.next()

    return ordering

def unquote(string):
    if string[0] == "\"":
        return string[1:-1]
    return string

def enforce_mapping(graph, mapping):

    if mapping:
        for edge in graph.edges():
            label = unquote(edge.attr['label'])
            start_edge = edge.attr['label'] != u"START"
            if start_edge and (label in mapping):
                    edge.attr['label'] = mapping[label]

    return graph

def load_automata(path):
    graph_file = open(path, 'ro');
    G = pgv.AGraph("".join(graph_file.readlines()))
    graph_file.close()
    return G
```

```
def load_mapping(path):
    mapping_fptr = open(path, 'ro')
    mapping = json.load(mapping_fptr)
    mapping_fptr.close()
    return mapping

def find_start_node(edges):
    for edge in edges:
        if edge[0] == u"start":
            return edge[1]

def remove_question_edges(graph):

    q_edges = filter(lambda edge : edge.attr['label'] == u"?",
                     graph.edges())

    for edge in q_edges:
        assert edge[0] == edge[1], "wildcard transitions between states not supported"
        graph.delete_edge(edge[0], edge[1])

    return graph

def create_probe_ordering(graph):
        probe_ordering = {}
        counter = count()
        for probe in unique_probes(graph):
            probe_ordering[probe] = counter.next()
        return probe_ordering

def unique_probes(graph):
    return list(set(map(lambda edge: edge.attr['label'],
                        graph.edges())))

def control_variables_declaration():
    return "int HAS_VERDICT;\nint state;"

def transition_function_declaration(graph):
    n_nodes = len(graph.nodes())
    distinct_probes = len(unique_probes(graph))
    return "int tf[{0}][{1}];\n".format(n_nodes, distinct_probes)

def state_initialization_statement(probe_ordering):

    # Determine the number of macro variables used, and
    # assign a macro variable for holding an (optional)
    # start state. The assigned macro variable is the
    # next available macro variable.
```

```
    macro_vars = set()

    pattern = re.compile("\$(\d)+")

    probes_w_macro_vars = filter(lambda p: pattern.search(p) != None,
                                 probe_ordering)

    macro_vars.update(map(lambda probe: pattern.search(probe).group(),
                              probes_w_macro_vars))

    sorted_macro_vars = sorted(list(macro_vars))
    idealized_macro_vars = map(lambda n: "$"+str(n),
                              range(1, len(macro_vars)+1))

    no_gaps = sorted_macro_vars == idealized_macro_vars

    assert no_gaps, "Gaps between macro variables not permitted"

    variable = "$" + str(len(macro_vars)+1)

    return "\tstate = (" + variable + " ? " +  variable + ": 0);"

def begin_block(graph, node_ordering, probe_ordering):

    statements = []
    statements.append("dtrace:::BEGIN\n{")

    tfi = transition_function_init(graph, node_ordering,
                                                probe_ordering)
    statements.append(tfi)

    statements.append("\tHAS_VERDICT = 0;");
    statements.append(state_initialization_statement(probe_ordering))
    statements.append("}\n")

    return "\n".join(statements)

def transition_function_init(graph, node_ordering, probe_ordering):

    def encode_entry(state, symbol, resulting_state):
        components = []
        components.append("\ttf[")
        components.append(str(state))
        components.append("][")
        components.append(str(symbol))
        components.append("] = ")
        components.append(str(resulting_state))
        components.append(";")
```

```
        return "".join(components)

    statements = []

    probes = set(map(lambda edge: edge.attr['label'], graph.edges()))

    for node in graph.nodes():

        # For every outbound edge from the node, insert the destination
        # state in the transition table

        for edge in graph.out_edges([node]):
            probe = edge.attr['label']
            statements.append(encode_entry(node_ordering[node],
                                           probe_ordering[probe],
                                           node_ordering[edge[1]]))

        # Since the transition function needs to be complete, we
        # determine which probes were used as labels in the outbound
        # edges and which where not. For those probes which had no
        # corresponding outgoing edge from this node, we encode in the
        # transition table that the state should remain in the same
        # state.

        covered_probes = set(map(lambda edge: edge.attr['label'],
                                 graph.out_edges([node])))

        uncovered_probes = probes.difference(covered_probes)

        for probe in uncovered_probes:
            statements.append(encode_entry(node_ordering[node],
                                           probe_ordering[probe],
                                           node_ordering[node]))

    return "\n".join(sorted(statements))

def plain_probe(probe):
    if "/" in probe:
        return probe.split("/")[0]
    return probe

def create_predicate(ordered_nodes, probe):


    core = " ||\n".join(map(lambda node: "state == " + str(node),
                            ordered_nodes))

    #Check if probe already contains a predicate
```

```
    if "/" in probe:
        try:
            custom_predicate = probe.split("/")[1]
            return "/ (" + custom_predicate + ") && (" + core + ")/"

        except IndexError:
            print "Malformed probe " + probe + ". Exiting..."
            exit(1)

    return "/" + core + "/"

def blocks_from_map(probe_map, node_ordering, accepting):
    blocks = []

    for probe in probe_map:
        nodes = map(lambda node: node_ordering[node], probe_map[probe])

        predicate = create_predicate(nodes, probe)
        verdict = "\"ACCEPTED\"" if accepting else "\"REJECTED\""

        blocks.append(plain_probe(probe) + "\n" + predicate
                      + "\n{\n"
                      + "\ttrace(" + verdict + ");\n"
                      + "\tHAS_VERDICT = 1;\n"
                      + "\texit(0);\n"
                      + "}\n")
    return blocks

def get_probe_map(graph, states):

    probe_map = {}

    for state in states:
        for edge in graph.in_edges([state]):
            probe = edge.attr['label']
            if probe not in probe_map:
                probe_map[probe] = []

            probe_map[probe].append(edge[0])

    return probe_map

def accepting_blocks(graph, node_ordering):

    greens = filter(lambda n: n.attr['fillcolor'] == u"green", graph.nodes())

    probe_map = get_probe_map(graph, greens)
```

```
    return blocks_from_map(probe_map, node_ordering, accepting=True)

def rejecting_blocks(graph, node_ordering):

    reds = filter(lambda node: node.attr['fillcolor'] == u"red", graph.nodes())
    rejecting_states = set(reds)

    probe_map = get_probe_map(graph, rejecting_states)
    return blocks_from_map(probe_map, node_ordering, accepting=False)

def neutral_blocks(graph, node_ordering, probe_ordering):

    neutral_edges = filter(lambda e: e[1].attr['fillcolor'] not in [u"green", u"red"],
                           graph.edges())

    probe_map = {}

    for edge in neutral_edges:
        probe = edge.attr['label']

        if probe not in probe_map:
            probe_map[probe] = []

        probe_map[probe].append(edge[0])

    blocks = []

    for probe in probe_map:

        ordered_nodes = map(lambda n: node_ordering[n], probe_map[probe])
        predicate  = create_predicate(ordered_nodes, probe)

        blocks.append(plain_probe(probe) + "\n" + predicate
                      + "\n{\n"
                      + "\tstate = tf[state][" +
                      str(probe_ordering[probe]) + "];\n"
                      + "}\n")

    return blocks

def inconclusive_block():

    return "dtrace:::END\n/ !HAS_VERDICT /\n" + "{\n\ttrace(\"INCONCLUSIVE\");\n}\n"

def comment_ordering(ordering):
    statements = []

    statements.append("/*")
```

```
    for key in sorted(ordering.keys()):
        statements.append(" ".join([key, "mapped to", str(ordering[key])]))

    statements.append("*/")

    return "\n".join(statements) + "\n"

def create_dscript(automata, mapping):

    G = enforce_mapping(automata, mapping)

    #The node we actually start in, not the artificial start node
    start_node = find_start_node(G.edges())

    #Having used the artificial start node to find the real start node,
    #we remove the artificial startnode.
    G.remove_node(u'start')

    G = remove_question_edges(G)

    # We want to use a two-dimensional static array of integers
    # to encode the transition function, therefore we need to
    # map each node to a number and every unique dtrace probe
    # to a number.

    # We map the start_node to 0 as this makes initializing the
    # state variable trivial, and the others to consecutive numbers
    # until all nodes are mapped to an integer.
    node_ordering = create_node_ordering(G, start_node)

    # Similarly, we map every probe to an integer.
    probe_ordering = create_probe_ordering(G)

    statements = []

    statements.append(dtrace_preamble())
    statements.append(script_signature())
    statements.append(comment_ordering(node_ordering))
    statements.append(comment_ordering(probe_ordering))

    statements.append(control_variables_declaration())
    statements.append(transition_function_declaration(G))
    statements.append(begin_block(G, node_ordering, probe_ordering))

    for block in accepting_blocks(G, node_ordering):
        statements.append(block)
```

```
        for block in rejecting_blocks(G, node_ordering):
            statements.append(block)

        for block in neutral_blocks(G, node_ordering, probe_ordering):
            statements.append(block)

        statements.append(inconclusive_block())

        return "\n".join(statements)

if __name__ == '__main__':

    arguments = docopt(__doc__, version=VERSION)

    automata = load_automata(arguments['FILE'])

    mapping = None

    if arguments['--mapping']:
        mapping = load_mapping(arguments['<map>'])

    print create_dscript(automata, mapping)
```

Bibliography

[1] Hamid Aalav et al. *Specification Patterns*. http://patterns.projects.cis.ksu.edu/. Accessed: 2015-08-13.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. 1st ed. Cambridge, Massachusetts: The MIT Press, 2008.

[3] Andreas Bauer, Martin Leucker, and Christian Schallhart. "FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science: 26th International Conference, Kolkata, India, December 13-15, 2006. Proceedings." In: ed. by S. Arun-Kumar and Naveen Garg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. Chap. Monitoring of Real-Time Properties, pp. 260–272.

[4] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL." In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (Sept. 2011), 14:1–14:64.

[5] Arif Bilgin et al. *Graphviz - Graph Visualization Software*. http://graphviz.org.

[6] J Richard Büchi. "On a decision method in restricted second order arithmetic." In: *Proc. Internat. Congr. Logic, Method. and Philos. Sci.* 1960, pp. 1–12.

[7] Bryan Cantrill. "Hidden in Plain Sight." In: *Queue* 4.1 (Feb. 2006), pp. 26–36.

[8] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. "Dynamic Instrumentation of Production Systems." In: USENIX Annual Technical Conference, General Track. 2004.

[9] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. Cambridge, Mass: MIT press, 1999.

[10] Homebrew contributors. *brew – The missing package manager for OS X*. http://brew.sh/.

[11] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. Cambridge, Massachusetts: The MIT Press, 2009.

[12] Oracle Corporation. *DTrace Guide for Oracle Solaris 11*. Oracle Corporation. 2012.

[13] Node.js Foundation. *About Node.js*. https://nodejs.org/en/about/.

[14] Node.js Foundation. *Node.js*. https://nodejs.org/en/.

[15] The Apache Software Foundation. *ab - Apache HTTP server benchmarking tool*. https://httpd.apache.org/docs/2.4/programs/ab.html.

[16] Brendan Gregg. *DTrace pid Provider Overhead*. http://dtrace.org/blogs/brendan/2011/02/18/dtrace-pid-provider-overhead/. 2011.

[17] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[18] The PostgreSQL Global Development Group. *PostgreSQL*. http://www.postgresql.org/.

[19] The PostgreSQL Global Development Group. *PostgreSQL Documentation: Dynamic Tracing*. http://www.postgresql.org/docs/current/static/dynamic-trace.html.

[20] Aric Hagberg, Dan Schult, and Manos Reneiris. *PyGraphviz*. https://pygraphviz.github.io/. 2004–2016.

[21] Vladimir Keleshev. *docopt–Pythonic command line arguments parser that will make you smile*. https://github.com/docopt/docopt.

[22] Orna Kupferman and Moshe Y Vardi. "Model checking of safety properties." In: *Formal Methods in System Design* 19.3 (2001), pp. 291–314.

[23] Martin Leucker. "Teaching Runtime Verification." English. In: *Runtime Verification*. Ed. by Sarfraz Khurshid and Koushik Sen. Vol. 7186. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 34–48.

[24] Martin Leucker and Christian Schallhart. "A brief account of runtime verification." In: *The Journal of Logic and Algebraic Programming* 78.5 (2009). The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07), pp. 293–303.

[25] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. New York: Springer-Verlag, 1992.

[26] Mozilla Developer Network. *JavaScript - Concurrency model and Event Loop*. https://developer.mozilla.org/en/docs/Web/JavaScript/EventLoop.

[27] Amir Pnueli. "The temporal logic of programs." In: *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE. 1977, pp. 46–57.

[28] Amir Pnueli and Aleksandr Zaks. "PSL model checking and run-time verification via testers." In: *FM 2006: Formal Methods*. Springer, 2006, pp. 573–586.

[29] Torben Scheffel and Malte Schmitz et al. *LamaConv—Logics and Automata Converter Library*. http://www.isp.uni-luebeck.de/lamaconv.

[30] Michael Sipser. *Introduction to the Theory of Computation*. 3rd ed. Cengage Learning International Offices, 2013.

[31] Martin Steffen and Volker Stolz. “Lecture notes on Specification and Verification of Parallel Systems: Linear Temporal Logic.” https://www.uio.no/studier/emner/matnat/ifi/INF5140/v15/slides/4-hlandtl.pdf. 2015.

[32] Volker Stolz and Klaus Indermark. “Temporal assertions for sequential and concurrent programs.” Prüfungsjahr: 2006. - Publikationsjahr: 2007; Aachen, Techn. Hochsch., Diss., 2006. PhD thesis. Aachen, 2006, II, 133 S. : graph. Darst.

[33] Moshe Y Vardi and Pierre Wolper. “An automata-theoretic approach to automatic program verification.” In: *Proceedings of the First Symposium on Logic in Computer Science*. IEEE Computer Society. 1986, pp. 322–331.

[34] Pierre Wolper. “Constructing Automata from Temporal Logic Formulas: A Tutorial.” English. In: *Lectures on Formal Methods and Performance Analysis*. Ed. by Ed Brinksma, Holger Hermanns, and Joost-Pieter Katoen. Vol. 2090. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 261–277.