Using DSLs for Software Testing  
*Mark Micalef and Christian Colombo*

Event-Based Characterisation of Temporal Properties over System States  
*Christian Colombo, Gordon J Pace and Justine Scicluna*

A Domain Specific Property Language For Fraud Detection To Support Agile Specification Development  
*Aaron Calafato, Christian Colombo and Gordon J Pace*

Remote and scalable interactive high-fidelity graphics using asynchronous computation  
*Keith Bugeja*

Tipping the Scales... Functionally!  
*Gordon J. Pace and Kevin Vella*

Using dynamic binary analysis for tracking pointer data  
*John Galea and Mark Vella*

Cloud and mobile security assurance: a memory forensics approach  
*Mark Vella*

Improving the Gherkin Specification Language using Aspect-Oriented Techniques  
*John Aquilina Alamango, Christian Colombo and Mark Micalef*

A DSL for Business Intelligence Monitoring  
*Christian Colombo, Jean Paul Grech and Gordon Pace*
Towards achieving efficient Runtime Enforcement for Component Based Systems
Ian Cassar and Adrian Frncalanza

Mobile Erlang Computations to Enhance Performance, Resource Usage and Reliability
Adrian Frncalanza and Tyron Zerafa

Enabling Usage of Cloud-based Applications when Offline
Joshua Ellul

Extracting Monitors from JUnit Tests
Christian Colombo, Jonathan Micallef and Mark Micallef

Investigating different Instrumentation Techniques in the context of ESB Runtime Verification
Gabriel Dimech, Christian Colombo and Adrian Frncalanza

Challenges faced when Forcing Malware Execution down Hidden Paths
James Gatt, Mark Vella and Mark Micallef

Improving Android Security through Real-time Policy Enforcement
Luke Chircop, Christian Colombo and Gordon J. Pace
Using DSLs for Software Testing

Mark Micallef\textsuperscript{1} and Christian Colombo\textsuperscript{2}

University of Malta

In her widely cited paper about the future of software testing, Bertolino \cite{1} claims that domain specific languages (DSLs) have emerged as an efficient solution towards allowing experts within a domain to express specifications in that domain. She goes on the claim that success of domain-specific approaches should be built upon and extended to the testing stage of software engineering. An intuitive place to start would be to explore DSLs in the context of software testing such that languages constructed by domain experts can be leveraged to specify not only requirements but also test cases which validate those requirements. In this talk, we present and discuss the outcomes of three exploratory case studies which we carried out in order to investigate the utility of DSLs as applied to specifying tests in different domains, with each case study focusing on a particular aspect/characteristic of this application of DSLs. The three case studies are as follows:

**Android Application Testing** - In this case study, the focus was on investigating the possibility of designing a language which was able to express tests over the domain of applications developed for the Android platform. The main characteristic of such applications is that they exist in a domain, which is sufficiently different from other domains (e.g. desktop applications) to merit a domain-specific approach yet whose concepts are well-defined in official documentation \cite{2} and understood by technical and non-technical stakeholders alike. In the study we developed a DSL which merged concepts from the Android platform with concepts from the domain of software testing thus allowing stakeholders to specify tests. Furthermore, we developed a prototype which implemented a subset of the features of the language as a proof-of-concept that the approach is feasible.

**Graphical Games Testing** - Whilst existing approaches to test automation sufficiently cater for the testing of “traditional” software systems which incorporate a standard set of user interface components, the same cannot be said when it comes to graphical games. Such games do not provide standard user interfaces and there is rarely any documentation which explicitly defines the domain. Hence in this case study we investigated the challenges involved in designing a DSL that expresses tests over a loosely-defined domain as well as the technical challenges involved in the execution of tests over graphical games. This work lead to the design of an extensible DSL for specifying tests over the domain of graphical games and the implementation of a prototype which executed tests against two popular games in the market.

**Technology-Agnostic Test Automation for B2B Websites** - Readers who purchase items online from a variety of online stores are likely familiar with notions such as a product search, a shopping cart, checkout process, and so on. These notions are basic building blocks which are found in most B2B websites. Yet when it comes to specifying and implementing automated tests in these domains, companies are forced to
start from scratch and build custom automation frameworks for their websites. In this case study we explored the idea of designing a DSL which is generic enough to express tests over B2B websites. We then used a classifier-based technique and prototype which demonstrates that it is possible for one to specify and automatically execute tests in the DSL without needing to be intimately familiar with the technical details of the website. This approach differs substantially from the current state of the art in which test engineers need to use application-specific hooks in order to implement automated tests. We believe this approach is likely to (1) encourage more focus on what the application should be doing rather than how it is doing it and (2) reduce the phenomenon of brittle tests whereby automated tests break as a system evolves, even if evolution is of a cosmetic nature.

Future work in this area will proceed down two tracks. Firstly, at least in the short term, we would like to maintain an exploratory mentality in which we carry out case studies across a number of domains in order to make observations and identify interesting research areas. Secondly, we would like to identify promising prototypes which arise from this work and continue to invest time in making them more feature-complete with a view of being able to apply them to real-life case studies with our industry collaborators.

References

2. Google. Android user interface overview.
Event-Based Characterisation of Temporal Properties over System States

Christian Colombo\textsuperscript{1}, Gordon J. Pace\textsuperscript{1}, and Justine Scicluna\textsuperscript{1}

University of Malta

The design of runtime verification \cite{3} (or monitoring) systems presents a myriad of options — how to instrument properties, in which logic to specify properties, what algorithms to use to implement the property checking, etc. One crucial issue is what elements of the system one is interested in observing, and what points-of-interest one must capture to be able to perform this monitoring. Many runtime verification tools base their properties on the control flow of the system (e.g. \cite{2}); method calls, object creation, exception raising, etc. Especially in the domain of distributed systems, one also finds communication-centric runtime monitoring, in which one focuses on the communication taking place between nodes (e.g. see \cite{1}). Finally, a minority of tools take a data-centric approach, in which one can write properties about the values stored in the system state. The choice of approach has a major influence on how monitoring code can be instrumented in the system. Typically, control-centric approaches use aspect-oriented programming (or similar) technologies to insert additional code identifying the events of interest in the system. On the other hand, to monitor communication in a distributed, message-passing system, one may create communication proxies (actual or local virtual ones) which capture and analyse the messages, i.e. the temporal points-of-interest in such a system. In a data-centric approach, one typically captures points of discontinuity in the values of variables — when they are assigned a value — to be able to capture properties which talk about how the values of the system state changes over time\textsuperscript{1}.

In this brief presentation, we will present ongoing experiments on adding data-oriented monitoring to the runtime verification tool LARVA \cite{2}, and in particular looking at temporal values which change continuously over time, such as the average of the value of a variable over time.

Real-Time System Behaviour. The behaviour of a system can be characterised by the values stored in variables, possibly changing over time. Let $\mathcal{I}$ be all possible interpretations of variables $\text{Var}$ ranging over values $\text{Val}$ for a single instant of time ($\mathcal{I} = \text{Var} \rightarrow \text{Val}$). The temporal behaviour of such a system is characterised by a total function $\mathcal{I}_t \in \mathbb{T} \rightarrow \mathcal{I}$ — where $\mathbb{T}$ is the time-domain, typically $\mathbb{R}^+$.

Practically Monitorable Points. In practice, monitors cannot keep track of the variables at each point on the real-time line, but typically, one would be able to monitor (effectively) points in time when a variable has been assigned to a new value.

\textsuperscript{1} In the literature, one also finds work on the monitoring of analogue systems e.g. \cite{4}, in which variables may change values in a continuous way over time. However, in this presentation, we will focus on digital system which exhibit memory values which discretely change (in a discontinuous manner) over time.
This provides a discrete way of characterising the value of a variable over time in a set of sequential time-stamped values identifying (all) points in time when the variable changed its value. Thus, for example, two variables $x$ and $y$ both with initial value 12 and whose value increasing by 7 and 9 (respectively) every 1.2 and 1.8 seconds (respectively) would be represented as a prefix of the infinite time-stamped trace:
\[
\langle (0, \{ x \mapsto 12, y \mapsto 12 \}), (1.2, \{ x \mapsto 19, y \mapsto 12 \}), (1.8, \{ x \mapsto 19, y \mapsto 21 \}), \ldots \rangle.
\]
The set of finite timed traces $Tr$ is thus equivalent to $(T \times I)^*$. We will assume that our variables do not exhibit Zeno-like behaviour (there can only be a finite number of value changes over any finite time interval), which makes the trace models equally expressive as the modelling of variables as a function over time.

**Data-Oriented Monitoring.** The time-stamped trace identifies the points in time which one can easily capture at runtime. To identify which of these are of interest to our properties, we add two operators $\overline{e}$ and $\overline{e}$, which respectively give the value of expression $e$ just before and just after the moment of evaluation. For example, $\overline{x} \neq \overline{x}$ identifies the points in time when the value of $x$ has changed. We can then design monitors based on these chosen events using any temporal logic to describe prohibited system traces. For instance, one may use LTL to write the property that the number of downloads may only increase until it is reset\(^2\): $\Box (\text{downloads} \leq \text{downloads}) U \text{reset}$.

We have explored this approach of data-oriented point-of-interest identification into the runtime verification tool LARVA [2], and applied it to a number of case studies, including data modification intensive red-black tree implementation and an online shopping system SoftSlate\(^3\).

**New Points-of-Interest.** Sometimes, we need to consider properties over values which change in a continuous manner along time. A typical example of this is the average value of a variable over time e.g. the event characterised when the average value of variable $x$ exceeds 20. Now, if $x$ has held value 10 from time $t = 0s$ till $t = 100s$ when it is assigned to 30, and if no other changes on $x$ occurs, the average will exceed 20 at time $t = 200s$, despite the fact that the system would not have otherwise been interrupted at that point in time.

To enable identification of such temporal points of interest, we characterise them using an arithmetic expression over the integral of system variables (written $\int v$), going above an upper limit (written $@_u(e)$) or below a lower limit (written $@_l(e)$).

For instance, the event which triggers when the average of $x$ exceeds 20 would be written as $@_u^{20} (\int x / \int 1)$. Since these integrals increase linearly over time, we can calculate the implicit events when to interrupt the system with every system event. For instance, in the average example, after the system has performed $\langle (0s, \{ x \mapsto 10 \}), (100s, \{ x \mapsto 30 \}) \rangle$, we can add a timer to cause an event at time 200s, which would be cancelled (or rescheduled) if $x$ changes its value again in the meantime.

Formally, we define a function $earliest$, which given a property (using the integral over time of variables) and a prefix behaviour, will return the earliest time (if any)

\(^2\) Although different approaches are possible, we assume that the inequality is checked upon any system event, including an assignment to any variable.

when the property would be violated if no other events happen from the system side. It turns out, that for linear and quadratic uses of the integral operator, this can be readily computed.

Discussion and Future Work. We are currently finalising the formalisation of these notions, and looking into extending the expressivity of the integral operator, by adding two special cases: (i) \([\int x]_n\) which takes the integral of \(x\) over the past \(n\) time units; and (ii) \([\int x]_e\) takes the integral of \(x\) since the last occurrence of event \(e\). It would be interesting to investigate how these added operators abstract away complex calculations and timer handling in our specification language.

References

A Domain Specific Property Language For Fraud Detection To Support Agile Specification Development

Aaron Calafato, Christian Colombo, and Gordon J. Pace
Dept. of Computer Science University of Malta

Fraud detection is vital in any financial transaction system, including the collection of tax. The identification of fraud cases was traditionally carried out manually, having fraud experts going through their records and intuitively selecting the ones to be audited — a lengthy and unstructured process. Although work has been done with regards to the use of artificial technology for fraud pattern discovery, the results are not encouraging without major intervention by fraud experts [4].

Nowadays, in practice, patterns identified by fraud experts are coded by the software developers who select fraud cases from a database. The resulting application is verified by the fraud expert, who may feel the need to refine the rules in multiple iterations. However, this process is prone to human-induced bugs due to the continuous manual work. A better approach would include the description of rules through the use of a structured grammar, understandable by a computer system. With a compilable set of descriptions, the rules may be automatically processed against historical data — limiting the dependency on a software developer solely to the process of setting up the system.

1 Proposed Solution

In order to enable an automated system, the fraud patterns need to be translatable to a more computerised language. Fully natural languages tend to lead towards ambiguous descriptions [1]. Domain Specific Languages (DSL) [2] are languages whose expressivity is focused on the domain in question. The control of expressiveness is derived from the implementation of core concepts with further functionalities built on top of these [3]. For instance, in a tax-related fraud detection system, a rule: “Any company declaring less than 2000 Euro in profits is fraudulent”, may need to be described. With a DSL, the core definitions (i) company, (ii) year, and (iii) profits would be defined first. These would subsequently be used to build rules, similar to Figure 1c. Being built on a grammar (with defined functions), a well-designed DSL reduces the ambiguity of the rules defined. For instance, the natural example does not specify the year in question, whilst in a DSL, the year may be compulsory for the definition.

DSLs like [3] however expose a syntax that is unnatural for fraud experts, since it is built on top of a programming language. Controlled Natural Languages (CNL)[5] are a type of DSL legible to human beings whilst still being easily processed by a computerised system. One of the major tools in this area is Grammatical Framework (GF) [1], which enables the definition of multiple grammars, linked through a common structure. Defining the grammars from scratch allows us to control the expressivity in the fraud detection language. We propose to translate an English-like sentence (Figure
1a) to a programming-like representation (Figure 1c), the former targeting the fraud detection patterns, whilst the latter would be used for the detection of cases. Figure 1b represents the intermediate representation between the translation. An English-like language is more natural for a fraud expert, due to the morphological inflections done with GF. For instance, “Any company” in Figure 1a would imply a singular noun, which is subsequently reflected in the “is fraudulent” part. Once the programming code is created, an automated system can select the matching cases from a historical data and return them to the fraud expert, as shown in Figure 1d. This feedback mechanism allows for rules to be defined in an iterative style, such as refining the “2000 Euro” to another value to view the difference in the selected cases.

(a) Any company declaring less than 2000 Euro in profits for the current year is fraudulent
(b) Declare(Company, Profit, less, 2000, CurrentYear)
(c) Company.Year(Current).Profit < 2000
(d) Company | Profits
--- | ---
Albatross Ltd | 1000
Ali Baba & Co. | 1500
...
...

Fig. 1. Automated process: (a) definitions with a CNL, (b) interpreted to an abstract language, (c) translated to Java-like syntax, and (d) feedback returned to the fraud expert.

2 Conclusions

In this work we have proposed a way to allow fraud experts to define fraud patterns in a natural way while reducing possible ambiguities. This work will involve the creation of grammars, and the resulting language will be evaluated by an auditor from the Inland Revenue Department. The translated rules will then be processed, and with certain pre-processing, matched cases will be returned to the expert in a timely manner.

References

Remote and scalable interactive high-fidelity graphics using asynchronous computation

Keith Bugeja\textsuperscript{1,2}

\textsuperscript{1} University of Malta
\textsuperscript{2} University of Warwick

Abstract. Current computing devices span a large and varied range of computational power. Interactive high-fidelity graphics is still unachievable on many of the devices widely available to the public, such as desktops and laptops without high-end dedicated graphics cards, tablets and mobile phones. In this paper we present a scalable solution for interactive high-fidelity graphics with global illumination in the cloud. Specifically, we introduce a novel method for the asynchronous remote computation of indirect lighting that is both scalable and efficient. A lightweight client implementation merges the remotely computed indirect contribution with locally computed direct lighting for a full global illumination solution. The approach proposed in this paper applies instant radiosity methods to a precomputed point cloud representation of the scene; an equivalent structure on the client side is updated on demand, and used to reconstruct the indirect contribution. This method can be deployed on platforms of varying computational power, from tablets to high-end desktops and video game consoles. Furthermore, the same dynamic GI solution computed on the cloud can be used concurrently with multiple clients sharing a virtual environment with minimal overheads.

1 Introduction

Modelling of global illumination increases the level of realism and immersion in virtual environments [8]. While a large number of methods for computing graphics of higher fidelity have been developed, they typically trade off quality for performance and are incapable of running on all but machines with the highest specifications. Cloud computing has enabled the use of low-performance devices for tasks beyond their computational capabilities. Complex tasks are assimilated into cloud services, allowing applications running on these devices to request and receive the results in a fraction of the time it would take the local device to compute. In terms of visualisation and rendering, this model has recently been exploited by providers such as OnLive to provide interactive streaming services for games [5]. A thin client connects to a data-centre in the cloud, where the service provider hosts and runs the actual game, and receives its audiovisual output stream. User input, such as directional controls and button presses, are transmitted by the client to the server, fed to the game and in response, the game output is sent back to the client in the form of a compressed video stream. The bulk of the computation is carried out at the provider’s data-centre, allowing a wide range of devices to consume the service, making the computational capacity of the client device largely
irrelevant. Although effective in providing the same experience to a plethora of devices with varying capabilities, this paradigm is highly susceptible to network latency and bandwidth constraints. High definition and ultra high definition (UHD) streams, especially at higher frame rates, transfer significant amounts of data (see Table 1), and may exclude some network configurations due to bandwidth limitations or the introduction of undesired lag in programs that require low response times. In these settings, each client connects to an application that performs the rendering in isolation. This one-to-one approach precludes the possibility of rendering algorithms that amortise computation complexity over a number of concurrent clients, such as is the potential with multi-user environments. As opposed to rendering entirely in the cloud, Crassin et al. use an approach similar to what we propose, where the rendering pipeline is only partially offloaded from the client [2]. In particular, they introduce a distributed rendering pipeline which computes the indirect lighting contribution in the cloud, amortising the computations across multiple clients in a multi-user environment. Three lighting algorithms were proposed, each with different bandwidth and reconstruction costs [3][7][4][6]. Two of these algorithms, the path-traced irradiance maps and real-time photon mapping, are asynchronous in nature, decoupling client updates from the cloud computation and the network performance. Irradiance maps yield low bandwidth requirements, and reconstruction costs are also cheap, but the difficulty in acquiring UV-parameterisation for moderately complex scenes doesn’t always make them a viable option due to the laborious nature of the parameterisation [2]. Photon tracing doesn’t require any parameterisations but has substantially larger bandwidth requirements, close to an order of magnitude more than the requirements of streaming cloud gaming platforms. Moreover, the indirect lighting reconstruction at the client poses prohibitive computational costs for some low to mid-range devices. The third algorithm, which adopts a synchronous approach, uses cone-traced sparse voxel global illumination, and although client updates at 30 Hz can be sustained for 5 clients, this soon drops to 12 Hz as soon as the number of clients is increased to 24. The system, CloudLight, supports vertical scaling by the addition of more GPUs to a server node, but it is unclear as to how the system scales horizontally.

<table>
<thead>
<tr>
<th>Service</th>
<th>Resolution</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netflix</td>
<td>720p</td>
<td>4.0 Mbps</td>
</tr>
<tr>
<td>Netflix</td>
<td>1080p</td>
<td>5.0 Mbps</td>
</tr>
<tr>
<td>Hulu Plus</td>
<td>720p</td>
<td>2.0 Mbps</td>
</tr>
<tr>
<td>Hulu Plus</td>
<td>1080p</td>
<td>3.2 Mbps</td>
</tr>
<tr>
<td>OnLive</td>
<td>720p</td>
<td>5.0 Mbps</td>
</tr>
<tr>
<td>Playstation Now!</td>
<td>720p</td>
<td>5.0 Mbps</td>
</tr>
</tbody>
</table>

Table 1. Bandwidth requirements for various video-on-demand services.

This work [1] proposes an asynchronous remote computation technique that provides an efficient method for computing indirect lighting. The proposed method has minimal bandwidth and client-side computation requirements, and can achieve client-
side updates at 60 Hz or more at HD and UHD resolutions. The global illumination solution is split into two components, direct and indirect lighting. The indirect lighting, which is the most computationally expensive, is decoupled from the rest of the rendering and carried out remotely. This service is provided to clients of multi-user environments, thus amortising the cost of computation over all connected consumers of the service. Computation results are stored in an efficient object space representation that does not require the large bandwidths of the solutions above. Furthermore, transfers being in object space are resolution agnostic; increasing client resolution does not increase bandwidth requirements, as is the case with streaming solutions. The lightweight client reconstruction makes this method suitable for any device that supports basic rendering functionality, while also scaling well to many clients connected to the same service, achieving a significant overall boost in performance.

Fig. 1. Test scenes running at over 60 Hz at full HD resolution (1920 × 1080) using our method for the asynchronous computation of indirect illumination in the cloud.

References

Tipping the Scales . . . Functionally!

Gordon J. Pace and Kevin Vella
Department of Computer Science, University of Malta

Although one might be tempted to attribute the choice of notes in musical scales and chords to purely aesthetic considerations, a growing body of work seeks to establish correspondences with mathematical notions. Intriguingly, the most important musical scales turn out to be solutions to optimisation problems.

In this abstract, we review mathematical characterisations which yield some of the most important musical scales. We also describe an implementation of these notions using the functional programming language Haskell, which has allowed us to conduct interactive experiments and to visualise various approaches from the literature.

1 Pitches, Pitch Classes and Scales
The pitch of a musical note is quantified by its frequency. Pitches are partitioned into pitch classes with frequencies $p \times 2^n, -\infty < n < \infty$ (doubling frequencies) under the octave equivalence relation. Pitches within the same pitch class, although having different frequencies, are perceived as being of the same quality [6].

The continuum of pitch classes can be discretised by choosing a finite subset of pitch classes as a palette. This is the chromatic pitch-class set, the raw material at a composer’s disposition. Frequencies are typically chosen at (or near) equidistant points on a logarithmic scale, so that the brain perceives the distance between successive pairs of pitches as being the same. Though by no means universal, the use of twelve pitch classes is predominant in western music [3] and is widely represented in other cultures. Western music also exhibits a preference for scales containing seven note pitch-class sets chosen from the twelve-note chromatic set. The scales are characterised not by the presence of specific pitch classes but by the distances between them.

The clock diagram [2] places chromatic pitch classes at equidistant points on the circumference of a circle. This representation is useful because it eliminates bias towards particular pitch classes or distance patterns (unlike staff notation or a piano keyboard), while embodying the cyclic nature of pitches with respect to octave equivalence.

2 Formalising Musical Scales using Euclidean Geometry
The chromatic pitch classes are represented as $c$ equidistant points on the circumference of a circle. We write $\text{pos}_c(i)$ (where $0 \leq i < c$) to represent the euclidean coordinates of the $i$th point (of $c$ equidistantly distributed points) on the circumference of a circle centred at the origin and with unit radius such that the 0th position lies at $(0, 1)$. A pitch-class set $\psi$ is a sequence of $d$ (with $d \leq c$) distinct pitch classes $\psi = \langle p_0, p_1 \ldots p_{d-1} \rangle$ from a chromatic set of $c$ (i.e. $0 \leq p_i < c$). We write $\Psi_{d/c}$ to denote the set of all possible pitch-class sets of size $d$ from $c$ chromatic pitch classes, and $\Psi_c$ to denote all possible pitch-class sets of any size from $c$ chromatic pitch classes. We write $\text{poly}(\psi)$ to denote the $d$-sided polygon which is formed by joining nearest pairs of chosen points from pitch-class set $\psi$ with straight lines. Given two distinct pitch classes $p$ and $p'$ in a pitch-class set $\psi \in \Psi_c$, we denote the euclidean distance between the points as $\delta_e^{\text{eucl}}(p, p') = |\text{pos}_c(p) - \text{pos}_c(p')|$ and the diatonic distance for a pitch-class set $\psi$ between the points as the number of lines between them: $\delta_d^{\text{diat}}_{\psi}(p, p') = \#(\{p, p'\} \cap \psi)$.

1Contrary to the mathematical definition, in musical set theory the pitch classes in a pitch-class set are ordered.
3 Scales as Solutions to Optimisation Problems

Many important scales can be identified as optimal solutions to geometric measures [1, 5, 4]. Given a metric \( f \in \Psi_c \to \mathbb{R} \), we identify the set of its maximal (or minimal) musical scales of length \( d \) from \( c \) chromatic pitch classes as: 

\[
\text{maximise}_{d/c}(f) = \{ \psi \in \Psi_d/c \mid f(\psi) = M \}
\]

where \( M = \max\{f(\psi) \mid \psi \in \Psi_d/c\} \). In the Haskell implementation, we use higher-order functions to choose the maximal and minimal solutions of a given metric. Maximisation is defined as:

\[
\text{maximise } f (c, d) = \text{maximumBy}
\]

\[
(x \ y \to \text{compare } (f (c, d) x) (f (c, d) y))
\]

\[
(scalesOfSize (c, d))
\]

Achieving maximal evenness [1] is analogous to seating a group of, for instance, seven people at a round table with twelve chairs in such a way that they are spread as evenly as possible. A set of \( d \) pitch classes chosen from \( c \) chromatic pitch classes is maximally even if the pitch classes are arranged as evenly as possible on the circle when their positions are restricted to the positions of the \( c \) chromatic `slots`. Going further, we may optimise metrics over a scale for pitch classes with a particular diatonic distance between them, rather than limit ourselves to ones which are a diatonic distance of one apart [5]. We can, for instance, implement the maximal evenness property for a generalised distance as:

\[
\text{maximalEvennessNApart } n = \text{minimise } (\text{unevennessNApart } n)
\]

\[
\text{unevennessNApart } n (c, d) s = \text{maximum } \text{distances} - \text{minimum } \text{distances}
\]

\[
\text{distances} = \text{map } (\text{diatonicDistance } (c, d)) (\text{pairsNApart } n s)
\]

With \( \text{maximalEvennessNApart } 2 \) \((12,7)\) we identify the seven-note scales which maximise evenness between alternate notes in the scale (also known as diatonic thirds), and our JavaScript visualiser illustrates the result using clock diagrams. The harmonic minor, the melodic minor, the harmonic major and the major scale appear on the screen:

4 Future Work

We observed that the abstraction and extensibility provided by Haskell lends itself to free-wheeling exploration. As the framework is augmented to handle higher degrees of freedom such as harmony and time we expect it to unlock substantial exploratory and experimental research potential.

References

Using dynamic binary analysis for tracking pointer data

John Galea* and Mark Vella

University of Malta
Department of Computer Science

1 Introduction

The examination and monitoring of binaries during runtime, referred to as dynamic binary analysis, is a widely adopted approach, especially in the field of security and software vulnerabilities. Fundamentally, it provides one with a means to understand and reason about binary executions. There are various applications of dynamic binary analysis, including vulnerability analysis, malware analysis, and Web security.

One technique typically employed to perform dynamic analysis is taint analysis, which revolves around inspecting interesting information flows [3]. In this approach, taint marks are associated with values that are (1) introduced via defined sources and (2) propagated to other values to keep track of information flow. Marks may also be removed (untainted) once a defined sink has been reached. In addition, taint checking is also carried out in order to determine whether or not certain runtime behaviours of the program occur. The properties describing how taint analysis is performed, i.e taint introduction, propagation and checking, are specified by a set of rules referred to as a taint policy. One convenient way to define taint rules is in the form of operational semantics rules, as it avoids ambiguity issues. Rule 1 specifies the general form of a taint rule used in this paper. Given the current machine context of the program $\Delta$ and a statement, the rule specifies the end result, after the computation has been carried out.

\[
\text{computation} \\
\{\Delta\} (\text{taintstate}) \text{instruction} \leadsto (\text{taintstate}')
\]

Rule 1: General Rule

2 A taint policy for UAF vulnerability detection

The main aim of the overall research is to detect Use-After-Free (UAF) vulnerabilities, which are caused by dereferences of dangling pointers (pointers that point to freed memory). Based on the work carried out by J. Caballero et al. [2], this work focuses on using taint analysis to monitor flows that introduce, move and delete pointers referring to heap memory. Ongoing contributions include defining a clear taint policy for tracking heap pointers and implementing an online prototype for detecting UAF vulnerabilities.

As an example, consider the taint introduction rule (rule 2). It defines the taint source as calls to heap memory allocation functions. The pointer $m$, which stores the returned

* The research work disclosed in this publication is partially funded by the Master it! Scholarship Scheme (Malta).
value \( v \) of the function call, is associated with a taint label structure \( t_2 \) that holds information about the allocated memory. The link between the pointer and the taint label are established by making use of two global maps, namely a forward map \( \tau \) and a reverse map \( \pi \). More specifically, \( \tau \) is responsible for mapping pointers to their respective taint labels, whilst \( \pi \) maps taint labels to a list of shared pointers, which refer to the same memory block. The benefit of the latter map is that a list of pointers associated with the same taint label can be efficiently retrieved without the need to iterate over the forward map.

\[
\triangle, \tau \vdash ret \downarrow (m, t_1) \quad \pi'' = \pi[t_1 \mapsto m] \\
t_2 = (Ptr, v, \triangle((pc))) \quad \tau' = \tau[m \leftarrow t_2] \quad \pi' = \pi''[t_2 \leftarrow m] \\
\triangle, \tau, \pi, alloc\_call(dst, v, ret) \leadsto \tau', \pi'
\]

**Rule 2 - Pointer Introduction Rule**

Taint labels propagate upon the execution of \texttt{mov} instructions, as defined in the rule 3. The computation of the rule mainly involves associating the taint label \( t_2 \) mapped by the source address \( m_2 \) with the destination address \( m_1 \).

\[
\triangle, \tau \vdash dst \downarrow (m_1, t_1) \quad \pi'' = \pi[t_1 \mapsto m_1] \\
\triangle, \tau \vdash src \downarrow (m_2, t_2) \quad \tau' = \tau[m_1 \leftarrow t_2] \quad \pi' = \pi''[t_2 \leftarrow m_1] \\
\triangle, \tau, \pi, mov(dst, src) \leadsto \tau', \pi'
\]

**Rule 3 - Move Propagation Rule**

### 3 Implementation using Dynamic Binary Translation

A prototype that implements the taint policy for pointer tracking has been built as a client using the DynamoRIO tool [1]. Since instrumentation on binary instructions is required to carry out taint analysis during runtime, the DynamoRIO tool [1] was found to be ideal, as it allows one to monitor, as well as manipulate, executed binary instructions without the need to modify the code of the application under examination. In essence, the DynamoRIO client acts as an intermediary and is placed between the application and the underlying operation system. By copying an application’s original instructions to a code cache one block at a time, changes to instructions can be easily conducted. The DynamoRIO prototype monitors for calls to memory allocation and deallocation functions. Moreover, instructions such as \texttt{mov} and \texttt{push} are also examined in order to carry out taint propagation as described in rule 3.

### References

3. T. A. Edward J. Schwartz and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In IEEE Symposium on Security and Privacy (SP ’10), 2010.
Cloud and mobile security assurance
A memory forensics approach

Mark Vella
Department of Computer Science, University of Malta

Cloud computing nowadays constitutes a substantial portion of new enterprise IT spending [4]. Whether signing up for applications deployed as a service, or outsourcing hosting concerns to third parties, cloud computing is taking us back to the days of the data center. Virtualization is a key enabler technology that enables consolidation of hardware utilization and provides the notion of elastic computing whereby hardware resources are allocated on demand [8]. Mobile devices represent a natural fit to this change in the technology landscape, be it a means to compensate for their limited storage or a way to fully justify investment in cloud computing, thereby fueling the bring-your-own-device (BYOD) concept [1].

This new scenario is uncharted territory for security assurance, which attempts to provide a measure of trustworthiness of security-critical objects provided by third parties. In this case, enterprise information is being entrusted to cloud providers and to personal employee mobile devices. When assuring IT infrastructure, code scanning and penetration testing have become established audit practices. However, the sufficiency or even relevance for the cloud/mobility computing infrastructure has to be questioned. Recent global security incidents, namely Heartbleed [5] and Shellshock [10], show that vulnerabilities can go undetected for years despite existing practices. The case of targeted attacks gets even more complicated [13]. Relevance is also of concern. In the case of cloud computing: should enterprise simply trust the cloud platform’s security stack? If the answer is negative, how should a company arrange for a penetrating testing exercise on the cloud-hosted, possibly multi-tenant, infrastructure? Mobile devices complicate the situation even further. One factor that sets them apart from the rest is their hardware constraints. Therefore, should the lack of installed scanning software fail security audits or should these devices be exempt? This decision seems to present one of those impossible practicality-security trade-offs.

The argument being put forward in the proposed research direction is that when the big bug or the resourceful attacker strikes, graceful recovery is key. The intuition is that information retrievable from kernel/user memory should trigger recovery. Periodic or trigger-driven physical memory dumps from virtual machines and mobile devices can be used to assess the current security state across an IT infrastructure. The value proposition is that memory dumps are expected to be comparatively much smaller than disk or network dumps, and can provide a practical solution. On the other hand, the potential effectiveness is still left for exploration. An effective solution should provide a shift from a situation where suspicion of a security breach cannot be verified to one where
not only verification is made possible but that also allows for a timely recovery to minimize impact.

The use of memory forensics is not new for the domain of computer security. Malware forensics is a case in point [7], however the predominantly manual and non-structured approach poses the main limitation for assurance. Forensic guidelines suggest for example that process structures in kernel memory could help spot backdoor installations, while investigating kernel data structures such as system service descriptor tables could disclose the presence of rootkits. Security assurance tools require that the whole process be fully automated in order to enable a practical audit process. For this to happen existing guidelines require evolving into proper assurance methods having rigorous measures of effectiveness that can be tied to security risk levels. In the longer term more substantial challenges have to be addressed. Mobile devices may not be always accessible for on-demand memory dumps while their hardware constraints could hinder complete acquisition. Furthermore, the memory forensics-based assurance process should go beyond merely detecting the presence of malware, but rather the entire ‘who/when/where/how’ forensic spectrum needs answering. This way all damage can be identified, infection vectors closed down and any attack sources identified so that trust in the system can be fully regained. Some of the challenges involved require searching for exploitation residue that could be present in memory. Also, instructions from code sections combined with register and data sections content could enable back-tracing to prior program states of forensic interest. Finally, complete situation rectification could require a generic exploit mitigation mechanism to be put in place rather than a specific patch. Any (memory) forensically derived information could potentially serve as basis for such mitigation.

Existing research efforts provide promising starting points. Leverage of virtual machine technology to provide cloud security assurance has been investigated in terms of tamper-proof virtual machine snapshots [12]. There are also various examples that demonstrate how virtualization facilitates the implementation of various security mechanisms, e.g. control flow integrity (CFI) [2]. Work in automated dynamic analysis of malware also makes use of memory forensic techniques [14]. The problem of automated crash dump analysis also starts off with a memory dump and tackles the problem of program state back-tracing [6], whilst object-code analysis techniques also make use of runtime data structures e.g. call stacks [3]. Finally, memory forensics should complement disk-based forensics [9] and network traffic analysis [11]. Correlating the different forensic levels should be part of the overall research direction in order to maximize the available tool-set for investigators. With proper adaptation and combination with the aforementioned malware forensic guidelines, these existing techniques could pave the way towards a security assurance process that fits today’s enterprise computing needs.
References

1. Bridges, V.: BYOD skyrockets in popularity for 2013 — here are the stats to prove it (Dec 2013), http://vbridges.com/2013/12/20/byod-skyrockets-popularity-2013-stats-prove/
4. Gartner: Gartner says cloud computing will become the bulk of new it spend by 2016 (Oct 2013), http://www.gartner.com/newsroom/id/2613015
Improving the Gherkin Specification Language using Aspect-Oriented Techniques

John Aquilina Alamango, Christian Colombo, and Mark Micallef

University of Malta

In the highly dynamic markets in which software customers operate, it is crucial that the software development process is able to incorporate the customers in the feedback loop, supporting the evolution of specifications and ensuring that software is according to the customers’ requirements.

A specification language frequently used for this purpose is Gherkin — a very simple language with three main keywords: **Given**, **When**, and **Then**. The semantics of these keywords are loosely defined as *given some precondition, when a particular event occurs, then some postcondition is expected to hold*. Other than these three keywords, the specification writer can use natural language and it is then up to the developer to translate the specification into executable tests.

While Gherkin has become a mainstream tool in contemporary software development, one cannot help but notice a significant problem: the simplicity and flexibility which have made Gherkin so popular amongst non-technical industrial customers, are the same aspects which makes the resulting scripts highly repetitive. Each given-when-then statement — known as a *scenario* — will typically be written over and over again albeit with just a different parameter.

---

1: Feature: Testing Country Charges
2: Scenario: Test Incoming Phone Call Fee
3: Given I am receiving a phone call from Australia
4: When I answer the call
5: Then I should charged at 0.00c per minute
6: Scenario: Test Incoming Phone Call Fee
7: Given I am receiving a phone call from United Arab Emirates
8: When I answer the call
9: Then I should charged at 0.00c per minute

---

After having considered a number of industrial case studies, we noted two main repeating patterns: *(i)* As in the above example, a number of scenarios where identical except for one or more parameters *(ii)* In other cases, apart from the parameters, a scenario might have one or more extra postconditions which are not present in the other scenarios.

In view of the identified recurring patterns in Gherkin scripts, our aim is to constrain the flexibility of Gherkin just enough so that the user can be afforded just enough automatic support in the writeup of the scripts. In the rest of the paper we describe our approach in tackling the two identified issues.
Set theory to the rescue Our first part of the solution is to introduce the notion of collections of objects — sets — enabling the script writer to collate related objects into a set and then allowing one to refer to all the individual set elements at once. Furthermore, we observed that using set operators, one could define sets as follows:

1: EUCountry = {Malta, UK}
2: NonEUCountry = {Australia, United Arab Emirates}
3: AnyCountry = EUCountry + NonEUCountry

Resulting in reduced repetition in the definition of the scenarios:

1: Scenario: Test Incoming Phone Call Fee
2: Given I am receiving a phone call from <NonEUCountry>
3: When I answer the call
4: Then I should charged at 0.00c per minute

Aspect-oriented programming [2] While set theory concepts greatly helped in structuring Gherkin specifications, a number of examples couldn’t benefit because the scenarios of the set elements would gave a single line which would be different. To this end, we propose an aspect-oriented programming extension to Gherkin, allowing the injection of the extra line to be done automatically. For instance, if an entry should be added in the log for every call (irrespective of whether it originates from a NonEUCountry or an EUCountry), then using the code below, we would be checking the log for every scenario without adding extra lines.

1: Matching: {"When I answer the call"}
2: [ Then call should be registered in call log ]

When we applied the proposed set-theory-extended-Gherkin to our industrial case studies, in three out of the four cases¹ we observed a significant reduction in the size of the script without any major impact on their readability. Perhaps more contentious is the aspect-oriented programming extension where more than one script file would have to be consulted in order to read a single scenario. This however could be alleviated with the right tool support where any matching aspects could be highlighted whilst browsing the corresponding scenario in another window. In the future we aim to provide such an editor which would ultimately enable the user to go from the Gherkin flavour presented here to the original Gherkin, providing backward compatibility while making the task of script writing easier.

References


¹ Two of the industrial case studies were performed at Ascent Software and Betclic respectively, while the other two were performed at companies who do not wish to disclose their identity.
A DSL for Business Intelligence Monitoring

Christian Colombo\textsuperscript{1}, Jean Paul Grech\textsuperscript{1}, and Gordon Pace\textsuperscript{1}

University of Malta

Social media has provided the business community with a unique and unprecedented opportunity to engage with their customers, critics, competitors, etc. Yet, this comes at the costs of continuously monitoring the various fora where the business name may be mentioned, where questions may be posted, where products may be compared, etc. Effectively dealing with social media in a world where a maximum of a few hours is the expected response time, is a challenging task.

Focusing in particular on Facebook, a typical business would have its own page as well as a strong interest in pages on which their products may be discussed or advertised. Typical events which are relevant for a business might include any mention of the brand or a product, an advertising post by a competitor, a comment by a customer (particularly if negative or if a question), and so on. To make the task of checking for these events manageable, dashboards \cite{1, 3, 2} are available allowing users to specify events of interest so that a notification is received when such an event is detected (e.g., a notification when more than five comments are awaiting a response).

The problem with existing tools is that while they allow the specification of a number of events of interest, they do not offer the flexibility which might be required by the business in question. For example one might want to prioritise the notifications in order of urgency (e.g., a comment from a new customer might is given precedence over that of an existing customer); alternatively one might want to group them into batches (e.g., a notification per five comments unless a comment has been posted for more than three hours). Such flexibility usually comes at the price of a tailor-made solution which is generally expensive both if developed in house or by a third party.

One way of allowing a high degree of flexibility while providing an off-the-shelf solution would be to present a simple interface which would allow a business intelligence analyst the flexibility to express the desired events for notification. These would in turn be automatically compiled into Facebook monitors without any further human intervention. While an automated compiler would struggle to handle natural language descriptions and a non-technical business analyst would struggle with a programming language, a domain-specific language presented to the user as a controlled natural language (CNL) \cite{6} may act as an intermediary: it provides the feel of a natural language but constrains the writer to particular keywords and patterns. Furthermore, using standard techniques to allow the user to write their scripts using particular user-interfaces (e.g. pull-down menus or fridge magnet style) bypasses the problem of syntax errors, thus making their writing more accessible to a non-technical audience.

Based on interviews with two business analysts, such a CNL should allow the user to specify patterns such as: (i) \textit{Create an alert when the service page has a post and the post contains the keywords fridge, heater, and freezer}; (ii) \textit{Create an alert when my page has a post and the post is negative and the post has 10 likes}. 

\textit{Create an alert when the service page has a post and the post contains the keywords fridge, heater, and freezer}: (ii) \textit{Create an alert when my page has a post and the post is negative and the post has 10 likes}. 
A number of the events proposed by the domain experts were significantly challenging or outright impossible unless further data is supplied:

**Sentiment analysis** While measuring the number of likes is easy, detecting whether a post is negative or positive is harder. To cater for this, we have used a ready-made sentiment analysis library\(^1\) which analyses posts and reports whether the post is positive, negative, or neutral.

**Distinguishing between users** If employees post regularly on the business’ page, then one would typically want to filter out their posts for the purposes of alert raising. Unfortunately, this functionality cannot be provided unless further information is available (e.g., a list of the employees).

Once a prototype CNL was designed, it could have potentially been compiled into any executable programming language. With the aim of keeping the translation as simple as possible, we translated the CNL into a intermediary specification from the runtime verification domain. Runtime verification [4, 7] is typically used for bug detection by matching the execution flow of a program to patterns encoded in terms of formally specified properties. Translating our CNL into the formal specification accepted by the runtime verification tool Larva [5] and using a simple adapter to present relevant Facebook events as method calls in the control flow of a program, we were able to detect Facebook behaviour through runtime verification software.

When it came to evaluating the CNL, we focused on the practical aspects of non-technical users using the CNL. To this end, a questionnaire was used to interview a number of users from a local company. The questionnaire was split into two main parts as follows:

**Understanding the CNL** The participants were presented with a number of statements expressed in the proposed CNL and they were expected to explain the meaning of the statements in natural language without any supporting documentation. In each of the four cases, more than two-thirds of the respondents explained the statement correctly although these include those who left out minor details in the description.

**Expressing statements in the language** The second exercise involved the opposite: given a textual description, respondents were expected to write it in terms of the CNL without any support. This proved to be harder but around 60% of the respondents got a good answer or a close enough answer which would have probably been correct with the help of a graphical user interface providing syntactic checks and auto-complete functionality.

In the future, we aim to continue evaluating the CNL with more users, enabling us to fine tuning it and improve its implementation.

**References**


\(^1\)http://sentiment.vivekn.com
Towards achieving efficient Runtime Enforcement for Component Based Systems

Ian Cassar* and Adrian Francalanza

Department of Computer Science, University of Malta

Formally ensuring the correctness of component-based, concurrent systems is an arduous task, mainly because exhaustive methods such as model-checking quickly run into state-explosion problems; this is typically caused by the multiple thread interleavings of the system being analysed, and the range of data the system can input and react to. Runtime Verification (RV) [7] is an appealing compromise towards ensuring correctness, as it circumvents such scalability issues by only verifying the current system execution. Runtime Enforcement (RE) [5] builds on RV by automating recovery procedures once a correctness violation is detected so as to mitigate or rectify the effects of the violation. We can therefore see Runtime enforcement as made of two parts: (i) Verification (Monitoring), and (ii) Recovery Actions.

Most RV and RE frameworks work by synthesising online monitors from properties specified in terms of a formal language, and then execute these monitors alongside the system to observe its behaviour. Online monitoring usually comes in two monitoring types, ie, Synchronous and (ii) Asynchronous Monitoring. In synchronous runtime monitoring, whenever the monitored system generates a trace event, an event notification is forwarded to the monitor and the system blocks, waiting for the monitor to inspect the event and send an acknowledge signal signifying that the event did not violate any of the correctness properties and allowing the system to continue executing at the point where it was blocked. By contrast, in asynchronous monitoring, the system does not pause when trace events are generated; instead, events are kept in a buffer and processed by the monitor at some later stage, thereby decoupling the execution of the system from that of the monitor.

Both monitoring types have their merits. Synchronous monitoring guarantees timely detection of property violations/satisfaction since the system and monitor execute in lockstep; this facilitates the runtime enforcement of properties, where remedial action can be promptly applied to a system waiting for a monitor response. Although asynchronous monitoring may lead to late detections, it is less intrusive than its synchronous counterpart. The associated overheads also tend to be lower [3] than its synchronous counterpart, as it is more of a natural fit for settings with inherent notions of asynchrony. Asynchronous monitoring also poses a lower risk of compromising system behaviour in the eventual case of an erroneous monitoring algorithm that forgets to acknowledge back or diverges (ie, enters an infinite internal loop) since, in asynchronous monitoring the system execution is decoupled from that of the monitor.

Issues relating monitor instrumentation are particularly relevant to actor-based [1, 8] component systems. Synthesising asynchronous monitors as actors is in accordance

* The research work disclosed in this publication is partially funded by the Master it! Scholarship Scheme (Malta).
with the actor model of computation, requiring independent computing entities to execute in decoupled fashion so as to permit scalable coding techniques such as fail-fast design patterns [4]. Certain code organisation design patterns (such as *supervisors*) are already prevalent in actor based languages and technologies such as Erlang [2] and Scala [6]. However, there are cases where tighter analyses through synchronous monitoring may be required, particularly when timely detections improve the effectiveness of subsequent recovery procedures. Crucially, the appropriate monitor instrumentation needs also to incur low runtime overheads for it to be viable.

**Lowering overheads for Runtime Enforcement**

Violating correctness properties might lead to serious, irreversible consequences. For this reason, it is ideal to timely detect violations, thus ensuring that the system’s incorrect behaviour is immediately corrected. Although Synchronous monitoring provides such timely detections, in a component-based system it is not ideal to block all concurrent components whenever the system generates an event which requires monitoring. Conversely, a more efficient approach would be to keep monitoring interactions as asynchronous as possible, and only block an individual system component whenever it generates a critical event, where critical events are system actions that may directly or indirectly lead to a violation.

**Choosing the Appropriate Enforcement mechanisms**

When implementing enforcement actions for component-based systems, we must consider the concurrent nature of these systems. Such enforcement actions should allow the user to apply enforcement actions only on the misbehaving components, thus leaving the original system behaviour intact as much as possible. Typical enforcement actions should coincide with the model employed by component based systems. For instance, typical enforcement actions for component-based should include (*i*) killing misbehaving components, (*ii*) restarting individual actors and (*iii*) clearing mailbox content.

**References**

Mobile Erlang Computations to Enhance Performance, Resource Usage and Reliability

Adrian Francalanza and Tyron Zerafa
University of Malta

1 Introduction

A software solution consists of multiple autonomous computations (i.e., execution threads) that execute concurrently (or apparently concurrently) over one or more locations to achieve a specific goal. Centralized solutions execute all computations on the same location while decentralized solutions disperse computations across different locations to increase scalability, enhance performance and reliability.

Every location affects its executing computations both directly (e.g., the lack of a resource may prohibit a computation from progressing) and indirectly (e.g., an overloaded location may slow down a computation). In a distributed environment, application developers have the luxury of executing each computation over its best-fitting location; the location (a) upon which the computation can achieve the best performance and (b) which guarantees the computation’s livelihood. Ideally, the decision to execute a computation over a location instead of another also load-balances the use of available resources such that it has the least impact over other computations (e.g., a computation should not execute over an already overloaded location further slowing down its computations).

Application developers can only execute computations over their best-fitting location if their distributed programming language provides abstractions that allow them to control the locality of computations both before they are started and during their execution. In the rest of this document, section 2 briefly justifies why these two forms of locality control are required and section 3 outlines the issues that arise, and will be tackled in the talk to be held at CSAW 2014, by them.

2 Control over Locality

The ability to determine the requirements of a computation (and hence its best-fitting location) before it starts executing depends on the computation’s type (i.e., whether it is of a functional or reactive nature).

Any computation of a functional nature is deterministic (i.e., its execution can be completely established from the computation’s executed code and its initial inputs). For instance, the execution of the factorial algorithm is of a functional nature since it cannot be affected by any other computation; this is just a mathematical function. Thus, it is possible to application developers to determine its requirements (e.g., processing power) and initialize it over the best location (e.g., the most lightly-loaded location).
Any computation of a reactive nature is non-deterministic (i.e., its execution depends on the input it receives from other computations). For instance, the execution of a publish-subscribe server is of a reactive nature since its behaviour can only be determined at runtime as it starts receiving requests. This inability to foresee the execution of a reactive computation before it starts executing limits application developers from foreseeing all its requirements and initializing it over its best-fitting location (e.g., although frequently communicating computations are best co-located on the same location to reduce communication overheads, it is impossible to start the execution of publish-subscribe server closer to its clients’ since these can only be determined at runtime).

Reactive computations can never be initialized over their best-fitting runtime systems since their requirements can never be determined before their execution starts. Furthermore, the runtime system best-fitting a (functional or reactive) computation may change throughout the computation’s lifetime (e.g., the factorial algorithm may impose a huge processing load on its location which affects the execution of the publish-subscribe server) or fail (e.g., the failure such location would terminate both the server and factorial computations). In such cases, the performance and reliability measures of a computation can be greatly enhanced through mobility: the ability to dynamically relocate an executing computation from one location to another. This additional level of flexibility over immobile computations allows application developers to dynamically adapt their solutions to the ever-changing nature of distributed systems (e.g., by relocating the executing server computation closer to its new clients that are determined at run-time so as to reduce communication overhead), enhance load-balancing (e.g., by relocating the executing server computation to a lighter-loaded location) and increase fault tolerance (i.e., by relocating both executing computations away from a location that is about to fail to a more stable location).

3 Talk Overview

The ability to control the locality of a computation both before it is started and during its execution gives rise to a number of important questions. Erlang, a language designed specifically for distributed systems, only offers rudimentary support for code and computation mobility which greatly limit the flexibility of this language. The talk that will be presented at CSAW 2014 will discuss some of the issue surrounding mobility in the context of Erlang and outline how such functionality can enhance performance, resource usage and reliability of Erlang-developed solutions. Specifically, this talk will try to answer the following questions: (1) What is the desired semantics of a computation that is initialized over (or relocated to) a remote location? (2) How can mobile computations be referenced (and located) at runtime? (3) Can mobile computations preserve the same message-passing semantics supported by Erlang for immobile computations? (4) Do conventional synchronous error-handling abstractions suffice for mobile computations? (5) Is it feasible to relocate computations?

This work is carried out following the award of a STEPS scholarship which is part-financed by the European Union - European Social Fund (ESF) under Operation Programme II - Cohesion Policy 2007-2013.
Enabling Usage of Cloud-based Applications When Offline

Joshua Ellul
University of Malta

1 The Problem

The Internet is ubiquitous. Office and home environments are increasingly making use of cloud computing platforms to facilitate day to day tasks. The main cloud related challenges highlighted by academia include: server consolidation, energy management, data security, storage technologies and data management [2]. Cloud related research either does not mention issues pertaining to Internet connectivity or assumes that Internet connectivity is available (and rightly so), and as pointed out by [1]: “No Internet, no cloud computing - it’s that simple.” However, situations often arise whereby connectivity is limited or unavailable (such as whilst travelling) and access to the cloud is still required.

2 Current State of Affairs

A common solution to provide offline access to cloud-based applications is to implement a desktop based version of the cloud-based application. While desktop application based solutions do provide a sufficient means of accessing such applications offline, they require a substantial amount of effort from the development teams to implement/re-implement/port.

Adobe AIR allows developers to package HTML, JavaScript, Adobe Flash, Flex and ActionScript into applications that can be used offline. Whilst, the technique does allow for code reuse (between the online site and the desktop version), it requires that end-users install each offline web-based application that they require to use.

Google Gears, no longer available or supported, is a retired project that aimed to provide a framework that allows for offline browsing of web pages. It achieves this by caching (static) resources (HTML, Javascript, images, etc.) locally in the client browser and then serving them later when required. It also provides a local SQLite database on the client where any required offline data can be synced to. Caching and serving static pages does not meet the dynamic requirements of web applications. Therefore, application developers using Google Gears are required to implement the data model’s synchronisation logic on the client side (in Javascript); which must also be exposed from the server side (via HTTP requests); and also implement all logic that is to be used offline in the client-side part of the application (that is in the client-side Javascript). Therefore, to allow for offline usage a developer must either duplicate the logic both in the server-side and client-side, or else opt for client-side code only. That said, server-side code is
often intended to run on the server for good purposes (vicinity to data, bandwidth re-
duction, etc.). Google Gears was abandoned in aim of standardising the capabilities into
HTML5. HTML5 inherently is client-side code and therefore the server-side/client-side
discussion above applies to HTML5.

3 Vision

Developers should be primarily concerned with the business logic specific to their appli-
cation (as much as possible). Therefore, in this project we are investigating techniques
to facilitate a framework for building cloud-based applications that allows for their usage when offline: without having to implement a desktop-based version; without having to implement client-side versions of server-side code; and without requiring application developers to implement synchronisation and offline/online switching logic themselves.

To achieve this undoubtedly changes would be required to the client web browser,
in a similar to Google Gears. Initiatives behind HTML5 are heading towards standardis-
ation and therefore the features implemented in HTML5 (and future versions) are likely
to be supported by all browsers. The changes required will therefore be implemented
by the browser developers.

As a first prototype towards the goals outlined above, we are implementing function-
ality on the client side that could eventually be integrated into the client web browser.
We plan to achieve this by implementing language constructs that will define what parts
of the code and data models can be used whilst offline.

References

2. Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research chal-
Extracting Monitors from JUnit Tests

Christian Colombo\(^1\), Jonathan Micallef\(^1\), and Mark Micallef\(^1\)

University of Malta

A large portion of the software development industry relies on testing as the main technique for quality assurance while other techniques which can provide extra guarantees are largely ignored. A case in point is runtime verification [1, 3] which provides assurance that a system’s execution flow is correct at runtime. Compared to testing, this technique has the advantage of checking the actual runs of a system rather than a number of representative testcases.

Based on experience with the local industry, one of the main reasons for this lack of uptake of runtime verification is the extra effort required to formally specify the correctness criteria to be checked at runtime — runtime verifiers are typically synthesised from formal specifications. One potential approach to counteract this issue would be to use the information available in tests to automatically obtain monitors [2]. The plausibility of this approach is the similarity between tests and runtime verifiers: tests drive the system under test and check that the outcome is correct while runtime verifiers let the system users drive the system under observation but still has to check that the outcome is as expected.

Notwithstanding the similarities, there a significant difference between testing and runtime verification which make the adaptation of tests into monitors a challenging one: tests are typically focused on checking very specific behaviour, rendering the checks unusable in a runtime verification setting where the behaviour is user-directed rather than test-specified. Test specificity affects a number of aspects:

**Input** The checks of the test may only be applicable to the particular inputs specified in the test. Once the checks are applied in the context of other inputs they may no longer make sense. Conversely, the fewer assumptions on the input the assertion makes, the more useful the assertion would be for monitoring purposes.

**Control flow** The test assertions may be specific to the control flow as specified in the test’s contest with the particular ordering of the methods and the test setup immediately preceding it. The control flow is particularly problematic if the assertion is control flow sensitive (e.g., checking the sequence of method calls called).

**Data flow** The test may also make assumptions of the data flow, particularly in the context of global variables and other shared data structures — meaning that when one asserts the contents of a variable in a different context, the assertion may no longer make sense.

**External state** A similar issue arises when interacting with external stateful elements (e.g., stateful communication, a database, a file, etc.): if a test checks state-dependent assertions, the runtime context might be different from the assumed state in the test environment.
In view of these potential limitations, most unit tests do not provide useful information for monitoring purposes (due to their narrow applicability). Thus, one approach in this context would be to build sifting mechanisms to select the useful tests while avoiding others which would add monitoring overheads without being applicable in general. Unfortunately, deciding which tests are general enough for monitoring use is generally a hard if not impossible task to perform automatically since the assumptions the developer made when writing the assertion are not explicit. Our solution was to explicitly ask the tester for the assumptions made. In practice this would however be impractical.

A better approach might be to help the user express the assumptions the test makes up front whilst constructing the tests. Another way of looking at it would be to create help the testers create monitors first, i.e., the checking part of the test, and then create a test for the monitor. Later, upon deployment, the driving part of the test would be discarded and only the monitoring part would be kept.

In the future we aim to explore this further on two counts:

- Exploring tests (other than unit tests) which make less implicit assumptions on the context in which they run. For example, higher level tests such as Cucumber\(^1\) tests are typically implemented for generic input with no assumptions on the order in which the tests are to be run.
- Possibly be creating a domain specific language which facilitates the specification of monitors and tests at the same time, allowing the user to switch between tests and monitors at any time.

References


\(^1\) http://cukes.info/
Investigating different Instrumentation Techniques in the context of ESB Runtime Verification

Gabriel Dimech¹, Christian Colombo¹, and Adrian Francalanza¹
University of Malta

With the increasing popularity of the service-oriented architecture, enterprise software is increasingly being organised as a number of services interacting over an Enterprise Service Bus (ESB). The advantages of this arrangement are numerous, not least the modularity and flexibility it affords and the ease with which multiple technologies can be integrated [2].

Yet, the highly dynamic nature of ESBs — where components can be introduced, replaced, or withdrawn transparently — brings with it a number of challenges, particularly to ensure the correct overall behaviour of the ESB [4]. In a flight booking ESB application, such as that depicted in figure 1, back-end airline services may be swapped at runtime, potentially causing unexpected behaviour. The reason for this may be that during the testing phase not all banking services may have been tested thoroughly or that the service experiences an update without the flight booking application being notified.

One way of ensuring that the ESB performs as expected is to check its behaviour at runtime rather than attempting to preempt the possible scenarios the ESB might find itself operating in. While this approach, known as Runtime Verification (RV), introduces an overhead, it ensures that anything which occurs at runtime adheres to a correctness specification [3]. In the context ESB applications, one main concern is to ensure that each incoming message is routed correctly [4]. For instance in the flight booking example, each request must be successfully routed to the correct airline and banking services before an acknowledgement is sent to the customer.

The correctness specification in the context of RV is typically specified in a mathematical notation and the runtime verifier is synthesised automatically. This introduces

Fig. 1: Flight Booking ESB Application
more confidence that the verifier is itself correct. However, other than correctness concerns, RV also raises overheads concerns which might have a negative impact on the performance of the ESB [1].

Of particular concern in this regard are the overheads related to intercepting relevant events which the runtime verifier would consider for correctness. This is because while the checking itself can be delegated to other processing resources, detecting and transmitting events to the verifier cannot. Through this work we attempt to investigate different options which can be considered for ESB events gathering by comparing the resulting overheads for each approach at runtime.

We start by identifying the different points at which relevant ESB events can be intercepted. In particular, we identify Mule ESB on which to perform our research, mainly due to its popularity amongst enterprises. Mule is a Java based open source ESB supporting XML specifications of Flows which may be used to configure SOA-based applications [5]. This arrangement exposes three different levels of event interception:

- One may intercept events at the source code level using techniques such as Aspect-Oriented Programming (AOP). This is by far the most commonly used technique for runtime verification of Java systems [6].
- At a higher level, one may include the verifier in the XML flow specification and divert copies of the messages to the verifier component for scrutiny.
- Finally, at the opposite end of the spectrum, the ESB depends on lower-level communication protocols such as HTTP, JMS, FTP etc. to connect components [5].

Thus, one could intercept such communication through a proxy and the verifier is able to intercept events for checking.

Each of these modes of events gathering provides its advantages: Intercepting events at the source code level provides full visibility of the ESB, including internal actions which might not otherwise be visible through the other modes. On the other hand, intercepting messages at the Flow specification level is an approach which allows the user to specify correctness properties more easily. Finally, the proxy approach decouples monitoring code from the ESB, making it easier to dynamically change which information is intercepted whilst reducing direct impact on the ESBs resources.

Notwithstanding these aspects, in this study we choose to focus on the performance issues of each approach. More precisely we consider two aspects of performance: the resources required to support runtime verification and the impact of this resource take-up on the end user experience.

**Resource Consumption** In this regard, we consider the CPU and memory used for each of the approaches.

**User Experience** To estimate the impact on user experience we measured the latency, i.e. the duration of time needed for a request to get a response, and the throughput, i.e. how many requests can be handled for a particular period of time.
Fig. 2: Resource Consumption Metrics

Fig. 3: User Experience Metrics

References

Challenges faced when Forcing Malware Execution down Hidden Paths

James Gatt, Mark Vella and Mark Micallef
University of Malta

1 Background

Dynamic Malware Analysis involves the observation of a malware sample at runtime, usually inside a sandbox, whereby probes are used to detect different actions performed by the malware in order to categorize its behaviour.

However, Dynamic Analysis is limited in that it can only observe a single run of the malware at a time, and there is no way of telling whether the run demonstrated the complete set of behaviours contained in the malware. Exploitation of this drawback is on the increase by malware authors as the presence of hidden and trigger-based behaviours has become more widespread\(^1\).

2 Unlocking Hidden Behaviour in Malware

Existing work on unlocking hidden behaviour takes an execution path exploration approach with the aim of maximizing precision by excluding infeasible paths and executing paths under correct runtime values, while also keeping performance at scalable levels. Symbolic execution in the form of concolic testing is one of the more popular approaches, as it generates inputs to explore as many feasible paths in the program’s execution tree as possible\[^1\]. However, it is difficult to scale to real-world malware binaries, as its use of SAT/SMT\(^2\) solvers for constraint solving is computationally expensive, which downgrades performance\[^1\]. Moreover, symbolic execution suffers from a number of known shortcomings that malware authors can take advantage of.

An alternative is the use of forced sampled execution techniques such as flood emulation, which explores a program’s execution tree in a depth-first fashion, enforcing execution iteration limits on blocks of code, and forcing execution down paths that might not normally be taken at runtime\[^2\]. While sacrificing some precision, as analysis could end up exploring infeasible paths\[^1\]\[^2\], we have chosen this approach for our work as it significantly improves the performance, and thus the scalability of analysis\[^1\]\[^2\].

3 Challenges

While flood emulation is an effective analysis technique, in practice there are situations where forcing code execution blindly down paths can lead to problems. Here we dis-

\(^1\) http://www.fireeye.com/resources/pdfs/fireeye-hot-knives-through-butter.pdf

\(^2\) Satisfiability/Satisfiability Modulo Theories
cuss a few such situations, while demonstrating how they can also be problematic for concolic execution.

3.1 Jump Tables
Jump tables in a binary usually result from the use of switch statements in the source code. A switch statement allows for the divergence of control flow into many possible paths, based on the value of some variable. At a lower level, this results in the creation of a jump table having an address entry for each possible path. Deducing the number of possible destination addresses in a jump table is a problem in itself. Moreover, jump tables generally use the switch variable as an index in order to calculate the address of the path to be taken on the fly, with the final control flow transfer usually taking the form of an unconditional jump to the destination address. This effectively flattens the control flow of the program and both flood emulation and concolic execution will only end up unlocking a single path, the one taken during a particular execution.

3.2 Blocking Behaviour
Any action that puts a program into a suspended state can be described as blocking behaviour. An example is blocking calls, which include any call to a function that halts program execution until the function returns, for example a network socket waiting for data to return from a remote server. Other cases include malware that waits for a certain number of user mouse clicks before continuing execution, or temporary suspension of a thread using sleep functionality. In all the above cases, flood emulation and concolic execution are stopped in their tracks until the malware resumes execution.

3.3 Program/System-wide State Corruption
Any non-adherence to the assumed state of a system or program at any point during execution infers state corruption. This might affect the program-wide state, for example trying to bind a network address to a socket that has not been created, or the system-wide state, for example trying to access a file that does not exist. Such sections of code exhibit path sequence sensitivity, and the order in which functions are executed is essential. Thus, forcing code execution down such paths even if one of the steps fails, as done during flood emulation, leads to executing infeasible paths, and does not reveal anything as the entire code section will not work. In the case of concolic execution, which strictly adheres to feasible paths, if any step fails, it will stop analysis from continuing further down that particular path.

References
Improving Android Security through Real-time Policy Enforcement

Luke Chircop*1, Christian Colombo2 and Gordon J. Pace3
University of Malta

The use of the Android operating system has become a very popular option with a vast variety of mobile devices. This popularity means that companies and other users are more likely to consider using android devices. Naturally there will be users and companies concerned with how their Android devices are used. Therefore, some sort of device management is required.

Let us consider a coffee distribution company that has employees visiting its customers to showcase new products and take orders. Such a company would need to provide its employees with portable devices containing sensitive data about their products and customers. Therefore, the company would want to limit access to such data to only authorized applications or users. It could also want to disable the android market or not allow untrusted applications from running on the mobile device. Another possible scenario could be that of having parents concerned with how their children use their mobile devices. It is well known that children love to play games, therefore a parent might want to control the amount of hours per day that they could spend playing on their mobile device. They could also want to make sure that the browsers that their children use, filter out bad websites. Parents could also want to control how many messages and phone calls their children make.

Unfortunately such checks or control over mobile devices cannot be achieved with the stock versions of Android. Instead techniques such as runtime verification can be used. This technique consists of introducing a runtime monitor that observes the behaviour of an executing process to be able to determine if it violates any predefined properties.

One common approach that has been implemented by various tools such as Weave Droid [1], RV-Droid [2] and Aurasium [3] focus on application-based monitoring. This approach consists of instrumenting applications that are usually downloaded from the android market with monitoring code before they are installed on the device. The instrumented monitoring code will then be able to observe the applications’ behaviour while executing and report any property violations that it encounters. Therefore with this approach, parents could write properties to raise warnings for situations such as when a browser allows its user to request a URL which could potentially be harmful to the user or device. Properties could also be introduced to limit the file/folder access for applications running on the device to safeguard against any attempts to leak sensitive data.

* The research work disclosed in this publication is partially funded by the Master it! Scholarship Scheme (Malta)
data (useful for company mobile devices).

Although it is able to provide a good monitoring framework, it has some limitations. One of which is the fact that a user could easily remove an instrumented application from the device and install an un-instrumented version to be able to bypass all the checks. For companies that provide its employees with devices containing sensitive data this is highly unwanted since the un-instrumented applications could then gain access to sensitive data. Furthermore, there are some properties that cannot be checked by this approach. For example, let us consider a parent that does not want her children to send more than 100 messages per hour. Such a property cannot always be correctly checked since application-centric monitors observe the behaviour of applications independently of each other. Therefore, in cases where more than one application capable of sending messages is installed on the device, the user would be able to send more than 100 messages per hour without a property violation being reported.

In our proposal, we are exploring a different approach to runtime verification. Instead of monitoring the applications separately, we are going to be introducing a device-centric approach. This will allow us to observe events at a system-wide level hence allowing us to monitor user behaviour, communication between applications, application installations, etc. To achieve this goal, a tool is going to be developed that will generate a loadable kernel module and event handling code which will be instrumented inside the dalvik virtual machine as shown in figure 1. The instrumented operating system would then need to be installed on the device that requires runtime monitoring.

The loadable module will be loaded by the Android kernel and act as an oracle to determine if a property has been violated or not. Therefore, to be able to check for a property violation such as not allowing a user to send more than 100 messages per hour, it would be required to instrument the send message function inside the dalvik virtual machine. Here for each event fired, the instrumented code would inform the loadable kernel module which would then determine if that action violates the property.

To be able to evaluate the proposed solution, we are first going to identify a number of scenarios that can be expressed as device-centric properties. These properties will then be passed on to our developed tool which will automatically generate and instrument a stock Android operating system. The operating system will then be installed on a corresponding device. Once the instrumented operating system is installed on the device, a number of actions will be carried out to be able to determine if the monitoring introduced does observe and detect any property violations whilst allowing good behaviour. Using this approach, we hope to provide the mobile device users with an extra layer of device management helping them have more control of their own device.
References