CSAW 2013

Day 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:30</td>
<td>Security Session</td>
<td></td>
</tr>
<tr>
<td>09:00</td>
<td>JG</td>
<td>Towards a Tunable, Sandbox-Independent Approach for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exploring Hidden Behaviour in Malware</td>
</tr>
<tr>
<td>09:40</td>
<td>GP</td>
<td>Implicitly-Enforced Contracts</td>
</tr>
<tr>
<td>10:20</td>
<td>coffee</td>
<td>Monitors and Types Session</td>
</tr>
<tr>
<td>10:50</td>
<td>AF</td>
<td>A Theory of Monitors</td>
</tr>
<tr>
<td>11:30</td>
<td>JM</td>
<td>Typing Actors using Behavioural Types</td>
</tr>
<tr>
<td>12:10</td>
<td>Ixaris</td>
<td>Using RV for calculating duration of page displays</td>
</tr>
<tr>
<td>12:40</td>
<td>lunch</td>
<td>Erlang Session</td>
</tr>
<tr>
<td>14:10</td>
<td>TZ</td>
<td>Towards an Abstraction for Remote Evaluation in Erlang</td>
</tr>
<tr>
<td>14:50</td>
<td>ET</td>
<td>Equivalence proofs for Erlang Refactoring</td>
</tr>
<tr>
<td>15:30</td>
<td>coffee</td>
<td>Runtime Detection Applications Session</td>
</tr>
<tr>
<td>16:00</td>
<td>MV</td>
<td>Detecting web server take-over attacks through verification actions</td>
</tr>
<tr>
<td>16:25</td>
<td>CC+GD</td>
<td>Applying runtime verification techniques to an ESB</td>
</tr>
<tr>
<td>17:05</td>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>
Towards a Tunable, Sandbox-Independent Approach for Exploring Hidden Behaviour in Malware

James Gatt, Mark Vella, and Mark Micallef

Department of Computer Science, University of Malta
{jamesc.gatt.06 | mark.vella | mark.micallef}@um.edu.mt

Introduction: Malware analysis is the reverse engineering of malicious binaries in order to explore and extract their complete behaviour for protecting against attacks and to disinfect affected sites. The dynamic analysis of malware inside sandboxes is useful since it removes the need for the analyst to look into the malware code itself. However, this approach could end up disclosing no behaviour at all if faced with trigger-based malware. The two main categories of trigger-based malware are the environment-specific type (for example, time bombs that only run when the date and time are right) and the externally-stimulated type (a recent case waited for a certain number of mouse clicks by the user before continuing execution[2]). Existing work that investigates how to unlock hidden behaviour in malware takes an execution path exploration approach with the aim of maximizing effectiveness by increasing both the path coverage along with the precision, the latter of which is increased by excluding infeasible paths and executing paths under the correct runtime values. In these terms, Symbolic Execution fits the bill. This technique uses constraint-solving to map out each execution path as a conjunction of all the values needed at each conditional branch in order to reach a certain point in the execution tree.

Issues: However, while encouraging results have been reported in work such as that by Brumley et al.[1], it is also true that due to its constraint-solving approach, symbolic execution is costly in relation to its overall performance, lowering its efficiency, measured in terms of the time elapsed versus the amount of hidden behaviour found. Existing work tries to increase efficiency by introducing heuristics, however, the implications of this in terms of effectiveness remain unspecified. Symbolic execution also has a number of inherent limitations, some of which have been tackled in existing work and others which have not, which degrade its overall effectiveness. Additionally, the majority of existing approaches use System-Wide Instrumentation, as it allows access to both kernel and user-level code at runtime, while also allowing for the creation of system-wide ‘snapshots’ which can be reverted to when backtracking. While this works to great effect, it also means that the solution is tied to a particular sandbox or full-system emulator, in a world where sandboxes usually come with hefty price tags. Sandbox Independence would require a shift to Process-Level Instrumentation, which introduces a new set of limitations that need to be overcome.
**Aim and Objectives:** The main aim of this project is to build on existing work in order to provide a solution for automated malware analysis that is capable of uncovering hidden behaviour in malware, but is also tunable in terms of its efficiency versus its effectiveness, while also being Sandbox Independent. Specifically, the objectives set for the scope of this work include: (1) the full exploration of environment-specific trigger-based malware; (2) a move towards Process-Level Instrumentation, which will require overcoming the resultant loss of control of any kernel-level code as well as the possibility of saving the system-wide state for backtracking; (3) a system that can be tuned in terms of the effectiveness-efficiency tradeoff; (4) a system that should be able to map a malware’s behaviour to the sequence of system library calls issued by the malware at runtime.

![Sandbox Diagram]

**Fig. 1: Proposed Solution**

**Solution:** Our proposed solution is presented in Figure 1 above. In terms of the objectives stated above, the malware will be launched by a Dynamic Binary Instrumentation (DBI), capable of running in any sandbox, and allowing us to control the malware at runtime through its Process Control Interface (PCI), provided by the underlying operating system. This approach allows us to provide system-wide snapshots for backtracking, while also taking away the need to rely on kernel-level code. Tunability will be introduced by also considering a lightweight alternative to Symbolic Execution which can still be used to detect hidden behaviour, at the cost of effectiveness. For this purpose, we have chosen Flood Emulation[3], which forgoes the constraint-solving approach, and resolves path exploration at the code block level. Thus, malware analysis can be tuned between both configurations, across the effectiveness-efficiency tradeoff.

**Evaluation:** We propose to evaluate our work as follows. The effectiveness of DBI as an equivalent to system-wide emulation will be measured by analysing a custom binary that requires backtracking in order to unlock all its hidden behaviour (for example, one which deletes and recreates files). Evaluating the tunability of the analysis configurations as well as the exploration of trigger-based malware is currently work in progress as there are a number of possible approaches (for example, testing purpose-built synthetic binaries as opposed to different categories of real malware). In each case, the effectiveness of analysis will be determined by the accuracy of the behaviour extracted from the malware.
Conclusion: The framework for this project is currently under construction, with WinAppDbg\(^1\) being used for the DBI and Flood Emulation components, and BAP\(^2\) (Binary Analysis Platform) being used to handle the Symbolic Execution side.

References


\(^1\) http://winappdbg.sourceforge.net/
\(^2\) http://bap.ecn.mun.ca/
Synthesising Implicit Contracts

Gordon J. Pace\textsuperscript{1} and Fernando Schapachnik\textsuperscript{2}

\textsuperscript{1} University of Malta
\textsuperscript{2} Universidad de Buenos Aires, Argentina

Abstract. In regulated interactive systems, one party’s behaviour may impose restrictions on how others may behave when interacting with it. These restrictions may be seen as implicit contracts which the affected party has to conform to and may thus be considered inappropriate or excessive if they overregulate one of the parties. In [5], we have characterised such implicit contracts and present an algorithmic way of synthesising them using a formalism based on contract automata to regulate interactive action-based systems. In this presentation, we outline the problem and future extensions of the work we are currently exploring.

1 Introduction

Consider the contract that binds a customer and a bank, which stipulates that opening new accounts is free of charge. And yet, at the moment of opening an account, the bank requires the release of personal information and allowance to send the customer promotional material. The bank is not strictly breaching the contract, but maybe it is asking “too much”. Can this “too much” be quantified? As another example consider an airline, which compensates for missed connections due to delays by providing the traveller with food and lodging. However, the airline has a policy of not providing this service unless the customers explicitly demands for it. In a way, the airline is turning its unconditional obligation of providing aid into a conditional or restricted one: given that the customer asks for help, support will be provided.

In interactive systems involving different parties, the behaviour of each party is inherently affected by the behaviour of the others. In particular, other parties may restrict or enforce behaviour resulting in the affected party having to behave in a different manner than if it were to act independently. In interactive systems regulated by contracts, each party thus may be seen to be restricted through the explicitly agreed upon contracts, but furthermore by the implicit constraints induced through the nature of interaction and the other parties’ behaviour. These implicit constraints can be seen as ‘invisible’ (or unspoken) agreements or contracts a party has to agree to and adhere to if they choose to participate in the interaction.

As the imposed behaviour gets stricter, it drifts from being compliant up to the point of being close to breaching the contract. Can this drifting be measured? As a first step, in this paper we develop techniques so that, given two interacting parties bounded by a contract, we can infer the implicit contract being enforced by one party on the other.

This approach is useful also during contract negotiation: given multiple service providers, a party may base her choice of provider not only upon the signed agreement, but also taking into consideration the implicit contract imposed on her.
2 Results

In [5], we have formalised the notion of implicitly enforced contract — corresponding to a contract which one party has to satisfy due to its interaction with the other party. For instance, if the bank party requires synchronisation over an action register before allowing the client party to engage in an action transfer, the implicitly enforced contract would include an obligation on the client to perform register before proceeding.

To formalise these notions, we have used contract automata [6], a formalism based on the notion of synchronising automata, that provides well understood and clear semantics to model interaction among parties. An automata-based formalism allows to model not only deontic formulae but also the choices that each party has and how decisions affect the other parties.

Contract automata give us a natural definition for a partial order of contract strictness. Using this notion, we have presented an algorithm to calculate the maximal implicitly enforced contract which can be proved to be unique.

3 Future Directions

Although contracts are a long-covered topic in deontic literature (e.g., [2–4]), we are not aware of work on synthesising implicit contracts. Process calculi (e.g., [1]) or even petri nets (e.g., [7]) have been used before to model “contracts”. However, the term contract is used there in the sense of interface, as way to guarantee services interoperability, but they are not rich enough for other types of contracts because they do not support deontic operators.

Although the concept of implicit contracts is interesting in itself, it becomes more interesting when one starts to compare this implicitly enforced behaviour to what a given contract demands, as a tool to highlight not only potential breaches but also subtle differences. Such a comparison permits the view of a continuum going from plain incompatibility to full compliance, with a lattice of possibilities in between — for instance, when one party restricts a permission of the other party by only allowing the permitted action after certain extra actions are performed.

To further explore implicitly enforced behaviour, synthesis for non-deterministic systems, is still an open problem. Future research directions also includes the dual of implicitly enforced behaviours, intrinsic well-behaviour: given the behaviour of a system, induce a contract which it obeys under all circumstances. Such a notion could, for instance, be useful to use to figure out how much more one can ask from the other party without them having to change their behaviour.

References

A Theory of Monitors

Adrian Francalanza
Department of Computer Science, University of Malta
adrian.francalanza@um.edu.mt

Monitor-Oriented Programming (MOP) [7] is a programming technique that augments a system with monitors, software entities that analyse and possibly affect the behaviour of the monitored system. The technique has been used extensively in Runtime Verification (RV), a lightweight formal method for improving system safety, reliability and dependability, that compares a system’s runtime behaviour against a set of requirements.

Monitored systems come in various forms. They range from organisations with a clear delineation between a system and its respective monitors [5, 9], others where this delineation is blurred [4], and others where monitors form part of the system itself, leading to hierarchical arrangements where monitors potentially monitor other monitors [13]. Some frameworks assume on a single monitoring entity [6, 1] whereas others consider monitor subcomponents [5, 3] or groups of monitors [13, 8]. Multiple monitors may execute in parallel i.e., not interacting with one another [13], concurrently [9], or in distributed fashion [8]. They may also be organised in an orchestrated or a choreographed manner and, when distributed, they may also migrate across locations and/or change configuration [8].

Although they have been used as a vehicle to analyse system behaviour, the monitors themselves have seldom been treated as an object of study. In this work we address this aspect, and strive towards developing a formal framework for comparing monitors. We limit our study to monitors that just observe systems, flagging an alert whenever a particular runtime behaviour is detected i.e., they do not interact with systems or try to modify their behaviour. More concretely, we aim to formalise a monitor preorder whereby, for arbitrary $m, n \in \text{MON}$:

$$m \subseteq n \text{ iff } \forall s \in \text{Sys.} \ m \text{ flags a behaviour for } s \implies n \text{ also flags that behaviour for } s \quad (1)$$

Such a framework can act as a unifying theory of monitors and serve as a method for comparing the various monitoring approaches discussed above. Being software entities themselves, our preorder can also be used as a formal basis for refactoring a monitor $m$ with another monitor $n$ that is more efficient or easier to maintain, without altering the set of system properties that are detected; this aspect is particularly relevant to automated monitor synthesis from logical/declarative specifications [12, 9]. Our preorder may also be used as a refinement methodology for monitors: in settings where monitor specification and implementation are described within the same formalism, e.g., [5], monitor $n$ could be envisaged as a correct implementation for a monitor specification $m$. Along these lines, our...
preorder can also be used to define a behavioural notion of monitor equivalence, \( \preceq \), obtained as the kernel equivalence of \( \subseteq \), i.e., \( m \preceq n \) iff \( m \subseteq n \) and \( n \subseteq m \).

Definition (1) brings to the fore a number of aspects to be considered. First, we need to determine what formalism to use to describe the monitors (and the systems that they monitor). We have decided to go for automata descriptions, which may be expressed syntactically as processes

\[ p, q \in \text{Proc} ::= \text{nil} \mid \alpha.p \mid p + q \mid \text{rec}.x.p \mid x \]

There are a number of reasons justifying this design decision. For starters, automata are abstract enough not to commit to any particular implementation platform or programming language, while still retaining a concrete algorithmic flavour. Automata (and variants such as Büchi automata) are often used as the target formalism in monitor synthesis [12, 1]. There is even a body of monitoring tools that works directly at the level of automata correctness specifications [5, 3]. Similarly, monitors that are given a semantics in terms of a Labelled Transition System (LTS) may also be viewed as automata [8, 9]. Finally, there is considerable work on monitor expressivity and monitorability that has already been conducted in terms of such a formalism [6], which would facilitate crossovers to and from our work.

Definition (1) forces us to consider alternatives relating to the statement \( m \) flags a behaviour for system \( s \), that bear effect on the semantics of our monitors. For instance, decisions on whether to instrument monitors synchronously or asynchronously impinges on our treatment of monitor divergences (infinite internal loops) i.e., whether to treat them as catastrophic or not. Concurrency and under specification in refinement settings introduce the possibility of having non-determinism in our monitor descriptions, which in turn opens up various possible interpretations for behaviour detection and flagging. For instance, we need to investigate whether we would like to limit ourselves to consistent detections, along the lines of a must-preorder [10] or a compliance relation [2], or whether we should aim towards a more generous may-preorder semantics [10].

Substantial effort needs to be devoted towards the development of tractable methods to reason about our monitor preorder, \( \preceq \), as well. Note that definition (1) quantifies over all possible system that can be analysed by the ordered monitors, which makes such preorders intuitive but, at the same time, hard to determine. We would like to investigate sound and complete characterisations for \( \preceq \) that are easier to work with from an algorithmic standpoint, such as trace-based and failure based denotational characterisations [10, 11].

References


Typing Actors using Behavioural Types

Joseph Masini¹ and Adrian Francalanza¹

Department of Computer Science, University of Malta
{joseph.masini.09 | adrian.francalanza}@um.edu.mt

The actor model is a computational model that assists and disciplines the construction of concurrent programs. It has been adopted by numerous programming languages e.g., Erlang [1] and Scala [3], and frameworks e.g., AKKA [5].

The model employs a number of concurrent single-threaded entities called actors, executing within their own respective local memory. Actors communicate with each other through asynchronous message-passing and reserve the ability to dynamically spawn additional actors during runtime. An actor is composed of three main elements, namely: i) a unique actor name employed for communication purposes; ii) an expression processed by the actor; iii) and a mailbox composed of a queue that orders received messages by time of arrival. Whereas any actor may write to a mailbox by sending messages to the respective actor, only the mailbox owner can retrieve these messages from the mailbox—this may be done selectively through a pattern-matching mechanism.

Figure 1 depicts a basic actor system consisting of three actors: a client cl, a server sr, and a bank bnk. It also shows the interactions between these actors, whereby the client sends a request for a particular service to the server, encoded as the message ⟨req, cl⟩; the service request includes a service tag, req, as well as the (unique) name of the client, to be used as the return address for the service. Before replying, the server checks with the bank whether the client is credit-worthy by sending the message ⟨chk, cl, sr⟩. The bank checks the client’s credit history and replies to sr with the message ⟨ok, cl, true⟩ if the client’s credit rating is good and ⟨ok, cl, false⟩ otherwise. If the bank’s reply is positive, the server sends a reply to the client containing the service, ⟨rply, service⟩.

Fig. 1. Simple client-to-server communication pattern.

In this work, we investigate type-based static analyses of actor systems to determine safety properties dealing with mismatches between messages sent and message received. Concurrency is one source of complication for such analysis, forcing us to consider all possible execution interleavings amongst actors: a server would typically receive requests from more than one client and it needs to ensure that multiple requests do not interfere with one another. Mailbox polymorphism is another source of complication, since different forms of messages may be received within the same mailbox. For example, in the system of Figure 1, actor sr receives two forms of messages, namely ⟨req, cl⟩ and ⟨ok, cl, bool⟩ (for some boolean value).

In practice, actor systems are even more complicated than the system of Figure 1. As it stands, this system may run into problems because the server represents a bottle-
neck for servicing throughput, and also a single point of failure. Figure 2 depicts an improved arrangement prevalent in actor systems, whereby the server delegates the service request task to a new actor which it spawns acting as a task handler, th. In this new arrangement, the server may remain responsive to other client requests, even when it is under heavy load from client requests. It is therefore the task handler that handles credit rating checking with the bank and completes the service request interaction with the client; importantly, the client is unaware that the service request was completed by the task handler instead of the server. Although beneficial, delegation decentralises control, introduces further process interleavings, and poses further complications to our proposed static analysis.

![Diagram of client-to-server pattern with task delegation.](image1)

The arrangement in Figure 3 takes this system a step further, introducing a level of backend redundancy, which may lead to improved responsiveness and better fault tolerance. More specifically, the thread handler consults with two banks regarding the credit-worthiness of the client and, as soon as it obtains a reply from one of them, it can complete the client request. This arrangement improves responsiveness because the client servicing depends on the fastest response from either bank; it is also fault tolerant because it can complete the client servicing even when one of the banks fails to return an answer.

![Diagram of client-to-server pattern with task delegation and component-message redundancy.](image2)

The added sophistication in Figure 3 complicates static analysis even further: in cases when both banks execute normally, the static analysis has to contend with ensuring proper message matching in the presence of redundant messages, while guaranteeing the absence of memory leaks. More precisely, when both banks return their replies, each iteration of a client request leaves residual messages that accumulate over time. For one, our static analysis must ensure that residual messages from the previous iteration do not interfere with the current servicing iteration. Moreover, the static analysis needs to ensure that redundant messages are disposed of at some point. In an actor setting,
this may be done by either explicitly consuming the message from the mailbox or else by terminating the respective actor, at which point mailbox contents are automatically garbage collected.

We plan to use behavioural type systems to statically reason about the actor systems discussed above. Although there has already been considerable work on this front [2, 6, 4], none of the existing work addresses satisfactorily the features discussed above. For instance, [6] applies session types to an Erlang calculus, but limits interactions to two parties, [2] applies a generalisation of conversation types to a Scala calculus but limits analysis to finite computations and [4] limits communications described by its type system to a static number of participants that is known upfront. By contrast, we plan to adapt this work to reason about potentially infinite actor computation involving communication between multiple actors that may be created and killed dynamically, some of whom may engage in redundant computation in order to achieve fault tolerance.

References

Towards an Abstraction for Remote Evaluation in Erlang

Adrian Francalanza and Tyron Zerafa

Department of Computer Science, University of Malta
{adrian.francalanza | tzer001}@um.edu.mt

1 Introduction

Erlang is an industry-standard cross-platform functional programming language and runtime system (ERTS) intended for the development of concurrent and distributed systems[1]. An Erlang system consists of a number of processes [2] (actors) executing concurrently across a number of nodes/locations. These processes interact with each other (mainly) through asynchronous messaging and are capable of creating (spawning) further processes, either locally or at remote locations.

2 Process Creation

In Erlang, a functional object (variable/value of type function) is a data type consisting of a set of attributes along a pointer to the function’s executable code. An Erlang process, which can only execute functional objects, starts by looking for the referenced executable code inside the underlying node’s directories and loads it into the code server. In a distributed setting, a process may request the execution of a functional object on a remote node; however, the executing (remote) location is not allowed to retrieve code from the original (requesting) node. Erlang assumes that during remote process creation the required code would be present on the executing node prior to execution initialization. A process would fail if it attempts to execute a function whose code is not defined on its (executing) node; alternatively, the end result would be different if the executing node holds executable code that differs from that residing at the functional object declaration location [1, 5].

Erlang’s remote process creation resembles the remote evaluation paradigm in which nodes demand the execution of code on remote locations through requests that contain the required code. We propose a solution that facilitates remote evaluation, i.e., transfers the (missing) required code during a remote process creation, while adhering to Erlang’s semantics and best practices. In the rest of this document, § 3 explains why existing Erlang support is not adequate to attain code migration while § 4 discusses considerations that arise in the design of our remote evaluation mechanism. § 5 concludes this document.
3 Inadequacies of the Existing Support

Erlang’s standard serialisation mechanism encodes data/ values that needs to be transmitted over a network into an intermediate representation known as the External Term Format (ETF). The intermediary representation of Erlang functions is composed of a number of attributes which include a symbolic link to the respective module’s binary file (called a BEAM file) containing the function’s code. Upon a remote execution request, the respective function ETF (with its symbolic references to the BEAM file) is sent to the remote node, assuming that the referenced BEAM file is present.

In order to overcome this limitation attributed to the serialization mechanism, Erlang provides two mechanisms that facilitate the dynamic linking/ loading of code modules/ binaries inside remote ERTSs. The simplest mechanism broadcasts whole modules onto entire Erlang clusters (a set of connected nodes) resulting in huge bandwidth usage spikes and superfluous memory overheads. On the other hand, the second mechanism transmits portable code resulting in a less-expensive finer-grained control over what’s loaded where.

At first these approaches may seem attractive, however, after a deeper analysis it becomes evident that these are far from complete. For starters, they lack any form of dependency analysis which has to be handled explicitly by end developers to ensure that all the required code is transferred. Furthermore, these mechanisms do not take into consideration the possibility of different code versions which are so critical in real life development environments. All these problems, coupled with the possibility to remotely execute higher-order functions (functions that accept other functions as argument) over remote nodes require a huge development effort from the end developers and necessitates a proper framework that manages Erlang code in such a heterogeneous distributed environment.

4 Considerations for Proposed Solutions

A solution to handle code management during remote evaluation can be programmed; however, as described in the previous section increases the responsibility and effort on the part of application developers, who would need to contend with the difficulties discussed. We propose a solution that abstracts over these difficulties and automates the functionality for code dependency analysis, code correspondence and code migration, in line with other proposed fine-grained code mobility approaches [3, 4]. This automation should aspire to mimic the behaviour of a local process execution in the presence of missing code using the least possible bandwidth and storage overheads.

The solution would need to determine a feasible unit of code migration to adopt. More specifically, whereas the unit of process creation is a functional object, the ERTS standard unit of code loading is the Erlang module. Issues may arise when, in order to remote execute a particular function whose code is not present at the destination node, an arbitrarily large module (containing the
required function) would need to be migrated and loaded; the problem could be more acute in the case of transitive function dependencies.

Conventions for how to migrate code would also need to be established. At one extreme, the solution may decide to migrate the missing code eagerly in one phase, once the missing dependencies are statically determined. Alternatively, code migration may happen incrementally in lazy fashion, whereby only the immediately execution functions are sent. The latter approach is in general more complex and may incur more bandwidth overhead. However it is able to use runtime information relating to code dependencies, e.g., code branches taken by the spawned remote actor, so as to minimise the code that is migrated—the function dependencies in branches that are not taken need not be migrated.

The proposed solution may even decide to adopt a hybrid model of code migration, that adapts according to the requirements of the nodes and that of the underlying network.

Ideally, the solution should also embrace the realities of distributed computing and adhere to the philosophy of the host language, i.e., Erlang. Failures such as nodes crashing and flaky node connections should not be ruled out by the proposed solution, which should in turn affect the underlying architecture and operations. For instance, in order to withstand a degree of failure, the proposed solution should be as decentralised as possible. Moreover, once the missing code dependencies are determined, the code need not be migrated from the source node; instead it may be obtained from another node having a faster or more reliable connection to the remote node where the processes is to be created.

5 Conclusion

We have argued why that the existing mechanisms for remote evaluations in Erlang is inadequate for a distributed setting with heterogeneous codebases. We then outlined possible requirements to consider for a language extension that addresses these shortcomings. We are currently working on a prototype that takes these suggestions into account.

References

Equivalence Proofs for Erlang Refactoring

Erica Tanti and Adrian Francalanza

Department of Computer Science, University of Malta
{erica.tanti.09 | adrian.francalanza}@um.edu.mt

Erlang [1, 2] is an actor-based programming language used extensively for building concurrent, reactive systems that are highly available and suffer minimum downtime. Such systems are often mission critical, making system correctness vital.

In industrial-scale systems, correctness is usually ascertained through testing, a lightweight verification technique trading analysis completeness for scalability. In such cases, a system is deemed correct whenever it “passes” a suite of tests, each checking for the correct functionality of a particular aspect of a system. This is also true for large Erlang systems: even when correctness specifications are provided, it is commonplace for Erlang developers to use testing tools, automating test-case generation from these specifications [6, 9].

Testing for concurrent systems is an arduous task. Since a concurrent system may have multiple execution paths—due to different thread interleavings—it is often the case that test runs fail to detect errors, only for them to crop up once the system is deployed. Because of this aspect, Erlang tests are often executed multiple times using tools that induce different interleavings [3], in the hope that enough execution paths are covered so as to be able to conclude (with a reasonable degree of certainty) that the error tested for is absent. To make matters worse, every time a system needs to be altered—because of either code maintenance, bug fixes or functionality enhancements—the entire testing procedure needs to be carried out again from scratch.

In most cases, the a code update is often expected to pass the same test suite that the previous code had originally passed (possibly extended by additional tests). This is particularly the case for refactoring, code restructuring that does not necessarily change functionality, but instead makes the code more readable, compliant to certain code practices, or more efficient. There are numerous tools assist or automate the refactoring process in Erlang systems [7, 8, 10]. However, none of these tools comes equipped with a guarantee that the substitute code preserves the behaviour of the code it is substituting, which can potentially violate correctness.

In this work, we strive towards a solution for this problem. We study a testing preorder [5, 4] for Erlang programs $P, Q$ and Erlang tests $T$. We limit our study to safety testing suites, i.e., suites of tests ensuring that nothing bad happens. Our safety testing preorder, $P \preceq_{\text{safe}} Q$, denotes that $P$ is as safe as $Q$, and is formally defined as:

$$P \preceq_{\text{test}} Q \quad \text{iff} \quad \text{for all } T \left( (P \text{ fails } T) \implies (Q \text{ fails } T) \right)$$
Note that, by the contrapositive, our testing preorder ensures that whenever \( Q \) passes a test \( T \), \( P \) also passes that test.

This preorder may be used as the semantic basis for the aforementioned Erlang refactoring tools. More specifically, in a setting where correctness means passing a suite of safety tests, a refactoring tool would be considered safe whenever it substitute a program \( Q \) with another program \( P \) that can be shown to be as safe as \( Q \), i.e., \( P \preceq_{\text{test}} Q \).

Unfortunately, reasoning about the preorder \( \preceq_{\text{test}} \), even when limited to safety tests, is generally non-trivial because its definition relies on a universal quantification over all possible (safety) tests. We therefore investigate a theory that facilitates reasoning about our testing preorder. In particular, we develop an alternative trace-based preorder \([5, 4]\) for Erlang programs; it relies on program traces \( s \), instead of tests, and would normally take the following general format:

\[
P \preceq_{\text{trace safe}} Q \iff \text{For all } s \left( (Q \text{ produces } s) \implies (P \text{ produces } s) \right)
\]

A trace based preorder is simpler to reason about because (1) interactions with a number of tests may be described using a single trace; (2) traces are somehow connected to the capabilities of the programs \( P, Q \) being analysed, whereas tests may not. The main result of our work would then be to prove a correspondence between the testing preorder, \( \preceq_{\text{test}} \), and the trace based preorder, \( \preceq_{\text{trace}} \).

However, the technical development does not follow directly from the work on testing preorders by Hennessy et al. \([5, 4]\), and certain characteristics pertaining to actor systems complicate the development of our trace-based preorders. For instance, actors often interact through asynchronous communication, which reduces the tests' power of observation. Put differently, asynchrony increases the number of trace executions that may lead a test to fail, thus making the above trace-based preorder too rigid. Another characteristic is actor persistence, meaning that once an actor is spawned, it is receptive to an infinite number of messages; this yields an infinite number of traces even for the simplest of programs, making trace-based analysis unwieldy.

Through a series of examples, our talk will explain the problems encountered when developing our testing theory for Erlang programs and discuss the potential solutions being investigated.

References

7. Li, H., Thompson, S.: Testing Erlang Refactorings with QuickCheck. In: IFL ’07. Freiburg, Germany (September 2007)
Detecting web server take-over attacks through objective verification actions

Mark Vella\(^1\) and Sotirios Terzis\(^2\)

\(^1\) University of Malta
\(^2\) University of Strathclyde, Glasgow

Attacks targeting web servers pose a major security threat. Typically prone to a mix of infrastructure and application-level security vulnerabilities, they serve as the lowest hanging fruit for intruders wanting to gain unauthorized access to the entire host network. This is specifically the case for ‘server take-over’ attacks, whose immediate objective is to gain unauthorized remote access to the host server, for example through shell-spawning, backdoor-ing or botnet joining\(^3\).

From attack/malware to exploit detection. The most common option to detect such attacks consists of recognizing attack packets or malicious binaries at the network and host levels. However, these detectors make use of misuse detection rules that can be easily evaded through obfuscation/polymorphism and are highly limited in matching attacks exploiting zero-day vulnerabilities. Recently, the research domain is witnessing a shift from this malware/attack-centered approach to one that focuses on the targeted application. Specifically, these what we call exploit detectors, operate by dynamically monitoring the execution of the potentially vulnerable application/platform for indications of successful exploits.

Analysis of runtime information either follows the classic misuse/anomaly detection dichotomy, such as recognizing known malicious system call sequences of parasitic malware [4] or detecting jump targets/control sequences considered tainted [5], as compared to recognizing anomalous HTTP-backend traffic pairings caused by SQL injection attacks [2], or goes one step further and outright change the execution environment to block successful exploit execution [3]. Overall, these detectors leverage the higher quality information obtained through dynamic analysis to generalize beyond exploits instances to an entire exploit category, and are also resilient to content obfuscation resulting in increased detection effectiveness. Furthermore, the dynamic verification of successful exploitation avoids false positives (FP) however at the cost of invasive instrumentation and high monitoring overheads. An aggregation of such detectors would be an obvious method to effectively protect from take-over attacks, though performance overheads and compatibility issues abound.

Problem definition. We aim for a dynamic analysis-based method that generalizes from known attacks over the objective dimension in order to detect web server take-overs. Specifically, the proposed solution is required to: 1) Combine

\(^3\) http://www.symantec.com/security_response/publications/threatreport.jsp
multiple relevant exploit categories in its detection scope; 2) Translate the high-level objective description to a low-level one in terms of events associated with the execution of the vulnerable web application; 3) Retain the polymorphic/zero-day attack resilience and low FP properties of dynamic analysis detectors; and 4) Not increase overheads beyond that of individual exploit detectors.

Proposition. We propose a solution that: focuses on attack rather than normal content to avoid FP; combines known exploit categories from an attack objective dimension into a single solution through causal relations; is verification-based as per existing exploit detectors; relies on externally observable dynamic analysis events so as not to impose intrusive instrumentation; and uses modified LAMBDA [1] to translate between a high-level detection heuristic and its low-level counterpart for the take-over objective. The result is an objective verification approach that verifies the objective’s success based on its pre-/post-conditions expressed in terms of dynamic analysis events, where: the pre-conditions are defined on process input, post-conditions are defined over events resulting from input processing associated with objective’s attainment, whilst the verification actions confirm the causal relation between the input and the events. We modify the LAMBDA language to fit this approach so that it reflects the single-step attacks and objective focus, where pre- and post-conditions describe the monitored process’s state, and verification and detection actions are fused together. The high-level detection heuristic “Attack input needs to inject code to setup the take-over and then either connects to a remote listening port, or start listening on a (newly-opened/reused) web port” gets the low-level translation:

objective \textit{WWW\_take\_over}(HTTPReq, Platform, WebApp\_Interpreter\_List, Interpreter\_List, WWWProcTree)

\hspace{1em} pre: injectable\_netcode(HTTPReq, [Platform : Interpreter\_List]) \lor codebase\_extending\_script(HTTPReq, WebApp\_Interpreter\_List) \\
\hspace{1em} post: net\_start\_listen(WWWProcTree, (Local\_IP, Local\_Port)) \lor \\
\hspace{2em} net\_connect(WWWProcTree, (Remote\_IP, Remote\_Port)) \lor \\
\hspace{3em} ((create(File) \lor modify(File)) \land interpretable(File, WebApp\_Interpreter\_List))

verification/detection: G1

\hspace{1em} where: action(G1) = ((Local\_IP, Local\_Port) \in import\_pairs(Injectable\_netcode) \lor \\
\hspace{2em} (Remote\_IP, Remote\_Port) \in import\_pairs(Injectable\_netcode)) \lor \\
\hspace{3em} contains(File, codebase\_script\_blocks(Codebase\_extending\_script))

The post-condition events can be used for the immediate recovery of a subverted system, whilst the HTTP request implicated in the precondition can be used to track the exploited vulnerability for long term recovery. The implementation relies on network/memory/disk forensic probes to supply the required events, and a purposely-built emulator to identify and extract information from potential instances of Injectable\_netcode and Codebase\_extending\_script. Ongoing work concerns the implementation of an experimentation test-bed that provides real-world traffic and a range of successfully executing attacks that are representative of the take-over objective.
References

Applying Runtime Verification Techniques to Enterprise Service Bus Architectures

Christian Colombo\(^1\), Gabriel Dimech\(^1\), and Adrian Francalanza\(^1\)

Department of Computer Science, University of Malta
{christian.colombo | gdim0002 | adrian.francalanza}@um.edu.mt

The increased usage of distributed systems has led to the problem of integration of IT systems that communicate via different protocols. In such setting, it is also typical for these components to be added/removed from the system at runtime, depending on requirements. These two characteristics give rise to integration challenges where systems should be able to communicate seamlessly with each other whilst being able to be easily integrated with an existing distributed system with minimal impact.

An ESB (Enterprise Service Bus) is a tool which attempts to address the above issues by integrating systems in a bus-like architecture where a central bus enables components to communicate via messages [5, 2]. This arrangement, transparently handling complex messaging functions such as transformation and routing (via routing patterns), enables the user to focus on the business logic, abstracting away communication concerns [6].

Despite facilitating orchestration of distributed components in a scalable manner, current ESBs provide little support for correctness guarantees of the overall system logic e.g. a booking component may only confirm the booking once the bank component has verified the payment details, and the airline component confirms that the dates specified are permissible.

Popular techniques for ensuring correctness, such as testing and model checking are not ideal for verifying the correctness of ESB systems due to the latter’s highly dynamic nature. For this reason, we propose to apply runtime verification [4, 3, 1] techniques, promising a scalable approach to checking all observed behaviour under any runtime circumstance — no matter how unpredictable this may be.

Design Options for a Runtime Verification Approach

There are a number of concerns which have to be considered when applying runtime verification techniques to ESBs: (i) **Dynamic updating of network** Due to the dynamic nature of ESBs, monitors should be able to tolerate a network of components which join and leave the network at will. (ii) **Expressivity** The formalism used to express the logic to be runtime verified should support the encoding of typical logic found in ESBs. (iii) **Execution efficiency** The verification code added to the ESB framework should not introduce prohibitive execution overheads. (iv) **Communication efficiency** Communication between
the potentially numerous components of the runtime verifier should not interfere with the rest of the messages. (v) **Privacy issues** The verification process should not lead to exposure of any private information across ESB components.

While there are numerous points on the design space of runtime verification applications to ESBs, in this short overview we focus on two main ones: the orchestration and the choreography approach. In the former, the verification of properties is done via a central monitor which is able to observe all the communication channels of the distributed system. Orchestration-based approaches are relatively straightforward to design and implement, as the verifier design does not require communicating with other verifiers. However, a disadvantage of orchestration-based approaches, is that since the monitor is in one central location, monitoring performance impact directly affects the ESB. Additionally, the network formed between components and the ESB is also impacted as information required for verification must be communicated over message channels. Figure 1 depicts an orchestration-based verification setup where three components A, B, and C are communicating events to a central monitor.

![Fig. 1. Orchestration-based monitoring](image)

Choreography-based verification involves dividing the verification code in separate sub verifiers designed to communicate with one another so that they are able to verify a property. One variant of choreography-based verification is to push the monitoring of properties at runtime onto the components forming the distributed system. In doing this, performance impact of verification on the ESB is decreased as the sub-verifiers on the components shall be performing most of the verification, only communicating with other verifiers when required. Performance impact on the network is also lessened due to the fact that the central verifier residing on the ESB now requires less information from the components for monitoring purposes. In addition, these monitors are able to verify local properties for the respective components in isolation from monitors residing on other connected components. However this approach requires that the remote components both allow and trust the local monitors to execute on their behalf. Distributing verifiers on remote components is usually only possible in a setting where the distributed system is controlled by a single organisation. One other variant of choreography-based verification is to apply sub verifiers on the message channels residing on the ESB rather than on the remote components. On
the one hand, this has the disadvantage of pushing the overhead onto the ESB infrastructure as in the case of the orchestration-based approach. On the other hand, having the verifier module split into sub-verifiers enables us to dynamically switch on and off parts of the verification network to keep the overheads to a minimum. Figure 2 shows a depiction of a choreograph-based verification setup.

Conclusion

The discussion presented in this short abstract does not cover all the criteria outlined in the previous section but gives an introduction to the issues involved in choosing the right design for a runtime verification setup. In the future, we shall be considering various design options in the context of real-life ESB applications with the aim of finding the right tradeoffs between expressivity, overheads, and correctness assurance.

References

CSAW 2013

Day 2

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:30</td>
<td>coffee</td>
<td>P2P Networks Session</td>
</tr>
<tr>
<td>09:00</td>
<td>AM+SA</td>
<td>Leveraging P2P networks to address the scenario explosion problem</td>
</tr>
<tr>
<td>09:40</td>
<td>SS</td>
<td>Pointcloud Decomposition for Scene Analysis and Understanding</td>
</tr>
<tr>
<td>10:10</td>
<td>KB</td>
<td>Collaborative rendering over P2P networks.</td>
</tr>
<tr>
<td>10:40</td>
<td>coffee</td>
<td>Industry Session</td>
</tr>
<tr>
<td>11:25</td>
<td>lxaris</td>
<td>Contention of data on highly concurrent systems</td>
</tr>
<tr>
<td>11:55</td>
<td>CCBill</td>
<td>Challenges and Problems within the Industry</td>
</tr>
<tr>
<td>12:30</td>
<td>lunch</td>
<td>Resource Constrained Systems Session</td>
</tr>
<tr>
<td>14:00</td>
<td>JE</td>
<td>A Complete Paradigm Independent Operating System for Resource Constrained Systems</td>
</tr>
<tr>
<td>14:40</td>
<td>MAC</td>
<td>Using Symbolic Execution for Equivalent Mutant Detection</td>
</tr>
<tr>
<td>15:20</td>
<td>coffee</td>
<td>Future Work Session</td>
</tr>
<tr>
<td>16:00</td>
<td>MM</td>
<td>Search-based software engineering</td>
</tr>
<tr>
<td>16:30</td>
<td>AS</td>
<td>The way forward for extending 'detectEr'</td>
</tr>
<tr>
<td>17:00</td>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>
Leveraging P2P Systems to address the Test Scenario Explosion Problem

Andrea Mangion¹, Sebastian Attard², Mark Micallef¹, and Conrad Attard²

¹ Department of Computer Science, University of Malta
{andrea.mangion.04, mark.micallef}@um.edu.mt
² Department of Computer Information Systems, University of Malta
{sebastian.attard.13, conrad.attard}@um.edu.mt

1 Introduction

Modern software development is characterised by a strong customer focus and electronic delivery mechanisms which make it very easy for customers to buy and install a vendor’s software. However, it also makes it very easy for customers to buy and install software from competing vendors and as such it is more important than ever for deployed software to be as correct and bug-free as possible.

Whilst certain types of testing can be done in the lab with a high degree of confidence that results will hold when the software is deployed in the wild, in reality software systems are subject to influence from whatever environment they end up being deployed in. Varying factors in customer environments can include operating systems, services packs, device drivers, network connectivity, resource usage by other software, and so on. Any variation or combination of these factors can lead to a situation where a system deviates from its expected behaviour. The problem is amplified even further on mobile devices whereby devices can move between different networks, interrupt apps for phone calls, have varying screen sizes, have user interference in the form of turning features on and off to preserve battery power, vendor-specific operating system code, and so on. A conservative calculation indicates that a software system can be subjected to tens of thousands of different scenarios. Even if one were to execute just one test case against each scenario, obtaining any form of realistic coverage is infeasible for even the most resource-rich organisations.

Cloud and grid infrastructures have been proposed as solutions to improving the scalability of software testing [1–3] since 2008. However, we argue that cloud computing systems are too homogenous and are not representative of real-world usage scenarios. We refer to this problem as the Test Scenario Explosion Problem.

2 Peer-to-Peer Systems

Peer-to-peer (P2P) systems are distributed systems which are usually associated with (1) a high degrees of decentralisation, (2) self-organisation and (3) an abundance and diversity of resources [4]. Although they are commonly associated with file sharing applications P2P systems have been used to leverage
the computing resources of cooperating users in order to achieve scalability and organic growth. These characteristics combined with its independence of a dedicated infrastructure and centralised control make P2P an interesting candidate paradigm for the solution of the test scenario explosion problem.

3 Hypothesis and Research Challenges

Our overarching hypothesis is that P2P systems can be leveraged to achieve a high level of scenario coverage within a feasible amount of time. That is to say that given a network of peers in which computing resources are made available, a developer can request execution of a test suite whilst placing desirable constraints (e.g. tests can be executed by users in Germany who own Android devices). Peers will then propagate this request appropriately such that the workload is shared amongst peers who satisfy the constraints with the developer subsequently receiving appropriate test results.

Whilst the idea is arguably interesting and credible, a number of challenges present themselves. The following is an incomplete but representative list of research challenges which make the problem non-trivial and interesting from both an academic and commercial point of view:

1. How can the problem be effectively reformulated into a distributed search problem such that known P2P algorithms can be reused effectively?
2. What is the best way for the P2P network to self-organise such that test suites can be propagated efficiently?
3. Given a particular network population, can a guarantee be provided that a test suite will be executed with a certain level of scenario coverage within a certain amount of time?
4. What incentive mechanisms can be utilised in order to encourage participation in such a system?
5. What are the security implications of executable code being transferred and deployed between peers?
6. How do peers negotiate obligations, permissions and restrictions between themselves?
7. How does one go about evaluating the effectiveness of such systems?

Whilst we are currently working on developing prototypes for both desktop and mobile devices, we would like to use CSAW 2013 as a discussion platform on the evaluation aspect of this research. Consequently, whilst the talk will consist of discussions about various design decisions that go into building such systems, we will be presenting options with regards to evaluation and soliciting feedback on the topic.

References


Over the past decade digital photography has taken over traditional film based photography. The same can be said for video productions. A practice traditionally reserved only for the few has nowadays become commonplace. This has led to the creation of massive repositories of digital photographs and videos in various formats. Recently, another digital representation has started picking up, namely one that captures the geometry of real-world objects. In the latter, instead of using light sensors to store per pixel colour values of visible objects, depth sensors (and additional hardware) are used to record the distance (depth) to the visible objects in a scene. This depth information can be used to create virtual reconstructions of the objects and scenes captured. Various technologies have been proposed and successfully used to acquire this information, ranging from very expensive equipment (e.g. long range 3D scanners) to commodity hardware (e.g. Microsoft Kinect and Asus Xtion). A considerable amount of research has also looked into the extraction of accurate depth information from multi-view photographs of objects using specialised software (e.g. Microsoft PhotofySynth amongst many others). Recently, rapid advances in ubiquitous computing, has also brought to the masses the possibility of capturing the world around them in 3D using smartphones and tablets (e.g. http://structure.io/).

In a similar fashion to digital photography, the widespread availability of hardware capable of capturing 3D information is also leading to the creation of
massive repositories of 3D data sets. These data sets, *point-clouds*, minimally consist of 3D coordinate values representing surface positions of the scene acquired (scanned). A wide variety of scenes can be scanned with the quality of the data captured depending on the acquisition method used. The office room (figure 1a) indoor scene was acquired using commodity depth sensors (in this case Microsoft Kinect) resulting in a relatively noisy and incomplete scene. Figure 1b shows the point-cloud of a section of the smallest of three temples in the Mnajdra pre-historic site. The acquisition process was carried out using professional grade 3D-scanners. Depending on the task at hand, manipulation (post-processing) of these point-cloud data sets usually requires extensive expertise in the use of CAD and modelling software. In this work we propose automated mechanisms to alleviate some of these tasks. In particular, we address the problem of scene understanding where meaningful structures (e.g. walls) and objects (e.g. chairs) are automatically extracted from raw point-clouds representing a variety of scenes. Previous work in the area has produced solutions which target specific environments, thus leading to assumptions limiting the adaptability of these techniques to other scenarios. Recent examples include [2] and [1]. In our case, we first tackled the problem of identifying generic structures within scenes [3] by partitioning point-clouds into connected surface segments, then generalised the solution to introduce object extraction.

Our current solution is split in two phases. Initially a training phase is carried out to concisely describe individual objects (e.g. tables, chairs, sofas, aeroplanes, etc.). Each model is described in terms of a set of feature description graphs storing surface connectivity and contextual information. As is custom in these learning scenarios the representation used tries to minimise the distance between objects in the same class (e.g. different chair models) and maximise that between classes. In the second phase, the target scene is first decomposed via a segmentation process to produce a meaningful set partition of surfaces, then a Markov decision process is applied on these segments in order to enumerate valid solutions. The presentation outlines current research progress, objectives and future directions.

References

Collaborative rendering over peer-to-peer networks

Keith Bugeja

Department of Computer Science, University of Malta
keith.bugeja@um.edu.mt

Abstract. Physically-based high-fidelity rendering pervades areas like engineering, architecture, archaeology and defence, amongs others [3][7]. The computationally intensive algorithms required for such visualisation benefit greatly from added computational resources when exploiting parallelism. In scenarios where multiple users roam around the same virtual scene, and possibly interact with one another, complex visualisation of phenomena like global illumination are traditionally computed and duplicated at each and every client, or centralised and computed at a single very powerful server. In this paper, we introduce the concept of collaborative high-fidelity rendering over peer-to-peer networks, which aims to reduce redundant computation via collaboration in an environment where client machines are volatile and may join or leave the network at any time.

1 Introduction

High-fidelity rendering makes use of physically-based quantities to compute light transport through a virtual scene, simulating the interactions between light and surfaces in order to compute energy levels in the scene. The equilibrium of light energy can be formulated mathematically via an integral equation known as the Rendering Equation [8], which has inspired a large variety of rendering algorithms and provides a definition of radiance at any point in the scene:

\[
L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\Omega} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos \theta_i| \, d\omega_i
\]

where \( p \) is a point in the scene, \( L_e \) is the emitted light, \( \omega_i \) and \( \omega_o \) are the incoming and outgoing directions of light, \( f \) is the bidirectional reflectance distribution function (BRDF) [10] [9], giving the proportion of light reflected from \( \omega_i \) to \( \omega_o \) at \( p \), and \( |\cos \theta_i| \), \((n_p, \omega_i, n_p \text{ being the normal at } p)\), is the foreshortening factor.

The main challenge in high-fidelity rendering is solving the rendering equation efficiently for any given scene. A number of approaches exist that can be broadly categorised into two methods: finite element (e.g., radiosity [6]) and point-based sampling (e.g., distributed ray tracing [4]). Finite element methods have some limitations which make them less than ideal for the goals of interactive rendering.
The general approach discretises surfaces into a finite set of patches, introducing difficulties when modelling arbitrary geometries. Moreover, solutions that go beyond the radiosity approach and model arbitrary reflections, present enormous problems due to memory consumption and computation times [11]. Most modern methods, such as Path Tracing [8] and Instant Global Illumination [12] use point-based sampling. Monte Carlo techniques, based on ray tracing, are employed for their applicability to arbitrary geometries and reflection behaviours [11]. Another advantage of Monte Carlo techniques is stability; they have the property that error bound is always $O(n^{-\frac{1}{2}})$ regardless of dimensionality. Point-based sampling methods lend themselves more to parallelisation than finite element methods.

A number of methods have been presented for solving the Rendering Equation within a distributed context. Some solutions use grid computing, desktop grids in particular, to provide high-fidelity rendering to a single client [1]. Rendering as a Service uses cloud computing to provide interactive rendering to multiple clients [2]. In both these works, potential imbalances in load that may occur due to resource volatility are handled by controlling and redistributing tasks to suit resource changes.

In many areas requiring interactive visualisation and collaboration, such as military and architectural simulations or online multiplayer videogames, participants interact with one another in a shared virtual world such as a battlefield, a building walkthrough or even a football pitch. Most of these applications are precluded from using high-fidelity physically-based rendering due to the large computational costs associated with such visualisation. [2] attempted to provide interactive physically-correct rendering by offloading expensive computations to the cloud. The system isolates resources allocated to a single client from other resources in the system, forgoing any communication or collaboration, notwithstanding the fact that sharing the same scene, isolated resources are carrying out duplicate computations.

In this work, we propose a method for exploiting the computation overlap of multiple clients visualising and freely interacting in a shared virtual scene. Furthermore, we assume that the number of constituent clients is not known beforehand; clients are volatile and may join or leave the network at any time. In particular, we put forward a system where computations can be encapsulated into uniquely marked transactions that can contribute towards a distributed shared state. The propagation of transactions neither makes use of an overlay network, nor requires the peer network be fully connected, but instead employs an epidemic paradigm, where peers gossip to exchange these transactions. We envisage the content of transactions to be arbitrary, although currently, a case study is being carried out with irradiance samples generated using [13]. During each exchange, peers also share details on their neighbours, which are then used to update and prune their respective peer neighbourhood directories.

The irradiance cache [13] is an acceleration data structure for caching diffuse irradiance samples from indirect lighting, applied to the distributed ray-tracing algorithm. The algorithm exploits spatial coherence by interpolating these irradiance samples for a given neighbourhood to estimate irradiance in specific
Distributed high-fidelity rendering regions, thus dramatically reducing render time. Searches within the irradiance cache are carried out to locate valid samples that can be used for irradiance interpolation; in order to speed up these searches, an octree is incrementally built: the diffuse irradiance value is stored and the tree topology updated.

We are looking at extending this work to cater for ordered transactions. For example, in static scenes, the update order of the irradiance cache is irrelevant. However, in dynamic scenes, knowledge of whether a transaction happened before another or not is required in ensuring consistency in the generated output. As such, we are looking at ordering each transaction using arrays of logical clocks [5] to generate a partial ordering of events and detect causality violations. Moreover, given the large number of participant, we are also investigating the application of some form of importance sampling for pruning the vector clock data structure to reduce bandwidth usage.

References

Using Symbolic Execution for Equivalent Mutant Detection

Mark Anthony Cachia\textsuperscript{1} and Mark Micallef\textsuperscript{1}

Department of Computer Science, University of Malta
{mark.a.cachia.09/mark.micallef@um.edu.mt}

Mutation Testing is a fault injection technique used to measure test adequacy score by generating defects (mutations) in a program and checking if its test suite is able to detect such a change. However, this technique suffers from the Equivalent Mutant Problem [3].

Equivalent mutants are mutants which on mutation retain their semantics [3]. Thus, although equivalent mutants are syntactically different, they remain semantically equivalent to the original program [3]. An automated solution which decides equivalence is impossible, as equivalence of non-trivial programs is undecidable [8, 3]. The fact that the Equivalent Mutant Problem is undecidable usually means that human effort is required to decide equivalence [3]. Equivalent mutants are the barrier keeping Mutation Testing from being widely adopted [3]. Moreover, in one study by Irvine et al [2], the average time taken for each manual mutant classification was fifteen minutes.

This work explores the application of Symbolic Execution to the Equivalent Mutant Problem. Symbolic Execution is a technique which enumerates a program’s paths and for each path outputs the conditions variables must satisfy for execution to reside in the path called \textit{Path Conditions} together with the output of the program at that path called \textit{Effects}. The combination of \textit{Path Conditions} and \textit{Effects} are known as the \textit{Path Condition ad Effects} pair or \textit{PCE}.

A Path Condition is an expression made up of symbolic values representing the conditions the variables must satisfy for execution to reside in a particular path [6, ?]. If at some point of execution the Path Condition resolves to \textit{false}, i.e. a path condition whose expression cannot be satisfied, the Path Condition is considered to be infeasible as it is mathematically impossible for execution to reach the path. Such PCEs are eliminated on the basis that it is impossible for the program to reach these paths [6].

Symbolic Execution can be used to approximate equivalence. This can be achieved by performing Symbolic Execution on versions of the same program and equating the outputted PCE pairs. The two most applicable variants of Symbolic Execution Differential Symbolic Execution[4] and Directed Incremental Symbolic Execution [5] were analysed however they were deemed to be not efficient enough for the Equivalent Mutant Problem. Hence, an algorithm called SEEM which is both efficient and effective to detect equivalent mutants was to be developed. SEEM works as follows. Initially the PCE pairs of the original version of the method being mutated are generated. The main reason why only the mutated method’s summary is generated is that as the code executed before the mutation has not been changed. Assuming there is no interleaving, determinism states that
both the original program and the mutated program will execute identically until before the mutated method is called.

After the summary of the original method has been generated, the mutation is performed and the PCE pairs of the mutated method is then generated. However, this process is done intelligently in order to improve efficiency. This is achieved by retracing the coverage of the original program on the mutant. That is, if a mutation is performed on the *then* branch of an *if* statement, only the paths passing through the *if* statement is explored. If the PCE pairs of the original program passing through the *if* statement are equal to the PCEs of the mutant, the mutant is considered to be equivalent. A proof of concept tool was implemented which makes use of the Microsoft Z3 constraint solver [1] used to determine feasibility and simplify Path Conditions.

Three separate investigations were performed to evaluate the Effectiveness, Efficiency and the extent to which the tool handles different levels of complexities in the Path-Explosion Problem. Various scenarios leading to equivalent mutants were encountered in the course of the work in which SEEM was able to correctly classify all but one. SEEM was compared to the state of the art tool Javalanche which performs Equivalent Mutant Detection by invariant violations [7]. When SEEM was used to classify the same mutants as Javalanche, SEEM was 89% accurate whilst Javalanche was only 65% accurate. The theoretical efficiency of SEEM was also studied. It was determined that the time and space saved by employing SEEM instead of traditional is $SEEM\ time\ savings = O(n \Sigma_{\text{program branches}} - \Sigma_{\text{method branches}})$. The final experiment was conducted in order to obtain typical running times of SEEM. Several experiments were performed. The highest running time recorded was that of just over nine seconds in a mutant which had over a thousand paths to explore.

From the results achieved, it was concluded that SEEM is a suitable technique to be used in the reduction of the Equivalent Mutant Problem.

References


Search Based Software Engineering

Mark Micallef
Department of Computer Science, University of Malta
mark.micallef@um.edu.mt

1 Motivation

Consider the following questions, which are posed by software engineers on a daily basis:

1. What is the smallest set of test cases that will cover all statements in this program?
2. What is the best way to organise classes and methods for this OO design?
3. What is the set of requirements that balances software development cost and customer satisfaction?

Whilst these questions seem to be addressing different problems, they do have some notable commonalities. Firstly, they form part of a large set of software engineering problems which can each be solved by a multitude of potential solutions. That is to say that if one were to ask the above questions to equally competent engineers, one would likely get back different yet correct solutions. Secondly, this class of problems is usually tasked with balancing a number of competing constraints. A typical example here is maximising customer satisfaction whilst keeping development costs low. Finally, whilst there is typically no perfect answer (and indeed no precise rules for computing the best solution), good solutions can be recognised.

When problems with similar characteristics were encountered in disciplines other than software engineering, they were solved with a large degree of success using search-based techniques. It was this realisation that gave rise to the field of search based software engineering.

2 Search Based Software Engineering

Search Based Software Engineering (SBSE) is the name given to a body of work in which Search Based Optimisation is applied to Software Engineering problems. Although the literature reveals earlier work in the area, the term itself was coined by Harman and Jones [1] in 2001, an event which seems to have legitimised the area as a sub-field of software engineering and led to an explosion of interest in the area.

Attempting to solve a problem using these techniques essentially requires reformulating software engineering as a search problem. That is to say that for a given problem, one needs to define:
– a representation of the problem which is conducive to symbolic manipulation
– a fitness function defined in terms of this representation

Consider the question “what is the smallest set of test cases that will cover all statements in this program?”. Assuming for the sake of example that the program is a method which takes two integers as parameters, one could propose an encoding whereby test cases are encoded as ordered pairs \((x, y)\) where \(x, y \in \mathbb{Z}\). Furthermore, the fitness function can be defined as the function which takes a set of integer pairs and returns a measure of coverage between 0 and 1.

\[ f : \mathcal{P}(\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{R} \]

The problem is thus reformulated as a search problem whereby the goal is to find the smallest list of integer pairs with a fitness function of 1.

Whilst the example is a simple one, it demonstrates that once you define an encoding and a fitness function, it simply becomes a matter of applying known techniques from search-based optimisation to find a solution to your problem. Of course, defining an encoding and fitness function is not always straightforward. In most non-trivial scenarios, one needs to deal with multiple goals to a fitness function, conflicting goals, and ones which are not easily encodable in an objective manner.

3 Talk outline

During my talk at the CSAW workshop, I will introduce the topic, discuss commonly used algorithms in this field and provide two or three concrete examples to illustrate what can be achieved using these techniques. I will then outline ongoing personal research work in this particular field.

References

The way forward for DETECTOR

Adrian Francalarza\textsuperscript{1}, Aldrin Seychell\textsuperscript{1}, Ian Cassar\textsuperscript{1}, Kurt Cutajar\textsuperscript{1}, Simon Said\textsuperscript{1}, and Blanche Schembri\textsuperscript{1}

Department of Computer Science, University of Malta
{adrian.francalarza | aldrin.seychell}@um.edu.mt

Monitor correctness is a prerequisite for the adoption of runtime verification as a lightweight formal technique ensuring program correctness. The tool DETECTOR [4,9] is a runtime verification tool synthesising correct monitors from $\mu$-calculus formulas [6] describing safety Erlang [2] properties. As opposed to previous work on monitor correctness [3,5], the correctness guarantees of the synthesised monitors in DETECTOR are close to the actual implementation; they are based on actual executable monitor translations in the form of Erlang programs, consider aspects impacting monitor runtime behaviour such as monitor instrumentation, and provide guarantees such as the absence of monitor interference during system execution.

In DETECTOR, one can specify safety properties such as

$$\max \left( X, \left( \left( \text{req} \land \text{ans} \land \text{ff} \right) \land \left( \text{req} \lor \text{ans} \lor X \right) \right) \right)$$

stating that a service request action req cannot be followed by two service answers, ans, subformula (1), and that this property must hold indefinitely after repeated sequences of req, ans pairs, subformula (2) in the context of (3). From this formula, the tool synthesises an Erlang monitor that checks for runtime violations of the property; where possible, the monitor employs concurrent processes to analyse the system, spawning them judiciously according to the behaviour of the system being monitored, so as to minimise overheads. The monitor instrumentation is asynchronous and minimally intrusive, requiring no changes to the source code of the system being monitored; this, in turn, facilitates adoption, which may even happen dynamically, while the system is already running.

Thus far, the focus of the work on DETECTOR has focussed on correctness [4,9]. The subsequent development stages plan to use this work as the foundation for extending the tool’s capabilities in terms of both expressivity and efficiency. We discuss these extensions below:

**Data:** We would like to introduce parametrisable actions and universal quantifications on data which would allow us to specify properties such as

$$\max \left( X, \forall x, y. \left( \left( \text{req}(x) \land \text{ans}(y) \land y \neq x + 1 \land \text{ff} \right) \land \left( \text{req}(x) \land \text{ans}(x + 1 \land X) \right) \right) \right)$$
Monitored actions may be parametrised e.g., req(x), and their respective values would then be instantiated at runtime. This allows us to specify symbolic constraints across actions such as that stating that whenever a system is requested a service with value x, it must reply with value x+1, subformula (4), or otherwise raise a violation, subformula (3).

**Beyond Safety:** So far the tool can only synthesise monitors for a syntactic subset of the µ-calculus describing safety properties [1]. In addition we would like to be able to synthesise monitors for formulas using other µ-calculus constructs, such as those describing monitorable liveness properties (co-safety) [8]. One example would be

\[
\min \left( X, (\text{end} \lor \text{tt}) \lor \text{req(\text{ans})X} \right)
\]

specifying that the servicing is correct but finite i.e., there exists a sequence of repeated (req, ans) interactions, subformula (6) after which the system terminates by producing a terminating action end, subformula (5).

**Synchronous and Hybrid Monitoring:** Despite its advantages, asynchronous monitoring may lead to late detections from the part of the monitor. We would like to explore the introduction of synchronous monitoring [7] (at the level of individual actions) and assess its impact in terms of overheads. The introduction of synchrony may also affect important properties that come for free in a completely asynchronous setting, such as the indirect effects of monitoring on system behaviour. More specifically, in a synchronous setting, the execution of the system and the monitor become intertwined and it would then be imperative that the monitor never enters infinite loops while the system is waiting for an acknowledgement to continue executing; in a completely asynchronous setting with fair executions, monitor divergences affect system performance but not behaviour.

**Performance optimisations.** The present synthesis in terms of concurrent monitors gives us scope for introducing optimisations in terms of reduced messages amongst (monitor) processes and reduced sub-monitor spawning. Approaches that are currently being explored are those relating to the shortening of chains of forwarders in the synthesised concurrent monitors and upfront pruning of redundant formula branches.

**References**