TECHNICAL REPORT
Report No. CS2012-03
Date: November 2012

Computer Science Annual Workshop
CSAW’12

Computer Science Department
## Programme and Table of Contents

### Day 1 - Thursday 8th November

<table>
<thead>
<tr>
<th>Time</th>
<th>Content</th>
<th>Speaker</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:00</td>
<td>Dealing with the Hypothetical in Contracts</td>
<td>Gordon Pace</td>
<td>3</td>
</tr>
<tr>
<td>09:45</td>
<td>Towards Proof-Theoretic Interpretations for LTL Monitoring</td>
<td>Clare Cini</td>
<td>7</td>
</tr>
<tr>
<td>10:15</td>
<td>Lightning Talk: Accountable Monitoring of Proof-Carrying Logs</td>
<td>Gordon Pace</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee: 10:30 - 11:00</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:00</td>
<td>Novel attack resilience by fusing events related to attack objectives</td>
<td>Mark Vella</td>
<td>8</td>
</tr>
<tr>
<td>11:45</td>
<td>Separation-Based Reasoning for deterministic channel-Passing Concurrent Programs</td>
<td>Aimee Borda</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch: 12:30 - 14:00</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:00</td>
<td>Making mutation testing a more feasible proposition for the industry</td>
<td>Mark Micallef</td>
<td>12</td>
</tr>
<tr>
<td>14:30</td>
<td>Towards Addressing the Equivalent Mutant Problem in Mutation Testing</td>
<td>Mark Anthony Cachia</td>
<td>15</td>
</tr>
<tr>
<td>15:00</td>
<td>Lightning Talk - Verification Techniques for Distributed Algorithms</td>
<td>Aimee Borda</td>
<td>n/a</td>
</tr>
<tr>
<td>15:15</td>
<td>Lightning Talk - Integrating Runtime Verification into the SDLC</td>
<td>Christian Colombo and Mark Micallef</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee: 15:30 - 16:00</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16:00</td>
<td><strong>mu-Larva Script: Rethinking the Larva scripting language</strong></td>
<td>Adrian Francalanza</td>
<td>17</td>
</tr>
<tr>
<td>16:30</td>
<td>Integrating Testing and Runtime Verification</td>
<td>Christian Colombo</td>
<td>19</td>
</tr>
</tbody>
</table>

### Day 2 - Friday 9th November

<table>
<thead>
<tr>
<th>Time</th>
<th>Content</th>
<th>Speaker</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:00</td>
<td><strong>Language extension proposals for Cloud-Based computing</strong></td>
<td>Adrian Francalanza</td>
<td>21</td>
</tr>
<tr>
<td>09:30</td>
<td><strong>Designing Correct Runtime-Monitors for Erlang</strong></td>
<td>Aldrin Seychel</td>
<td>23</td>
</tr>
<tr>
<td>10:15</td>
<td>Lightning Talk - Preserving determinism in actor based concurrency</td>
<td>Aldrin Seychel</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee: 10:30 - 11:00</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:00</td>
<td><strong>Formal Fault-Tolerance Proofs for Distributed Algorithms</strong></td>
<td>Mandy Zammit</td>
<td>24</td>
</tr>
<tr>
<td>11:45</td>
<td><strong>Real-time Selective Rendering</strong></td>
<td>Steven Galea</td>
<td>26</td>
</tr>
<tr>
<td>12:15</td>
<td>Lightning Talk Slot (free)</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Lunch: 12:30 - 14:00</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:00</td>
<td><strong>Distributed High-Fidelity Graphics using P2P</strong></td>
<td>Daniel D’Agostino</td>
<td>28</td>
</tr>
<tr>
<td>14:30</td>
<td><strong>High-fidelity rendering as a service using heterogeneous supercomputing systems</strong></td>
<td>Keith Bugeja</td>
<td>31</td>
</tr>
<tr>
<td>15:00</td>
<td><strong>Point Cloud Searching and Understanding</strong></td>
<td>Sandro Spina</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee: 15:30 - 16:00</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16:00</td>
<td>Epistemic and Deontic: The Two Tics</td>
<td>Gordon Pace</td>
<td>n/a</td>
</tr>
<tr>
<td>16:30</td>
<td>Wrap-Up</td>
<td>Gordon Pace</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Dealing with the Hypothetical in Contracts

Gordon J. Pace
Dept. of Computer Science, University of Malta, Malta
gordon.pace@um.edu.mt

The notion of a contract as an agreement regulating the behaviour of two (or more) parties has long been studied, with most work focusing on the interaction between the contract and the parties. This view limits the analysis of contracts as first-class entities — which can be studied independently of the parties they regulate. Deontic logic [1] has long sought to take a contract-centric view, but has been marred with problems arising from paradoxes and practical oddities [2]. Within the field of computer science, the holy grail of contracts is that of a deontic logic sufficiently expressive to enable reasoning about real-life contracts but sufficiently restricted to avoid paradoxes and to be computationally tractable.

Contract automata [3–5] have been proposed as a way of expressing the expected behaviour of interacting systems, encompassing the deontic notions of obligation, prohibition and permission. For instance, the contract automaton shown in Fig. 1 expresses the contract which states that ‘the client is permitted to initialise a service, after which, he or she is obliged to submit valid user credentials and the provider is prohibited from increasing the price of the service.’ Note that the states are tagged with the deontic information, explicitly stating what actions are obliged (\(O\)), permitted (\(P\)) and forbidden (\(F\)) by which party (given in the subscript). The transitions are tagged with actions which when taken by the parties induce a change of state, with * being used as shorthand to denote anything-else.

Fig. 1. Service provision contract

The use of automata to describe contracts allows for their composition in a straightforward manner using standard techniques from computer science. For instance, consider a bank client who has a savings account and a loan with the bank. The complete actual contract between the bank and its client is the com-
position of the two contracts regulating the different accounts. This approach thus not only enables the description, analysis and validation of contracts independently of the agents, but also allows for a compositional approach to contract construction.

A limitation inherent to the notion of contract automata is that they describe norms and ideal behaviour only through the knowledge of the agents’ actions. No part of a contract may depend on whether or not another part of the contract was violated or not. For example, clauses such as ‘if the bank does not give permission to the client to open a new bank account, it is obliged to pay a fine.’ Similarly, it is impossible to specify clauses which depend on whether another clause is also in force, such as ‘if p is obliged to pay, he is forbidden from leaving the shop.’ Note the difference between the two cases — whereas the first depends on whether or not a clause is satisfied, the second depends on whether a clause is enforced. We are currently looking into ways of extending the semantics of contract automata to allow for these two kinds of scenarios.

Reparation: Contracts identify ideal behaviour which is not necessarily adhered to — although the contract between the bank and the client states that the client is to ensure that the repayments are done on time, the client does not always do this, in which case another clause would ensure that he or she is now also obliged to pay a fine. These reparatory contract clauses are crucial to allow for the description of behaviour under non-ideal behaviour. This can be handled, for instance, by qualifying transitions also with deontic clauses (or their negation), ensuring that they can only be taken if the clause is satisfied (or violated). The initial state of the automaton shown in Fig. 2(a) shows how this construction is used to ensure that party 1 is permitted to download file A (\textit{downloadA}), as long as party 2 has permission to download file B (\textit{downloadB})\textsuperscript{1}.

Although in the example, the tagged clause appears in the source state of the transition, this is not necessarily the case and the approach can be used to reason about clauses as though they were in force in that state — hypothetical clauses. For instance, leaving out the \(\mathcal{P}_2(\textit{downloadB})\) in the initial state of the automaton in Fig. 2(a) would result in a similar contract but in which no violation is tagged when the transition to the next state is taken. Note that, for a complete semantics one would have to have two types of violations: (i) those violations which took place but have an associated reparation (violating \(\mathcal{P}_2(\textit{downloadB})\) in the initial state of Fig. 2(a)); and (ii) violations for which there is no associated reparation (violating \(\mathcal{P}_1(\textit{downloadA})\) in the same state). This allows us to make a distinction between satisfying the contract outright and satisfying but through the taking of reparations.

Conditional clauses: Another extension to contract automata are conditional clauses, based on whether a clause is in force at a particular moment in time e.g. ‘if Peter is obliged to pay, then Peter is forbidden from leaving the shop’.

\textsuperscript{1} For completeness we would also want to add which party is satisfying or violating the clause on the transition.
Dealing with the Hypothetical in Contracts

or ‘if Peter is not permitted to leave the shop then Peter is prohibited from smoking.’ Fig. 2(b) shows how the first of these examples can be encoded in a contract automaton. Such conditional clauses are particularly useful when composing contract automata, since the presence of particular clauses may not be known before the actual composition.

Fig. 2. (a) party 1 is permitted to download file A, as long as party 2 has permission to download file B; (b) if p is obliged to pay, then p is forbidden from leaving the shop.

The interaction of these two forms of deontic tagging can be used to express complex scenarios. For instance, the last example can be extended in such a manner that if Peter violates the prohibition from leaving (when in force), he would be forbidden from entering the shop again. Note that in such a case the transitions may also need to be tagged with conditional deontic clauses.

Furthermore, in these examples we have simply tested for the satisfaction/violation or presence/absence of a single contract clause. Extending this approach to deal with sets of clauses (in a conjunctive or disjunctive manner) or even for whole boolean expressions over these new operators can possibly lead to complex constructiveness issues similar to the ones which arise in Esterel [6]. Even without complex expressions, such cases may arise. For instance, how should the clause !O_p(a) ⊃ O_p(a) be interpreted? Does it mean that (i) the obligation to perform a should always be in force even when it was going not to be the case, or that (ii) it is a void statement which should be disallowed, since if the obligation is not in force, then we add it, but since it is now in force the clause enforcing it no longer triggers and hence cannot be considered to be in force?

These and other similar issues make the definition of the formal semantics of such contracts challenging and the development of algorithms for the discovery of conflicts in such contracts non-trivial.
References

Towards Proof-Theoretic Interpretations for LTL Monitoring

Clare Cini and Adrian Francalanza

Department of Computer Science, University of Malta
{ccin0005 | adrian.francalanza}@um.edu.mt

In this talk we will present a study on how runtime verification (RV) algorithms can be given one common proof-theoretic interpretation. In RV, there is a lot of work that deals with verification algorithms, yet there is no standard notation for such algorithms making it hard to understand and compare such work.

We aim to have a common underlying framework where such algorithms can be interpreted in a proof-theoretic way. Therefore we are going to investigate how verification algorithms, specifically the algorithms by Geilen [2], Sen et al. [3] and Bauer et al. [1], can be mapped to this kind of interpretation. In particular, we investigate whether these algorithms can be mapped to the coinductive interpretations for LTL. The coinductive interpretation appears to lend itself more naturally to the formalisation of runtime verification algorithms given the model over which LTL is defined i.e. infinite strings. Coinduction is often used for analysis that have an extensional flavour which seems in line with the notation of runtime monitoring (where you arrive to a conclusion by observing only part of the entity being analysed).

Preliminary results show that Geilen’s algorithm can be mapped to such proof systems, hence allowing us to integrate his runtime verification algorithm as a proof search over LTL rules. Furthermore, this algorithm is defined in a coinductive manner, thus confirming our initial idea, that coinduction is the ideal interpretation for verification. Yet, work is still ongoing and the results are not yet conclusive. If we are able to map every algorithm to our proof system, the latter can be seen as a framework for verification algorithms.

References

Novel attack resilience by fusing events related to objectives

Mark Vella
mark.vella@um.edu.mt

Research in intrusion detection systems (IDS) is mainly restricted to the misuse and anomaly detection dichotomy, and therefore to their limitations. Web attack detectors are a case in point, where ones that perform misuse detection are prone to miss novel attacks, whilst those performing anomaly detection produce impractical amounts of daily false alerts. Detectors inspired from the workings of the human immune system (HIS) have proposed new effective detection approaches, however without tackling the issue of novel attack resilience separately from anomaly detection.

Danger Theory-inspired detection. This paper attempts to leverage inspiration from the Danger Theory (DT), which is a recent model of the HIS [1, 2]. Specifically, the focus is on the process that enables the HIS to respond to previously unseen infections without attacking host cells or symbiotic organisms. At a high level of abstraction, the process consists of the sensing of generic signs of an ongoing infection which triggers a process that identifies the make-up of the germ responsible, resulting in a response that is highly specific. Seen from a computational point of view, immune responses as suggested by DT follow a generic-to-specific information fusion process. Generic signals of infection that germs cannot avoid producing, enable the immune system to respond to novel infections. Their fusion identifies the make-up of the responsible germ, thereby preventing the production of anti-bodies that do harm. The aim of this work is to explore how this process can be used to develop effective detectors, meaning that they are resilient to novel attacks and suppress false positives. This is achieved by translating the process from the HIS to the web attack domain, and then subsequently realized as concrete detectors.

Approach. There are two types of generic signals of ongoing infection: Pathogenic Associated Molecular Patterns (PAMP) and danger signals, that originate externally and internally to the human body respectively. PAMPs constitute molecular patterns associated with evolutionary distant organisms. PAMPs are not specific to germs, but are rather associated with entire germ classes and so any germ is expected to feature such patterns. Danger signals on the other hand are molecules that originally form part of the internals of the body’s own cells. Successful infections cause cells to explode, causing cell internals to be released and picked up by the HIS as danger signals. Like PAMPs, danger signals do not identify the germs causing them, and are expected to be provoked by any successful infection.
Figure 1 summarizes the proposed translation, with suspicious HTTP requests modeled on PAMPs and attack symptoms modeled on danger signals, both defined as attack objective-related events that attacks cannot avoid producing. The fusion process however does not follow the workings of the HIS since it was found to be counterproductive in earlier work [3]. Rather, it leverages feature-based correlation of events in order to identify attack HTTP requests. Since suspicious HTTP requests and attack symptoms are not exclusive to attack behavior, this component is required to distinguish those related to attacks. This can be achieved through feature similarity links that capture both the causal relation between suspects and symptoms as well as their association with an ongoing attack, in a similar manner to existing security event correlation systems.

Results and conclusions. The approach has been evaluated through three detectors that cover both high impact and popular web attacks within their scope. Effectiveness results are encouraging, with novel attack resilience demonstrated in terms of attacks aiming for a specific objective but modify the exploited vulnerability, payload or use obfuscation. False positive suppression is achieved until requests that do not contain the same attack content as per coincident and successful attack requests. However, their implementation is rendered difficult by requiring extensive knowledge of the deployment platform and secure coding. A performance study showed that the efficiency challenges tied with the stateful detection of events can be mitigated. These results merit a follow-up through further detector development and a formalization of the method.
References

Separation-Based Reasoning for Deterministic Channel-Passing Concurrent Programs

Aimee Borda and Adrian Francalanza
University of Malta

Tractable proof systems for concurrent processes are hard to achieve because of the non-deterministic interleaving of statements. This is particularly true when the scheduling of statements have an effect on the final state of the program, referred to as racy conditions. As a result, when we try to reason about concurrent programs, we have to consider all possible thread interleaving to accurately approximate the final result. Since, scalability is important we limit our reasoning to a subset of the concurrent programs that are race-free. By doing so we gain serialized reasoning, where examining one possible path will automatically encapsulate the other interleaving.

Fundamentally, races are caused when two concurrent processes try to access shared resources simultaneously (for example two concurrent processes trying to alter a location in memory at the same time) and hence the final result depends on the last statement that updates the location.

A popular approach for delimiting the class of race-free processes is through resource separation. [3] Through the resources assigned to each process we can approximate the interference that a process can create to its concurrent processes. Therefore, if the resources available are split disjointly among the concurrent processes, we guarantee that no interference between the processes is created. More concretely, if all concurrent processes work within different parts of the memory, each process will not create race conditions for other processes. Moreover, this guarantee allows us to reason about each process independently, as a sequential process.

Nonetheless, the formalism must also handle dynamic resource separation and resource sharing, where a subset of the resources’ access rights are shared amongst the processes. A popular example is the reader-writer problem. Here, race-conditions are created only, if one of the concurrent processes modifies the content of a shared location, since trying to read that location will depend on whether the statement is scheduled before or after the write has been committed. Hence, resource sharing can be allowed if the actions performed by each process on that resource do not create races to values, which in this example imply that we have concurrent reads but exclusive writes.

Separation logic is a possible formalism which handles this quite elegantly. [3,2] In [1], a proof system inspired from separation logic to reason about concurrent processes for the message passing model is described. However, it can only reason in terms of the complete disjointness of resources. In this work, we are trying to push the boundaries of non-racy processes by examining some form of resource sharing and how these structures preserve determinism.

References
Making mutation testing a more feasible proposition for the industry

Mark Micallef, Mark Anthony Cachia

Department of Computer Science, University of Malta, Malta
mark.micallef@um.edu.mt, mcac0019@um.edu.mt

Software engineering firms find themselves developing systems for customers whose need to compete often leads to situations whereby requirements are vague and/or prone to change. One of the prevalent ways with which the industry deals with this situation is through the adoption of so-called Agile development processes. Such processes enable the evolutionary delivery of software systems in small increments, frequent customer feedback, and, ultimately, software which continuously adapts to changing requirements. In this fluid scenario, developers rely on automated unit tests to gain confidence that any regressions resulting from code changes will be detected. Consequently, trust in the software system can only follow from the quality of the tests. Unfortunately, the industry tends to rely on tools that calculate primitive measures such as statement coverage, a measure which has been shown to provide a false sense of security [2].

Mutation testing [3] is an analysis technique based on the following idea: Given a program $P$ and a test suite $T$, if one judges $T$ to adequately cover $P$, then executing $T$ against $P'$ (where $P'$ is an altered version of $P$), should result in at least one failing test. Therefore, from an original program $P$, a number of modified programs $P_1\ldots P_n$, called mutants, are produced by applying a number of syntactic modifications to $P$. These modifications are carried out by mutation operators which are designed to change $P$ in a way that corresponds to a fault which could be introduced by a developer. Mutants which are detected by $T$ are said to be killed. Undetected (unkilled) mutants require manual investigation by developers, possibly resulting in improvements to $T$. In comparison to techniques such as statement coverage analysis, mutation testing provides a significantly more reliable measure of test suite thoroughness.

Despite its effectiveness, mutation testing suffers from three recognised problems. These are (1) the computational expense of generating mutants\(^1\), (2) the time required to execute test suites against all mutants\(^2\), and (3) the equivalent mutant problem. The latter refers to situations where syntactically different mutants turn out to be semantically identical, thus wasting time and effort. Besides the three cited problems with mutation testing, we also argue that there

---

\(^1\) The computational complexity of Mutation Testing is $O(n^2)$ where $n$ is the number of operations in the source code [1].

\(^2\) Executing all tests against each mutant renders mutation tools unacceptably slow and only suitable for testing relatively small programs[4].
is a fourth problem, one concerned with the time and effort required to investigate and address unkilled mutants. Each unkilled mutant requires a developer to understand the mutant’s semantics, determine if a change to the test suite is required and finally modify the test suite to kill the mutant. We argue that this effort can be a deterrent to the wider uptake of mutation testing because the time and cognitive effort required to carry out the task may not be perceived as being worth the potential benefits gained.

In this talk, we will provide an overview of mutation testing and subsequently discuss problems which prevent its wider uptake. We then discuss our research activities in this area and present a technique which we term as *localised mutation*. The technique leverages the iterative nature of Agile development such that one only creates mutants from sections of the codebase which have changed since the last mutation run. The hypothesis is that if mutation testing is carried out in bite-sized chunks on code which has recently changed, then computational expense can be drastically reduced and developers should experience less cognitive load during analysis. Consequently, the main hurdles of mutation testing adoption in industry would be significantly reduced. Preliminary results from this research will also be discussed and related to our ongoing research activities.

References


Integrating Mutation Testing into Agile Processes through Equivalent Mutant Reduction via Differential Symbolic Execution

Mark Anthony Cachia
Department of Computer Science
University of Malta
Msida, Malta
mcac0019@um.edu.mt

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

I. Agile Programming

In agile programming, software development is performed in iterations. To ensure the changes are correct, considerable effort is spent writing comprehensive unit tests [2]. Unit tests are the most basic form of testing and is performed on the smallest or smaller set of code [7]. These unit tests have multiple purposes, the main one being that of acting as a safety net between product releases. However, the value of testing can be called into question if there is no measure of the quality of unit tests [2]. Code coverage analysis is an automated technique which illustrates which statements are covered by tests [4]. However, high code coverage might still not be good enough as whole branches or paths could still go completely untested which in turn leads to false sense of security [8]. Mutation Testing is a technique designed to successfully and realistically identify whether a test suite is satisfactory. In turn, such tests lead to finding bugs within the code. The technique behind mutation testing involves generating variants of a system by modifying operators (called mutants) and executing tests against them. If the test suite is thorough enough, at least one test should fail against every mutant thus rendering that mutant killed. Unkilled mutants would require investigation and potential modification of the test suite [3].

II. Mutation Analysis

Mutation analysis is a process which determines the effectiveness of a test suite. This is achieved by modifying the source of a program and ensuring that at least one test fails thus ensuring the tests are sensitive to particular source changes. A mutant who is detected by tests is called a killed mutant. In mutation analysis, a large number of mutants are generated. Mutants are programs which have been syntactically altered. Sometimes, such alterations lead to the generation of Equivalent Mutants. Equivalent mutants is a problem in Mutation Testing and can be defined as the mutants “which produce the same output as the original program” [3]. Performing Mutation Analysis on Equivalent mutants is a waste of computation time. As program equivalence is undecidable, automatically detecting equivalent mutants is impossible. The equivalent mutant problem is a barrier that prevents Mutation Testing from being widely adopted. The result of Mutation Analysis is a ratio or percentage of the killed mutations divided by the sum of equivalent mutants [3]; Figure 1 illustrates.

There are various reasons why mutants may be equivalent. Grun et al. [1] manually investigated eight equivalent mutants generated from the JAXEN XPATH query engine program. They noticed four main reasons which cause a mutant to be equivalent are

- mutants generated from unneeded code,
- mutants which improves speed,
- mutants which just alter the internal states, and
- mutants which cannot be triggered.

III. Symbolic Execution

Symbolic execution is the process of analysing a program by executing it in terms of a symbolic parameter α instead of a concrete instance. The class of inputs represented by each symbolic execution is determined by the control flow of the program based on branching and operations on the inputs. Each branch leads to a unique path condition (PC). A PC is a set of conditions which the concrete variables have to adhere to for the execution to be in the given particular path; hence, for any given set of concrete parameters, the given parameters can reside into at most one path. Initially, the path condition is true, however at each branch operation, the branch condition is ANDed to the previous PC. In the else path, the NOT of the branch condition is added to the current PC [5].

Symbolic execution takes normal execution semantics for granted and is thus considered to be a natural extension of concrete execution. At the end of symbolic execution, an execution tree can be graphed illustrating all the possible paths
the program can follow. At each leaf, there exists a unique PC which can be satisfied by a concrete input. The PC at any of the leaves is distinct i.e. ¬(PC₁ ∧ PCₙ). As symbolic execution satisfies a commutative property (relative to concrete examples, symbolic execution has the “exact same effects as conventional executions” [5].

A. Differential Symbolic Execution

Differential symbolic execution is a technique introduced by Person et al. [6] which efficiently performs Symbolic execution on two versions of the same class. The aim of Differential symbolic execution is effectively and efficiently determine whether the two version of the code is equivalent; and if not, characterise the behavioural differences by identifying the sets of inputs causing different behaviour [6].

Differential symbolic executions works by symbolically executing each method version to generate symbolic summaries. Symbolic summaries pair input values with the constraint (also known as the branch condition) together with the operation on a symbolic value (the effect or operation). The summaries from both versions are compared. If the summaries are equivalent, the methods are considered functionally equivalent. Otherwise, a behavioural delta (Δ) which characterise the input values where the versions differ.

IV. PROPOSED WORK AND EVALUATION

The proposed work is to combine Differential symbolic execution to the most effective optimisations of Mutation Testing. Direct bytecode manipulation as well as Localised Mutation; a technique which only mutates modified code when compared to previous versions as introduced in the FYP, will be employed to ensure the utmost performance in the generation of mutants. Differential symbolic execution, will be used to perform efficient functional equivalence between the mutated method and the original method to ensure the method’s semantics have been altered. This determines if the versions have the same black box behaviour as well as have the same partition effects [6].

The evaluation under consideration will assess the effectiveness of equivalent mutant reduction in Mutation analysis. The evaluation will be performed in two iterations with two different algorithms:

1) Mutation testing with the most efficient optimisations
2) Mutations testing with the most efficient optimisations as well as equivalent mutant reduction using Differential symbolic execution.

Various metrics are envisaged to be utilised. Such metrics currently include the execution time to perform the complete Mutation analysis and the time taken for the developer to implement enough tests until a satisfactory Mutation score is achieved. Another aim of this work is to find the most efficient point in the software life cycle at which it is ideal to start performing mutation analysis. Particularly, if it’s worth performing in just one chunk at the very end of the cycle or if it is most feasible to perform incremental Mutation testing during the whole development process. It is planned that the interval at which Mutation analysis is ideal to commence and at what interval the analysis should be performed (such as per commit or by some other heuristic) will be also determined during the empirical evaluation.

REFERENCES


μLarvaScript: Rethinking the Larva Scripting Language

Adrian Francalanza
adrian.francalanza@um.edu.mt
University of Malta

1 Synopsis

polyLarva, the latest incarnation of the Larva runtime-verification (RV) tool suite, experienced a major redesign to its scripting language (used for specifying the monitors that carry out the RV.) As opposed to previous versions, where the programmer specified the monitoring automata describing the RV procedure, in pLarvaScript (the scripting language for polyLarva) the programmer specifies monitoring as a sequence of (guarded) rules of the form

\[ p, c \rightarrow a \]

where \( p \) denotes an event pattern, \( c \) denotes a condition and \( a \) stands for a monitor action. A monitor synthesised from this sequence of rules would then listen to a stream of events: for each event \( e \), it attempts to match an event pattern of the list of rules; when this happens, the monitor evaluates the corresponding condition of the rule and, if successful, the respective rule action is triggered.

By and large, the new version of LarvaScript has so far been developed organically, motivated more by the pragmatic concerns of the applications considered rather than by language design issues. Also, because of backward-compatibility concerns, a number of design decisions were inherited, carried over from its precursors. Although effective and pragmatic, this method of development has hampered a full understanding of the resulting language: at present, the only formal semantics available is the polyLarva compiler itself which is not ideal for a number of reasons (i) an understanding of the language constructs requires a detailed understanding of the compiler; (ii) we have no way how to determine whether the compiler implementation is correct; and (iii) concurrency often introduces subtle semantic and language design issues that are best studied at a higher level of abstraction.

In this talk, I will discuss preliminary investigations regarding the analysis of the pLarvaScript language from a foundational perspective. For instance, we consider inherited constructs such as foreach for defining sub-monitors together with associated design decisions, such as the hierarchic organisation of these submonitors, and reassess them again from first principles. We do not not have any conclusive results, and thus far our guiding principles have been simplicity, elegance (admittedly both subjective measures) and a potential concurrent implementation of the tool. I will therefore be seeking feedback from the audience.
during the talk, which should help us hone our present positions regarding the understanding of the language.
Combining Testing and Runtime Verification

Christian Colombo
Department of Computer Science, University of Malta

Testing and runtime verification are intimately related: runtime verification enables testing of systems beyond their deployment by monitoring them under normal use while testing is not only concerned with monitoring the behaviour of systems but also generating test cases which are able sufficiently cover their behaviour. Given this link between testing and runtime verification, one is surprised to find that in the literature the two have not been well studied in each other’s context. Below we outline three ways in which this can be done: one where testing can be used to support runtime verification, another where the two techniques can be used together in a single tool, and a third approach where runtime verification can be used to support testing.

1 A Testing Framework for Runtime Verification Tools

Like any piece of software, runtime verification tools which generate monitoring code from formal specification have to be adequately tested, particularly so because of its use to assure other software. State-of-the-art runtime verification tools such as Java-MOP [12] and tracematches [3] have been tested on the DaCapo benchmark [2]. However, the kind of properties which have been monitored are rather low level contrasting with our experience with industrial partners who seem more interested in checking for higher level properties (such as the ones presented in [5, 4]). Whilst we had the chance to test our tool Larva [6] on industrial case studies, such case studies are usually available for small periods of time and in limited ways due to privacy concerns. Relying solely on such case studies can be detrimental for the development of new systems which need substantial testing and analysis before being of any use.

For this reason, we aim to develop a testing framework which would provide a highly configurable mock transaction system to enable thorough validation of systems which interact with it. Although not a replacement of industrial case studies, this would enable better scientific evaluation of runtime verification systems.

2 Combining Testing and Runtime Verification Tools

While testing is still the prevailing approach to ensure software correctness, the use of runtime verification [1] as a form of post-deployment testing is on the rise. Such continuous testing ensures that if bugs occur, they don’t go unnoticed. Apart from being complementary, testing and runtime verification have a lot in common: runtime verification of programs requires a formal specification of requirements against which the runs of the program can be verified [11]. Similarly, in model-based testing, checks are written such that on each (model-triggered) execution step, the system state is checked for correctness. Due to this similarity, applying both testing and runtime verification
techniques is frequently perceived as duplicating work. Attempts [9] have already been 
made to integrate the runtime verification tool Larva [6] with QuickCheck [10]. We 
plan to continue on this work by integrating the Larva tool with a Java model-based 
testing tool, ModelJUnit\textsuperscript{1}.

\section{Using Runtime Verification for Model-Based Testing Feedback}

To automate test case generation and ensure that the tests cover all salient aspects of a 
system, model-based testing [7, 8] enables the use of a model specification from which 
test cases are automatically generated. Although successful and growing in popularity, 
model-based testing is only effective in as much as the model is complete. Sequences 
of untested user interaction may lead to huge losses for the service-provider if any of 
these lead to a failure. Since coming up with a model for test case generation is largely 
a manual process [7, 8], it is virtually impossible to ensure the model is complete.

In this context, we propose to use runtime information to detect incompleteness in 
the test case generation model: by considering the execution paths the system takes at 
runtime, a monitor checks whether each path (or a sufficiently similar one) is in the test 
case generation model.

\begin{thebibliography}{9}
\bibitem{11} Leucker, M., Schallhart, C.: A brief account of runtime verification. JLAP 78(5), 293–303 (2009)
\end{thebibliography}
\textsuperscript{1} http://www.cs.waikato.ac.nz/ marku/mbt/modeljunit/
Language Extension Proposals for Cloud-Based Computing

Adrian Francalanza
adrian.francalanza@um.edu.mt, and Tyron Zerafa
tzer0001@um.edu.mt
University of Malta

1 Synopsis

Cloud computing can be described as the homogenisation of resources distributed across computing nodes, so as to facilitate their sharing by a number of programs. In some sense, the use of virtual machines in languages such as Java and Erlang, are a first step towards this idea of cloud computing, by providing a common layer of abstraction over nodes with different characteristic. This has fostered various forms of distributed computing mechanisms such as web services based on remote procedure calls (RPCs) and code on demand (COD) applets executing in sandboxed environment.

Erlang is an actor-based programming language that lends itself well to the construction of distributed programming through a combination of language features such as message-passing and error-handling mechanisms. It also offers mechanisms for dynamically spawning processes (actors, to be more precise) on remote nodes, for reasons ranging from load balancing to the maximisation of computation proximity vis-a-vis the resources that it uses. Although the mechanism work well, it relies on a rather strong assumption when programming for the cloud, namely that the source-code at every node is homogeneous.

The first aim of our study is to create a layer of abstraction that automates the necessary migration of source-code so as to allow seamless spawning of processes across nodes “on the cloud”. The challenge here is to migrate the least amount of code, at the least amount of cost/effort, so as to allow the remote computation to be carried out. The solutions we explore range from those that rely on code dependency static analyses to “lazy” dynamic methods that migrate source code only when needed by the host node. There are also issues relating to source-name clashes and versioning.

The second aim of the study is to enhance the security of resources advertised on the cloud, by delineating their expected use. Our approach will rely on the mechanism of having a policy file per node, describing the restrictions imposed on resource usage. We plan to explore feasibility of code migration mechanisms that take into consideration these policy restrictions imposed by the receiving nodes. We shall also explore various ways how to enforce these policies.

The third aim of the study is to analyse mechanisms for tolerating network errors such as node disconnections; this is particularly relevant to distributed
computations spanning over more than two nodes. Again, we plan to use the policy-file approach to automate decisions that need to be taken to seamlessly carry out distributed computation in the eventuality of such failures; the policy files may also be used to specify redundancy mechanisms that should, in turn, enable better tolerance to such faults.
In runtime verification, a monitor continuously checks the execution of a program that is running concurrently with it. Normally, the runtime monitor checks that the system does not violate a correctness property. Any runtime monitor is expected to satisfy the following:

*If a system does not obey a property \( \phi \), then the monitor for \( \phi \) MUST flag a failure.*

Showing that this statement holds is not trivial since we have to consider all possible execution paths of the monitor. In our work, we show how one can prove this indirectly by proving a number of correctness criteria on our synthesised monitors.

For instance, a desirable property for monitors is that the execution of a monitor does not result in divergence since this would result in infinite computation by the monitor. Having divergence in the monitor results in an unacceptable computational overhead that would hinder the performance of the monitored system. If the computational overhead incurred on the system is unacceptably high, the monitor cannot even be considered as a valid monitor since it will never be used.

Another desirable property that monitors should observe is that there exists at least one execution path that when the program does not satisfy a property, the monitor will give a *fail* notification. This is generally enough if the monitors are executed in a sequential manner.

The monitors might have to execute using concurrent processes for example to check multiple branches of a property at the same time. Thus, in a concurrent environment it is not enough that there exists a single path that gives the expected result. Having a proof of determinism would extend the latter property to be valid for all execution paths. Determinism also means that if a specific execution trace is monitored multiple times, the monitor will always give the same result.

In this talk, a design of monitors in Erlang for Erlang programs is discussed. A proof that this design of monitors satisfies the theorem stated above is also given. The design uses concurrent processes in order to monitor multiple sub-properties combined under conjunction. These synthesised monitors are shown to be non-divergent, are correct and are determinate. In order to show that the monitors preserves determinism, the monitors are proved to be confluent since confluence implies determinism.
Formal Fault-Tolerance Proofs for Distributed Algorithms

Mandy Zammit Adrian Francalanza
Department of Computer Science
Faculty of ICT
University of Malta
mzam0041@um.edu.mt adrian.francalanza@um.edu.mt

Distributed Algorithms express problems as concurrent failing processes which cooperate and interact towards a common goal [1, 2]. Such algorithms arise in a wide range of applications, including distributed information processing, banking systems and airline reservation systems amongst others. It is desirable that distributed algorithms are well behaved both in a failure free environment and even in the presence of failure (i.e. fault tolerant). To ensure well behavedness for all executions of distributed algorithms formal correctness proofs are needed. This is due to the concurrent nature of such algorithms, where executions of the algorithms result in different interleavings amongst parallel processes (i.e. there is a large number of possible execution paths).

However distributed algorithms are often described in semi-formal pseudo code and their correctness criteria in natural language. The current status quo has a number of shortcomings. First is, the discrepancy between the semi-formal description of the algorithm and the actual implementation which is expressed in a formal language. This discrepancy is often a principal source of errors in the deployment of these algorithms. Second, due to the exponential interleaving amongst parallel processes and the subtle forms of process interference at these interleavings, becomes rather subtle and delicate.

Unfortunately formalisation tends to increase complexity because formal analysis forces you to consider every possible interleaving. At the same time the use of proof mechanizations still does not yield a clearly expressed proof that is easy to construct or understand; rather, it yields monolithic mechanical proofs that are very hard to digest.

In our work we address the complexity problem introduced by formalisation. We formalise and prove the correctness of three Broadcast Protocols, namely; Best Effort Broadcast, Regular Reliable Broadcast, and Uniform Reliable Broadcast. We use an asynchronous Pi-calculus together with fail-stop processes and perfect failure detectors to encode the protocols and their correctness criteria, and bisimulation (≈, a notion of coinductive equivalence) for the correctness proofs. In order to alleviate complexity, we extend the work conducted by Francalanza et. al. [3] where we make use of the concept of Harnesses, and a novel approach to correctness proofs whereby proofs for basic correctness (correctness in a failure free setting) and for fault tolerance (correctness in a setting were failures can be induced) are disjoint. We extend their methodology to produce inductive proofs containing coinductive techniques at each inductive step rather than having one large coinductive proof. This would allow us to move away from the large monolithic proofs.

References


1 correctness criteria specify the well behavedness of an algorithm


Abstract - Traditional physically-based renderers can produce highly realistic imagery; however, suffer from lengthy execution times, which make them impractical for use in interactive applications. Selective rendering exploits limitations in the human visual system to render images that are perceptually similar to high-fidelity renderings in a fraction of the time. This paper outlines current research being carried out by the author to tackle this problem, using a combination of ray-tracing acceleration techniques, GPU-based processing, and selective rendering methods. The research will also seek to confirm results published in literature, which indicate that users fail to notice any quality degradation between high-fidelity imagery and a corresponding selective rendering.

1 Introduction

At its essence, computer graphics is concerned with synthesizing a pictorial representation of objects and environments from their computer based models [1], through a process known as rendering. Traditional rendering methods make use of rasterization to project 3D models to a 2D image plane, together with an array of specialized methods to approximate real-world phenomena like shadows, reflections and indirect lighting [2]. These techniques have evolved into mature APIs (e.g. OpenGL, DirectX) and can achieve real-time performance at the cost of limited realism and increased application complexity.

In contrast, physically-based renderers seek to replicate the actual properties of lighting and material. For instance, ray-tracing simulates the path taken by light through a virtual environment, and can generate images that not only look realistic, but are also physically plausible. However, rendering in this manner is very computationally intensive, and until recently, physically-based rendering was largely limited to off-line rendering [8].

This paper outlines the authors research in selective rendering with the aim of generating high-fidelity imagery at interactive rates and confirming experimental results published in literature. Section 2 gives a short description of the problem to be tackled. Section 3 reviews the relevant literature. The proposed solution is discussed in Section 4. Section 5 provides an overview of the current project as well as possible directions for further research. Finally, Section 6 concludes the paper.

2 Problem Description

According to Chalmers and Ferko [4], rendering high-fidelity images in real-time has long been the "holy grail" of the computer graphics community. To achieve this, research on physically-based renderers has tried to tackle the problem from different points of view. On one front, researchers have focused on finding effective data structures that reduce the number of operations carried out for each ray of light [3]. Other works have focused on implementing massively-parallel ray tracers on GPUs [5], and distributing rendering load over a computing cluster [6] or supercomputer [7]. However, simulating real-world light interaction of complex scenes at interactive rates on a desktop PC still remains out of reach, even using modern GPUs [8].

More recent attempts have adopted principles from psychology and perception to target computation to parts of the scene that matter most to a human observer [9]. Selective rendering allows for substantial reductions in computation while creating an image that is perceptually equivalent to rendering a high-fidelity image of the whole scene.

3 Literature Review

Physically-based rendering algorithms, such as ray-tracing, generate high-fidelity images by simulating the physics of light and its interaction with surface materials. In particular, global illumination effects, such as surface inter-reflections and caustics, play a major role in enhancing the realism of rendered images [10]. However, the huge amount of computation required to produce physically-correct images prevents these methods from being utilized in interactive applications on desktop computers [4]. Several techniques have been developed to improve ray tracing performance:

Acceleration data structures, such as kd-trees and BVH trees, reduce geometric complexity by testing for ray-object intersections against groups of geometric primitives [11].

Instant global illumination (IGI) pre computes the distribution of lighting in the scene by following light paths from light sources in the scene and depositing virtual point lights (VPL) at each vertex [12]. Rendering indirect light is achieved by casting shadow rays from the shaded point to each VPL to determine visibility.

Interleaved sampling exploits the slowly changing nature of the direct lighting function to include the contribution from a given number of VPLs in less time [13]. This is achieved by modifying the IGI pre-processing stage to generate k subsets of VPLs. The image is then partitioned into tiles of n by m (n*m=k, usually n=m=3 or m=n=5) pixels, with each pixel sampling indirect light from the relative VPL subset. Finally, a discontinuity buffer is used to calculate the contribution of all VPLs at each pixel by interpolating values from neighboring pixels, provided that they do not cross geometric discontinuities [12].

Selective rendering seeks to produce images that are perceived to be of an equivalent quality to a physically-correct image, but using a fraction of the computation. This is possible since the human visual system does not process an image in raster-like fashion, but rapidly directs attention to salient features in a scene, such as sudden movement or brightly coloured objects [14]. This knowledge can be used to construct saliency maps, i.e. images highlighting the most perceptually interesting parts of the scene, which are then used to direct computation appropriately.

4 Proposed Solution

Saliency maps generated using image processing techniques are usually expensive to compute and do not give the region of interest where the user is currently looking at, but only where the user is likely to direct attention to. This MSc will look into various methods of achieving real-time performance for rendering using selective methods. This includes the possibility of using an eye tracker to get an accurate measurement of the user's current region of interest, with all the associated computation being carried out by the eye tracking device. This should significantly reduce the time spent on computing the saliency map, making more resources available for rendering. To our knowledge, the use of an eye tracking device for real-time selective rendering has never been explored before.

Additionally, this project will also seek to confirm results published in literature which indicate that users fail to notice any differences in quality between a high-fidelity rendering and a selective rendering of the same scene. Previous psychophysical experiments have used pre-rendered images and animations for the evaluation. If the renderer is successful at achieving real-time performance, the project can also look into carrying out a psychophysical experiment with interactive imagery, where the participants are able to freely navigate the scene.
5 Project Status

A prototype to demonstrate proof of concept is being developed as an extension to the GPU-based ray-tracing framework NVIDIA OptiX [15]. The framework provides a high-performance ray-tracing core comprising a small set of building blocks that are used in most ray-tracing algorithms.

Several basic components such as model-loading and rendering of direct lighting have already been developed. Rendering of global illumination is achieved by a GPU-based implementation of the IGI algorithm, together with interleaved sampling and discontinuity buffer to improve execution time.

Interleaved sampling provides good speedups at the cost of some degradation in image quality, especially near geometric discontinuities, where the averaging across nearby samples cannot be performed. The loss in image quality arises mainly from the fixed 3x3 or 5x5 tiling in interleaved sampling. Current research is investigating whether it is possible to obtain a better quality image by smartly positioning the samples across the image according to saliency. This is done by considering the full set of VPLs at each sample, but positioning the samples densely in salient regions (e.g. at pixel intervals), and more sparsely (e.g. sampling once per 3x3 or 5x5 tile) in less salient regions. Missing illumination values can then be interpolated from nearby samples.

Preliminary results show that this technique gives a good quality indirect lighting image. However, a more in-depth analysis still needs to be carried out to compare the output of the proposed method with that of traditional interleaved sampling.

The saliency map currently being used is a GPU implementation of the method by Hou et al. [16]. Although fast, the method is quite basic and several enhancements have been proposed in literature to improve accuracy. Future work will look into substituting the current saliency map with an improved version, thus improving the quality of the indirect lighting image.

Once the selective rendering component is ready, the project may also look into using an eye tracker to provide the saliency information instead of the saliency map.

6 Conclusion

This paper started with a review of the existing literature, outlining the performance issues of rendering high-fidelity imagery at interactive rates. An overview of the current research was given, proposing a combination of ray-tracing acceleration techniques, GPU-based processing, and selective rendering methods as a way to tackle this problem. The current status of the project was discussed, together with preliminary results and directions for future research.

References

Distributed High-Fidelity Graphics using P2P

Daniel D’Agostino

In the field of three-dimensional computer graphics, rendering refers to the process of generating images of a scene from a particular viewpoint. There are many different ways to do this, from the highly interactive real-time rendering methods [1] to the more photorealistic and computationally intensive methods [5].

This work is concerned with Physically Based Rendering (PBR), a class of rendering algorithms capable of achieving a very high level of realism. This is achievable thanks to physically accurate modelling of the way light interacts with objects in a scene [7], together with the use of accurately modelled materials and physical quantities.

Unfortunately, this realism comes at a cost. PBR computations are very expensive, and it may take several hours to render a single image. Hence, it is no surprise that most of the research in this field attempts to find ways to reduce that cost, and produce images in less time.

In spite of their diversity, all physically-based rendering algorithms attempt to solve the same problem: imitating the visual appearance of an environment as it would look in real life. This problem is formulated as the rendering equation [6]. Based on the principle of conservation of energy, this equation states that the light leaving a surface comprises both the light emitted from the surface, and that being reflected from other surfaces.

Due to factors relating to geometric complexity of the scene, it is not possible to solve the rendering equation analytically [7]. This difficulty mandates the use of numerical techniques, the most popular of which are Monte Carlo techniques. These were introduced to PBR with distributed ray tracing [3], a technique based on the ray tracing method [11] which could collect light arriving at a surface from many different directions in order to include fuzzy phenomena (such as soft shadows).

Irradiance caching [10] is a particularly useful technique that can be used in conjunction with distributed ray tracing (as well as other algorithms). Based on the observation that indirect diffuse lighting varies slowly over a surface [10], it stores irradiance (view-independent lighting data) in a data structure (usually an octree) and uses fast interpolation in order to speed up computation.

While one side of PBR research has been formulating more efficient algorithms to render physically-based images faster, another has been exploiting the embarrassingly parallel nature of ray tracing [4] in order to distribute the rendering load on many processors or interconnected machines [9].

Research into parallel PBR has so far almost exclusively used the client/server model. Although some rendering research based on the peer-to-peer (P2P) model has recently emerged [8, 2, 12], at present (to the best of our knowledge) there is none that is designed to improve physically-based rendering systems.
The method we are developing is based on the irradiance caching algorithm. Given that irradiance stored by a renderer is view-independent, it is easy to share with other machines, who can then use it directly without having to recompute it locally. This makes it particularly suitable for use over a P2P network.

Aside from the obvious benefits of free computation resulting in overall speedup, this novel way of sharing irradiance between peers is expected to provide new and interesting scenarios in which PBR can be used for collaboration between several peers in the same scene, as opposed to the traditional client/server model where the clients would do the rendering work and the server would be the only one to see the final images.

References

Fig. 1. The Sibenik Cathedral, rendered using our Irradiance Cache

Fig. 2. The Kalabsha Temple, rendered using our Irradiance Cache
Rendering as a Service

Keith Bugeja
Department of Computer Science
University of Malta

High-fidelity rendering requires a substantial amount of computational resources to accurately simulate lighting in virtual environments [1]. While desktop computing, boosted by modern graphics hardware has shown promise in delivering realistic rendering at interactive rates, rendering moderately complex scenes may still elude single machine systems [2]. Moreover, with the increasing adoption of mobile devices, which are incapable of achieving the same computational performance, there is certainly a need for access to further computational resources that would be able to guarantee a certain level of quality.

Cloud computing is a distributed computing paradigm for service hosting and delivery. It is similar to grid computing, but leverages virtualisation at multiple levels to realise resource sharing and provision [3], providing resources on demand and charging customers on a per-use basis rather than at a flat rate. Cloud technologies have the potential of providing a large number of resources to dedicate to a single application at any given point in time [5]. This paradigm offers the possibility of interactive high-fidelity graphics for users that do not have access to expensive dedicated clusters and can be delivered on any system from mobile to tablet to a desktop machine.

Fig. 1. Mnajdra site, generated using our high-fidelity renderer, Illumina PRT
The aim of our research is that of leveraging the cloud infrastructure to provide high-fidelity rendering and thus present a first attempt into rendering as a service, via a system that can provide parallel resources for rendering which could be either dedicated or adapt to the servers’ workload. As such, there exist challenges across three broad aspects: efficient high-fidelity rendering, resource allocation and task distribution, and low-latency client-server communication.

The focus of this presentation is the task distribution framework for cluster computing [4] that is currently being researched and developed; whereas grid and cluster infrastructures favour job submissions for medium to long term executions, our task distribution framework is targeted at short term execution and interactivity, in the vein of Many-task Computing [6]. We also consider future directions that our research might take, specifically vis-a-vis generalisation and heterogeneous computing.

References

Iterative Partitioning and Labelling of Point Cloud Data

Sandro Spina
Department of Computer Science
University of Malta

Over the past few years the acquisition of 3D point information representing the structure of real-world objects has become common practice in many areas[4]. This acquisition process[1] has traditionally been carried out using 3D scanning devices based on laser or structured light techniques. Professional grade 3D scanners are nowadays capable of producing highly accurate data at sampling rates of approximately a million points per second. Moreover the popularisation of algorithms and tools capable of generating relatively accurate virtual representations of real-world scenes from photographs without the need of expensive and specialised hardware has led to an increase in the amount and availability of 3D point cloud data. The management and processing of these huge volumes of scanned data is quickly becoming a problem[2].

Fig. 1. Section of Mnajdra pre-historic temple scanned by Heritage Malta

Following the on-site acquisition process, substantial data processing is usually carried out. This scenario is contributing to an increase in importance for algorithms which are capable of analysing and processing point clouds efficiently. In general this post processing of data takes a set of points as input (the point cloud) and returns a set partition of the input. For example, point clouds acquired from external sites would usually require a cleaning process in order to remove points which are not relevant to the site. The input would thus consist of all points acquired and the output a set partition consisting of two sets; the
first storing those points which are relevant and should be kept and the second storing those which should be discarded. In many areas, another important post processing task is that of grouping together points which logically fall under the same semantic definition. For example all points making up the floor of a site, or the windows, the benches, doors, etc.

![Fig. 2. Recognition and reconstruction of scanned 3D indoor scene [3]](image)

In a similar fashion, techniques have been developed which address the problem of reconstructing scanned indoor scenes into their respective components. Figure 1 illustrates one such example. In these cases the challenges are mainly related to clutter, resulting in both object interferences and occlusions. This presentation focuses on a point cloud segmentation/partitioning pipeline which is being developed in order to facilitate (and automate) post-processing tasks following point cloud acquisition. We address both scenarios described above. The pipeline consists of two main stages; a segmentation stage and a fitting stage. The initial segmentation stage[5] partitions the input point cloud into the various surfaces making up the site and produces a structure graph. The fitting stage then tries to maximise usage of every surface (node) in the structure graph. The presentation outlines current research progress, objectives and future directions.

References