Optimisation Techniques for Runtime Verification

Progress Report

By:
Helena Cuenca

Supervisors:
Dr. David Rydeheard
Dr. Giles Reger

School of Computer Science
University of Manchester, UK

May 21, 2014
# Contents

Abstract 2

1 Introduction 3
  1.1 Aim and Objectives ................................................. 4
  1.2 Scope ................................................................. 5
  1.3 Deliverables .......................................................... 5
  1.4 Report Structure ..................................................... 6

2 Background 7
  2.1 Runtime Verification Basics ........................................ 7
  2.2 Runtime Verification Frameworks ................................... 8
    2.2.1 Runtime Verification Taxonomies .............................. 8
    2.2.2 Main Runtime Verification Frameworks ...................... 9
  2.3 Quantified Event Automata (QEA) .................................. 12
  2.4 1st International Competition of Software for Runtime Verification ... 12

3 Research Methods 14
  3.1 QEA Definition ...................................................... 14
  3.2 Methodology .......................................................... 14
  3.3 Evaluation ............................................................ 16
    3.3.1 Competition Result .......................................... 16
    3.3.2 Internal Benchmarking ...................................... 16
    3.3.3 Evaluation with Respect to Other Tools ................... 16
  3.4 Tools ................................................................. 17

4 Progress 18
  4.1 Current Implementation ............................................ 18
    4.1.1 Project Structure ........................................... 18
    4.1.2 QEA Properties and Monitors Hierarchy ................... 19
  4.2 Initial Evaluation Results ....................................... 21
  4.3 Project Plan .......................................................... 21

References 23
Abstract

There is an increasing need to prove correctness of safety-critical and ubiquitous systems, as well as software components designed according to contracts. None of the traditional verification techniques works for all kind of systems and thus program verification is considered to be "unsolvable in general" [1]. Runtime Verification (RV) emerged as a complement to the available verification techniques, providing the possibility for the system under scrutiny to be notified when a fault is detected and act accordingly [2].

Runtime verification is defined as a dynamic analysis method, determining whether a run of a computational system satisfies a given correctness property [3]. A number of runtime verification techniques have been developed in the last decades; some of them are focused on expressiveness while some of them are focused on performance. A recent approach, Quantified Event Automata (QEA) [4], brings together both characteristics: being as expressive as the current most expressive formalisms and allowing for optimisation.

This year, 2014, the 1st International Competition of Software for Runtime Verification (CSRV-2014) [5] has been introduced as part of the 14th International Conference on Runtime Verification (RV’14) [6]. This project is preparing an entry for this competition in both online and offline monitoring tracks. The methodology in use consists in the implementation of a set of custom properties and corresponding monitors based on the QEA formalism. A hierarchy is defined in terms of the combinations of structural characteristics of the properties, such as the number of quantified variables, the quantification alternation and the optional use of free variables. Additionally, a range of optimisations techniques will be applied, e.g., redundancy elimination and eager monitoring. The expected output is an efficient and expressive system for runtime verification of parametric properties.
Chapter 1

Introduction

There is an increasing need to prove correctness of programs by reason of the emergence of safety-critical and ubiquitous systems. These are usually highly complex and require a significant effort in the development cycle to ensure proper operation, otherwise the consequences can be very serious. In addition, software components designed according to contracts (which specify preconditions, postconditions, input, output, etc.) represent a challenge in terms of verification [2]; in this case, the requirement is to prove that each component behaves according to the contract. None of the traditional verification techniques works for all kind of systems and thus program verification is considered to be "unsolvable in general" [1]. Static techniques, such as model checking and theorem proving, are either not scalable or expressive enough for many systems [3]; while dynamic techniques like testing are useful for finding bugs rather than showing correctness [2]. Runtime Verification (RV) emerged as a complement to the available verification techniques. One of the distinctive characteristics is the possibility for the system under scrutiny to be notified when a fault is detected and act accordingly [2]. Moreover, runtime verification can be used during program testing and during the operations phase [1].

Runtime verification is defined as a dynamic analysis method, determining whether a run of a computational system satisfies a given correctness property [3]. This process is performed by an external application, the monitor, and requires the system to be instrumented, which enables the monitor to observe the system. The instrumentation layer sends a trace (series of events) to the monitor, each event containing a name and, optionally, a number of data values as parameters. The property is set up in the monitor and defines a (possibly infinite) set of acceptable traces. According to the property and the events, the monitor determines if the system passes and produces a verdict: success or failure. One of the main challenges in runtime verification is to reduce the overhead caused by the instrumentation layer and the monitor itself over the system.

A number of runtime verification techniques have been developed in the last decades; some of them are focused on expressiveness i.e. increasing the properties that can be described, while some of them are focused on performance i.e. reducing the overhead of the monitoring process over the system. A recent approach [4] brings together both characteristics: being as expressive as the current most expressive formalisms and allowing for optimisation. This approach is Quantified Event Automata (QEA), where a property is represented by an extended finite state machine, combined with a notion of logical quantification.
The field of runtime verification has been evolving in recent years. In 2001, the First Workshop on Runtime Verification (RV’01) was held, event that takes place every year since then. In 2010, the workshop became an international annual conference with the main goal of bringing together researchers from academia and industry to debate about how programs execution should be monitored [7]. This year, 2014, the 1st International Competition of Software for Runtime Verification (CSRV-2014) [5] has been introduced as part of the 14th International Conference on Runtime Verification (RV’14). It is a response to the need of comparing the tools and techniques that have been developed, and create a benchmark suite for future reference [6]. Competitions of this kind, present across many fields of computer science, act as a catalyst for research in the area, allowing the community to focus on key challenges.

In this project we implement a set of properties and corresponding monitors based on the QEA formalism to participate in the competition. An initial prototype implementation was developed in Scala to establish the viability of the approach. This explored initial optimisation techniques and identified a number of special cases that could be developed further. Experiments were positive, however the exploratory nature of this implementation meant that it still remained inefficient in certain areas. A more efficient implementation building on these discoveries, and focusing on the identified special cases, is therefore required.

1.1 Aim and Objectives

The aim of the project is to explore, implement and evaluate optimisation techniques for efficient runtime verification by developing efficient runtime monitors based on the QEA formalism to participate in the 1st International Competition of Software for Runtime Verification (CSRV-2014).

In order to achieve this goal, the following specific objectives have been defined:

- Develop an entry to the competition in the following tracks: online monitoring for Java and off-line monitoring
- Implement the functionality defined by the QEA formalism based on a hierarchy of expressiveness, in terms of the number of quantified variables, quantification alternation, free variables use, deterministic/non-deterministic QEA and specific optimisations
- Identify specific cases that allow for optimisation, such as structural characteristics of monitored QEA, and implement optimisation techniques for these cases
- Benchmark the implementations using a range of real-world and artificial workloads relevant to the competition track
- Document the results of the optimisation exploration detailing the progress made with each technique
1.2 Scope

The runtime verification competition contains three tracks: online monitoring for C, online monitoring for Java and off-line monitoring. This project only considers the second two tracks.

To ensure that a reasonable implementation is ready for the competition, this project targets the most relevant optimisations techniques first. The optimisation space considered includes the following categories:

- **Structural optimisations**: depending on the structure of the monitored QEA different processing steps are required and different indexes can be used. This family of optimisations aims to automatically identify these structural characteristics of the QEA properties which allow us to take advantage of optimisation, e.g. the number of quantified variables, and use the relevant monitoring algorithm. Due to the complexity of the QEA structure there are many choices here. These optimisations are to be implemented before the competition.

- **Redundancy elimination**: different forms of redundancy can be identified and eliminated automatically. For example, objects monitored at runtime may become garbage and therefore be removed from the monitor.

- **Eager monitoring**: From a theoretical point of view it is sometimes possible to identify that an error must necessarily occur later by considering the reachability of a successful state. These techniques aim to finish monitoring early by identifying such points.

- **General indexing**: Where no structural indexes can be developed, general indexing techniques can be used. There are many choices here and previous work has identified promising new avenues. This part is being implemented by a collaborator and will be integrated to the implementation of this project.

1.3 Deliverables

The artefacts to be delivered in this project are:

- **Monitor**: the source code and the executable of the monitor suitable for release will be delivered

- **Technical documentation**: includes the documentation of the design of the system, the development documentation in Javadoc format and the user documentation in an associated website

- **Report of optimisation techniques**: the report will detail the techniques used along with an evaluation of their effectiveness
1.4 Report Structure

This report is structured as follows:

Chapter 2 sets a wider context by presenting a definition of runtime verification, its basic concepts and a summary of the most relevant existing RV frameworks, including those which laid the foundation for QEA. Additionally, the background of the competition is presented.

Chapter 3 presents a formal definition of QEA and the methodology in use for the implementation. In addition, it introduces the way in which the results of this project are going to be evaluated.

Chapter 4 provides details of the current implementation progress and the initial results obtained in tests already executed, as well as the project plan.
Chapter 2

Background

This chapter presents the background for the project. In Section 2.1, the basic concepts on runtime verification are introduced; Section 2.2 presents two taxonomies of runtime verification along with the main frameworks and some of their characteristics. Section 2.3 introduces Quantified Event Automata (QEA) through an example. Finally, Section 2.4 presents the background for the competition including the participants and some information about the evaluation.

2.1 Runtime Verification Basics

The literature presents two definitions of runtime verification. In the first one, runtime verification is a dynamic analysis method, determining whether a run of a computational system satisfies a given correctness property [3]. A second broader definition states that runtime verification is the discipline of computer science that studies verification techniques, their development and application [2].

The execution of a system can be described as a sequence of events where an event represents an action or selected states of the system [3]. An event can consist of a name only, in which case is called propositional; or a name and a number of data values as parameters, called parametric [4]. A trace is a finite sequence of events, and a property defines a (possibly infinite) set of acceptable traces. Traces and properties associated to propositional events are named propositional traces and propositional properties respectively, while those associated to parametric events are named parametric traces and parametric properties. Parametric properties allow to describe behaviour of objects [8].

An external application, the monitor, is typically used to check whether a trace satisfies a property. The property is set up in the monitor before it can read a finite trace to yield a verdict: yes/true or no/false [2]. This requires the instrumentation of the system under study, i.e., it has to be modified with additional code to send the relevant events to the monitor. There are different forms of instrumentation: source code instrumentation, byte-code instrumentation and object code instrumentation [1]. Ultimately, the instrumentation enables the monitor to observe the system. The monitor can also provide the system with some feedback, usually when a property is violated, so that the system can take corrective actions [3]. Figure 2.1 [3] shows the components involved in the runtime verification process described above.
2.2 Runtime Verification Frameworks

Numerous runtime verification frameworks have been developed in recent years. Making a classification of them is not an easy task for different reasons. One of them is that the components of each framework are diverse due to different interests of their creators; for example, some were conceived by people with a strong logical background who desired a rigorous formal specification language, so they consist of a formalism for properties definition; others were created by software engineering teams interested in the monitoring of specific types of applications, so they include the implementation of monitors. It seems there is not a general consensus within the community about which of these frameworks belong to runtime verification. Moreover, some of the characteristics of the frameworks are not intrinsic to them, an example being the supported specification language, given that the specification of a property in a particular language can usually be translated into another one.

2.2.1 Runtime Verification Taxonomies

We present here two previous attempts to define a taxonomy of runtime verification, which can help to organise and summarise what has been done in the field [9]. The first one was introduced in [10] along with a catalogue of runtime software-fault monitoring tools. This taxonomy is based on elements inherent to the monitoring systems, such as the specification language, monitor and event-handler, plus operational issues. The specification language element takes into account characteristics of the language used to
define monitored properties. The monitor element encloses the features of the application that observes and analyses the system under study. The event-handler element describes the way in which the system reacts to the violation of a property is classified. Finally, the operational issues element shows different components external to the monitoring system. Figure 2.2 shows the classification.

A second taxonomy of runtime verification was introduced in [9]. This one presents simpler categories than the previous one, including the type of monitored trace, the moment in which the monitoring process occurs (online/offline), the location of the monitor (inline/outline) and the application area. Figure 2.3 [9] shows the classification.

### 2.2.2 Main Runtime Verification Frameworks

A collection of the main runtime verification frameworks is presented in Table 2.1. Here we only consider the frameworks that are consistent with the definition given in Section 2.1, i.e., those that involve a monitor processing events produced by the system under scrutiny. The characteristics are:

- **Specification language**: the language(s) supported for the specification of the properties
Figure 2.3: Taxonomy of runtime verification [9]
• **Computational model:** it is said to be assertion-based when the monitoring process consists of runtime checks that are inserted in specific points in the code [1]. In this case, no information about the current state of the system is stored. In the rewriting-based model, a set of objects (e.g. formulas or facts) are rewritten for each event based on a set of rewrite rules. The set of objects represents the current state of the property and the rules update this state. In the automata-based model, properties are translated into a finite state machine. The monitor starts on an initial state and according to the events received from the system the execution state is updated, which determines the verdict. Finally, ad-hoc algorithms that do not follow any of the other patterns are considered program-based.

• **When monitoring occurs:** online if the monitoring occurs while the system is running, or offline if the monitoring is conducted over log files, after the system’s execution [3]

• **There the monitor is placed:** inline if the monitor code is part of the system, or outline if the monitor is external to it [3]

• **When verdicts are returned:** violation if the verdict is returned when the property becomes false, or validation if it is returned when it becomes true [3]

<table>
<thead>
<tr>
<th>Name</th>
<th>Specification Language</th>
<th>Computational Model</th>
<th>Online/Offline</th>
<th>Inline/Outline</th>
<th>Violation/Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hawk [12]</td>
<td>Eagle</td>
<td></td>
<td>online</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>Jass [10]</td>
<td>assertions</td>
<td>automata-based</td>
<td>online</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>jContractor [12]</td>
<td>contracts</td>
<td>–</td>
<td>online</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>J-Lo [13]</td>
<td>ParamLTL</td>
<td>rewriting-based</td>
<td>online</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>JPaX [10]</td>
<td>LTL</td>
<td>rewriting-based</td>
<td>offline</td>
<td>outline</td>
<td>violation</td>
</tr>
<tr>
<td>LARVA [2]</td>
<td>DATEs</td>
<td>automata-based</td>
<td>online</td>
<td>inline</td>
<td>–</td>
</tr>
<tr>
<td>LogScope [14]</td>
<td>rule-based</td>
<td>automata-based</td>
<td>online/outline</td>
<td>outline</td>
<td>violation/validation</td>
</tr>
<tr>
<td>LoLa [2]</td>
<td>linear µ-calculus with future and past modalities</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>JavaMaC [15]</td>
<td>PastLTL</td>
<td>–</td>
<td>online/outline</td>
<td>outline</td>
<td>violation</td>
</tr>
<tr>
<td>JavaMOP [8]</td>
<td>LTL, ERE, FSM, CFG, SRS, others</td>
<td>automata-based, program-based, rewriting-based</td>
<td>online/outline</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>P2V [13]</td>
<td>PSL</td>
<td>–</td>
<td>online</td>
<td>inline</td>
<td>validation/validation</td>
</tr>
<tr>
<td>PQL [12]</td>
<td>PQL</td>
<td>program-based</td>
<td>online</td>
<td>inline</td>
<td>validation</td>
</tr>
<tr>
<td>PTQL</td>
<td>SQL</td>
<td>program-based</td>
<td>online</td>
<td>outline</td>
<td>validation</td>
</tr>
<tr>
<td>QEA [4]</td>
<td>QEA</td>
<td>automata-based</td>
<td>online/outline</td>
<td>outline</td>
<td>violation/validation</td>
</tr>
<tr>
<td>RuleR [16]</td>
<td>rule-based</td>
<td>rewriting-based</td>
<td>online</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>Spec# [17]</td>
<td>contracts</td>
<td>assertion-based</td>
<td>online/outline</td>
<td>inline/outline</td>
<td>violation</td>
</tr>
<tr>
<td>Temporal Rover [18]</td>
<td>LTL, MTL</td>
<td>automata-based</td>
<td>online/outline</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>TraceContract [19]</td>
<td>state machines + temporal logic</td>
<td>program-based</td>
<td>online/outline</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>TraceMatches [20]</td>
<td>regular expressions</td>
<td>automata-based</td>
<td>online</td>
<td>inline</td>
<td>validation</td>
</tr>
</tbody>
</table>

Table 2.1: Runtime Verification Frameworks
Table 2.1 does not include the tools participant in the competition; due to their lack of maturity there is not published work about them yet. These tools are: ARTi-Mon, Breach, E-ACSL, jUnitRV, LogFire, MonPoly, prm4j, RiTHM, RTC, STePr, and ZOT+SOLOIST.

2.3 Quantified Event Automata (QEA)

One of the most powerful automata-based approaches is Quantified Event Automata (QEA). It was introduced in [4] as a formalism that is as expressive as the current most expressive formalisms, while allowing for optimisation. It takes the parametric trace slicing approach from JAVAMOP and extends it with three new characteristics: an event name can be associated to more than one variable list; non-quantified variables can vary during the monitoring process; and the addition of existential quantification to universal quantification.

Figure 2.4 [21] presents a simple property for the life cycle of a file: it starts closed, can be opened only if closed, and can be used or closed only when open. The property comprises an event automaton (EA) describing a set of states and transitions, and a quantification list, \( f \) universally quantified, indicating the property must be true for all files.

\[
\forall f \ \text{open}(f) \Rightarrow \text{close}(f) \\
\forall f \ \text{close}(f) \Rightarrow \text{use}(f) \\
\forall f \ \text{use}(f) \Rightarrow \text{open}(f)
\]

Figure 2.4: A QEA for the proper usage of a file [21]

A formal definition of QEA is given in Section 3.1.

2.4 1st International Competition of Software for Runtime Verification

Nowadays, it is very difficult to make an objective evaluation of runtime verification tools and techniques due to the lack of benchmarks against which to compare. In response to this, this year (2014) the 1st International Competition of Software for Runtime Verification has been introduced [5]. This event is of great importance since it is the first one of its kind and its participants are actively involved in the runtime verification field.

We give here a brief summary of [22], which presents the scoring method for the competition. It considers three different scores: Correctness score, concerned with the expressiveness; overhead score, related to the time added to the monitored system by the monitor; and memory utilisation score.

The correctness score in a specific benchmark is calculated as follows:

- 0, if the property cannot be expressed
-10, if the property can be expressed, but the monitored program crashes
-5, in online monitoring, if the property can be expressed but no verdict is reported after certain time
-5, in offline monitoring, if the property can be expressed but the monitor crashes
-5, if the property can be expressed, the tool does not crash, and the verdict is incorrect
10, if the tool does not crash, allows to express the property and produces the correct verdict

The overhead score considers in the case of offline monitoring, the elapsed time until a verdict is produced. In the case of online monitoring, it measures "how much longer a program takes to execute due to runtime monitoring". Similarly, the memory utilisation score considers the maximum memory allocated in the execution of offline monitoring tools, and the extra memory needed by the program due to runtime monitoring, in the case of online monitoring.

This project is preparing an entry for the online monitoring for Java and off-line monitoring tracks. The other participants are, for the online monitoring for Java track: Larva, jUnitRV, jUnitRV (MMT), JavaMOP, Java-MaC, prm4j; and for the offline monitoring track: ZOT+SOLOIST, LogFire, RiTHM-2, MonPoly ARTiMon -2, STePr, Breach.
Chapter 3

Research Methods

This chapter presents in Section 3.1 the formal definition of QEA, as the main method of this project. Section 3.2 explains the hierarchy of QEA implementations and monitors. Finally, Section 3.3 describes how the evaluation of the project is going to be done.

3.1 QEA Definition

The work in this project is based on [4] which presents as follows the formal definition of Event Automaton (EA) and Quantified Event Automaton (QEA).

**Definition 1 (Event Automaton)** An Event Automaton \((Q, \mathcal{A}, \delta, q_0, F)\) is a tuple where 
- \(Q\) is the set of states,
- \(\mathcal{A} \subseteq \text{Event}\) is the alphabet,
- \(\delta \in (Q \times \mathcal{A} \times \text{Guard} \times \text{Assign} \times Q)\) is the transition set,
- \(q_0\) is the initial state, and
- \(F \subseteq Q\) is the set of final states.

**Definition 2 (Quantified Event Automaton)** A QEA is a pair \((E, \Lambda)\) where \(E\) is an EA and \(\Lambda \in (\{\forall, \exists\} \times \text{vars}(E) \times \text{Guard})^\ast\) is a list of quantified variables with guards.

3.2 Methodology

One of the principles guiding the design of the system is the construction of a hierarchy of properties and monitors. Here we make use of the observation that many common forms of specification only use a subset of the QEA available features i.e. conform to a particular structural restriction. Properties are then categorised according to their structural characteristics and a custom QEA implementation and monitor for each category is built. For example, a property represented by a deterministic QEA with a single quantified variable and no free variables requires much less functionality and data structures than a non-deterministic, multiple quantified QEA with free variables. For this reason, the core of the system consists of a set of custom QEA implementations and corresponding monitors with diverse features. They differ from each other in the expressiveness of the QEA, which is defined in terms of the combination of values for the following dimensions:

- **Number of quantified variables**: different indexing techniques are used according to the number of quantified variables, reducing the complexity of the indexing structures and the matching process. For instance, in the single quantifier case it is possible to index directly on the quantified value.
Table 3.1: Basic Space

- **Quantification alternation**: The combination of quantifications (universal and existential) for the variables allows different optimisations. Here we consider not only the standard quantifications here but negations.

- **Free variables use**: A monitor for a QEA without free variables is simplified by removing the structures for free variables bindings, the support for guards and assignments and the matching process of the arguments of an event with the parameters defined.

- **Deterministic/non-deterministic**: When the monitored QEA is deterministic, the processing complexity is reduced by removing program repetitions and the constant creation of structures required to store multiple states/configurations.

- **Specific optimisation**: Sometimes it is worth to optimise a very specific common case. For example the fixed quantified variable case consisting in a QEA where all the events contain as the first parameter the unique quantified variable. Hence, the value of the quantified variable is instantly retrieved i.e. the affected binding, while the rest of the parameters are matched against the free variables only.

It is not necessary to create a different QEA implementation and monitor for each one of the possible combinations of values. Some cases are grouped together as long as it makes sense. For example, in the case of a single quantified variable, the quantification (universal or existential) only represents a small change in the way the verdict is computed, so the same implementation is used.

Table 3.1 shows what we have called the "basic space"; this is, the different cases arising from the combination of values for the dimensions previously described, specifically when there is one quantified variable or no quantified variables at all. Each case
represents then a type of property with specific characteristics. For example, case 1 illustrates a property with a single quantified variable, universal quantification, without free variables and deterministic, which is one of the simplest cases. This case has an associated QEA implementation and monitor with the minimum complexity required to support the characteristics.

3.3 Evaluation

Evaluation is an essential part of this project to determine if aims and objectives were met. Section 3.3.1 presents the competition result as one of the main components of evaluation; section 3.3.2 explains how the different QEA implementations and monitors will be evaluated against each other; finally section 3.3.3

3.3.1 Competition Result

One of the main objectives of this project is to develop an entry in the 1st International Competition of Software for Runtime Verification (CSRV-2014) [5], so the result of the competition is an important and clear indicator of how expressive is QEA as a specification formalism with respect to other approaches, and how effective is the system in terms of time overhead and memory consumption with respect to other tools. In other words, it will allow to determine if a good balance between expressiveness and efficiency was achieved.

3.3.2 Internal Benchmarking

A general representation of QEA was developed to allow the specification of the components of a property: quantified and free variables, states, transitions, events, guards, assignments, etc. The structure is inspected and then the simplest QEA implementation and monitor containing the required features is selected. As this selection is done automatically based on the property characteristics, one of the evaluation components is to determine whether this selection is the most appropriate i.e. the chosen implementation has the best performance among all the available implementations.

It is expected that simpler QEA implementations and monitors show a better performance, however it is possible to find some cases where an implementation that was considered more complex performs better that a simpler one. This evaluation component will allow to identify these cases and make the corresponding adjustments. The properties defined in DaCapo benchmarks and the rover case study [19] will be tested with all the applicable implementations.

3.3.3 Evaluation with Respect to Other Tools

As a preamble to the competition, the system is being tested against JAVAMOP and the previous QEA implementation in Scala.
3.4 Tools

The following tools have been identified to be used in this project:

- Programming language: Java. The system is being developed in Java 7
- Development environment: Eclipse. Some plugins have been installed to allow additional features such as version control and tests coverage measurement
- Unit test framework: JUnit. Currently in use to evaluate the correctness of the system
- Profiling tools such as VisualVM will allow to identify the execution points affecting the performance of the system in terms of time overhead and memory utilisation
Chapter 4

Progress

A substantial part of this project has been already developed. Section 4.1 provides the details of the current implementation of the system, including the structure of the Java project and the hierarchy of QEA properties and monitors. Section 4.2 presents the results of the initial evaluation.

4.1 Current Implementation

4.1.1 Project Structure

The Java project consists of the following packages:

- **structure**: contains the QEA implementations according to the different structural characteristics of the properties. Additionally, the implementation for bindings, transitions, assignments and guards.

- **monitoring**: contains the monitors corresponding to the QEA properties in the previous package and the monitor factory that, given a QEA property, selects the appropriate monitor.

- **properties**: contains a sample of properties such as those from the rover case study, the use file property and the action bidding property.

- **creation**: contains the QEA builder that, given a QEA property, instantiates the simplest QEA implementation required to support the characteristics of the properties.

- **test**: contains a set of unit tests to evaluate the correctness of the system.

- **benchmark**: contains the functionality to test the performance of the system. Currently, there is an evaluation suite for the properties defined in the rover case study as well as a test for the Resource Lifecycle property with different implementations of QEA properties and monitors.

- **util**: contains some helper classes with methods used across the system.
4.1.2 QEA Properties and Monitors Hierarchy

The hierarchy of the QEA properties and monitors is given by the combination of values for the dimensions introduced in Section 3.2. The basic space described in is already implemented. Additionally, a fully-functional monitor, which is a naive implementation of the process described in [4] was implemented. Figure 4.1 presents the hierarchy of classes for QEA.

![Hierarchy of QEA properties](image)

Figure 4.1: Hierarchy of QEA properties

Figure 4.2 presents the hierarchy of classes for the monitors. It can be observed that the classes comply with a naming convention. In the case of the QEA:

\[
[QVar0|QVar1|QVar2|...|QVarN]_[NoFVar|FVar]_[Det|NonDet]_[SpecOpt]_QEA
\]

The naming convention for the monitors is as follows:

\[
[QVar0|QVar1|QVar2|...|QVarN]_[NoFVar|FVar]_[Det|NonDet]_[SpecOpt]_QEAMonitor
\]
Figure 4.2: Hierarchy of monitors
<table>
<thead>
<tr>
<th>Test</th>
<th>Monitor 1</th>
<th>Monitor 2</th>
<th>Monitor 3</th>
<th>Monitor 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>169064</td>
<td>109</td>
<td>94</td>
<td>63</td>
</tr>
<tr>
<td>Test 2</td>
<td>167389</td>
<td>109</td>
<td>78</td>
<td>62</td>
</tr>
<tr>
<td>Test 3</td>
<td>167203</td>
<td>109</td>
<td>78</td>
<td>62</td>
</tr>
<tr>
<td>Test 4</td>
<td>167635</td>
<td>110</td>
<td>94</td>
<td>62</td>
</tr>
<tr>
<td>Test 5</td>
<td>167288</td>
<td>110</td>
<td>93</td>
<td>62</td>
</tr>
<tr>
<td>Avg. Time</td>
<td>167715.8</td>
<td>109.4</td>
<td>87.4</td>
<td>62.2</td>
</tr>
</tbody>
</table>

Table 4.1: Execution times in milliseconds for the ResourceLifecycle property test

### 4.2 Initial Evaluation Results

The ResourceLifecycle property from the rover case study [19] was used to test how different monitors with more general or specific functionality perform in a fixed scenario. This property specifies the possible actions over a resource: it can be requested and then either denied or granted. When granted it can be rescinded multiple times before it is cancelled [19].

The test consisted in creating 5,000 resources and generating randomly 1,000,000 events conforming to the property. It was executed five times with each one of the incremental monitors under study and the time required to process the trace was measured.

First, the fully-functional monitor was used (Incr_Naive_Det_Monitor). For this monitor the average execution time was around 167k milliseconds. The second monitor used (Incr_QVar1_FVar_NonDet_QEAMonitor) differs from the first one in the number of quantified variables supported: while the first one accepts any number, the second one is restricted to one quantified variable, thus direct indexing is used. The average execution time for this monitor was 109.4 milliseconds, which is 1500X faster. This large speedup in this case was expected as we have gone from a case without indexing to one using indexing.

In the third monitor (Incr_QVar1_FVar_Det_QEAMonitor) the support for nondeterminism is removed, so that there is no need to deal with multiple states for each resource. This characteristic is not required in the ResourceLifecycle property as every state has at most one outgoing transition for each event. In this case the average execution time was 87.4 milliseconds, 1.25X faster than the previous monitor.

Finally, a fourth monitor (Incr_QVar1_NoFVar_Det_QEAMonitor) was tested where the support for free variables is removed. The only variable present in all events is $r$ which is quantified. The result was an average execution time of 62.2 milliseconds, 1.4X faster than the third monitor. Overall a 2700x speedup was achieved, and after moving to an indexing approach it was 1.75X faster.

Table 4.1 presents the results of the five tests using the four monitors.

### 4.3 Project Plan

Figure 4.3 is a Gantt chart showing the main activities of the project with their duration, as well as the current progress.
Figure 4.3: Project plan
References


