L-Università ta' Malta

Ensuring Correctness in Distributed Systems

An example report

by

Duncan Paul Attard

Department of Computer Science
Faculty of ICT

supervised by

Adrian Francalanza
Luca Aceto
Anna Ingólfsdóttir

November 9, 2020

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

Ensuring Correctness in Distributed Systems

submitted is my work, except where acknowledged and referenced.

Duncan Paul Attard

November 9, 2020

Abstract

A number of software systems today are built in terms of independently executing components that typically reside on different physical locations. While these software organisations offer a number of advantages, including the use of replication to improve robustness and quality of service, they are hard to design and implement. Ascertaining their correctness, therefore, becomes a chief concern. Traditional formal verification techniques, such as testing or model checking, tend to be applied with limited success in these scenarios due to a number of reasons. Runtime verification may be employed as a lightweight and dynamic alternative that complements the aforementioned verification approaches.

...

Contents

List of Figures

Chapter 1

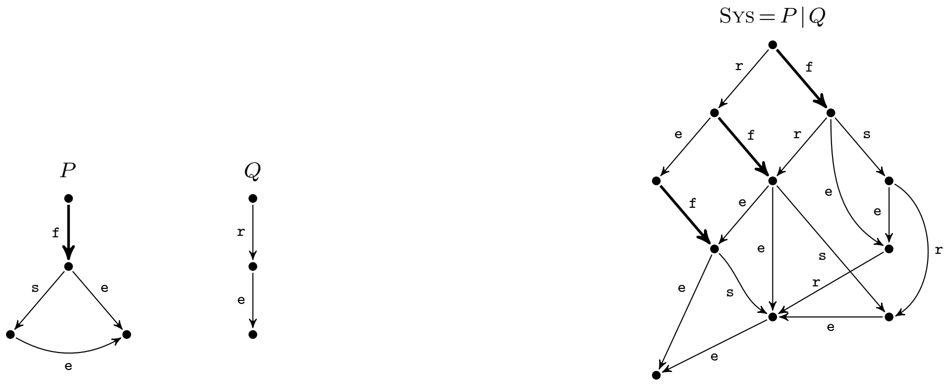
Introduction

Numerous software systems are nowadays architected in terms of *asynchronous components* [9, 18, 1] that execute independently to one another without recourse to a global clock or shared state. Instead, components interact together via well-defined interfaces and non-blocking messaging [17] to create dynamic and loosely-coupled software organisations. Such architectures facilitate incremental updates, tolerate independent component failures and permit the various units of execution to be *distributed* across different locations [20, 11]. Despite their advantages, these systems are notoriously hard to design, and even harder to program and get right, and ensuring their correctness in terms of their expected behaviour becomes paramount.

1.1 Distributed Runtime Verification

While distributed systems inherit the characteristics inherent to asynchronous settings, their execution is further complicated due to physical constraints, such as the lack of a global clock and possibility of independent failures [16, 11]. In the local asynchronous case, traditional pre-deployment verification techniques such as model checking and testing [19, 22] often *scale poorly* because the set of execution paths considered is invariably dwarfed by the vast number of possible execution paths of the system. In a distributed scenario, use of these verification approaches is often problematic, if not *impractical*, due to the aforementioned complications.

Runtime Verification (RV) [21, 13] is complementary approach that evades some of the limitations of pre-deployment techniques by deferring the analysis until runtime. It employs *monitors* to *incrementally* analyse the system's behaviour (exhibited as a sequence of *trace* events) up to the current execution point, to determine whether a correctness specification under investigation is satisfied or violated.



(a) *The labelled transition systems for processes P and Q*

(b) *System* SYS *as a composition of processes P and Q*

Fig. 1.1. *Local and global states of a component-based system*

...

Example 1.1. Consider the system SYS composed of processes P and Q, given in terms of the *labelled transition systems* in fig. 1.1. P can initially fork (denoted by action `f`) a process, after which it either sends (`s`) a message and exits (`e`), or exits (`e`) immediately; Q is a simpler process that performs a single receive (`r`) and exits (`e`). A possible correctness property states that *"*SYS *does not fork processes at startup"*.

When SYS exhibits the witness trace ***f.r.e.e***, the monitor can detect a violation of this property. For a different execution interleaving, *e.g.* ***r.e.f.e*** (where ***f*** is not the first event), the typical RV analysis would be unable to detect the fact that SYS is capable of performing `f`. Recouping this *lack of precision* is possible, but this would require the specification to consider *all* the possible trace event permutations that the composition of P and Q may exhibit (fig. 1.1b). One easily observes that adding new components to SYS aggravates the specification task to the point where it becomes unwieldy and error-prone. Reformulating the original property to consider P in isolation, *i.e., "P does not fork processes at startup"*, eliminates the need to account for the behaviour of other nonrelevant components. ■

...

Chapter 2

Background

This chapter presents an overview of the background literature and state of the art in the field of RV for distributed systems. We first give a brief synopsis of RV, and define the terms introduced in the preceding chapter. This is followed by an account of distributed systems that acquaints readers with the concepts used throughout this report. Finally, a series of recent works is discussed, comparing and contrasting similarities and differences between them. This exposition should reassert the current limitations in the area, and in doing so, underscore the novelty of our research contributions proposed in ch. 1.

We start by briefly recalling the two most basic sorting algorithms, since these shall be used later when comparing our work to others. The bubble sort (lst. 1 left) is a simple sorting algorithm that works by repeatedly going through the list to be sorted, comparing adjacent elements and swapping them if they are in the wrong order. This procedure is repeated until the list is left in a sorted order. . . .

```
Require: Array a of numbers
Ensure: Sorted a in ascending order
procedure BubbleSort(a)
  int i,j,tmp
  n = length(a)
  for j = 1 to n do
    for i = 0 to n−1 do
      # Swap current if larger than next
      if a[i] > a[i+1] then
        tmp = a[i]
        a[i] = a[i+1]
        a[i+1] = tmp
      end if
    end for
  end for
end procedure
```

```
Require: Array a of numbers
Ensure: Sorted a in ascending order
procedure InsertionSort(a)
  int i,j,key
  n = length(a)
  for j = 2 to n do
    key = a[j]
    i = j − i
    # Insert a[j] in the sorted sequence a[1...j−1]
    while i > 0 ∧ a[i] > key do
      a[i+1] = a[i]
      i = i − 1
    end while
    a[i+1] = key
  end for
end procedure
```

Lst. 1. *The bubble and insertion sort algorithms*

	Centralised	*Global state*	*Asynchronous*	*Shared memory*	*Message passing*	*Total ordering*	*Static setup*
Attard *et al.* [2]	✓	✓	✓	·	✓	·	✓
Attard *et al.* [3]	·	✶	✓	·	✓	·	✓
Bauer *et al.* [4]	·	✓	·	·	✓	✓	✓
Berkovich *et al.* [6]	✓	✓	·	✓	·	✓	✓
Cassar *et al.* [7]	✓	✓	✓	·	✓	·	✓
Cassar *et al.* [8]	✓	✓	✓	·	✓	·	✓
Colombo *et al.* [10]	✓	·	✓	·	✓	·	·
El-Hokayem *et al.* [12]	✶	✓	·	·	·	✓	✓
Falcone *et al.* [14]	✓	✓	·	·	✓	✓	✓
Francalanza *et al.* [15]	✓	✓	✓	·	✓	·	✓
Sen *et al.* [23]	·	✓	✓	✓	·	·	✓

Tbl. 2.1. *State-of-the-art on concurrent monitoring classified by characteristics*

There are a number of works [15, 2, 3, 8, 7, 10, 23, 6] that address RV in a local concurrent setting; others [4, 14] use the term decentralised to refer to synchronous monitoring. In another work, El-Hokayem *et al.* [12] present a framework whereby orchestrated and choreographed monitor arrangements for LTL_3 [5] can be studied in terms of their performance, including the amount of memory consumed by monitor participants and the number of messages exchanged between them. Although relevant to ours, the cited works do not assume any of the defining characteristics of distributed systems, *e.g.*, different network locations, partial failure, *etc.*, and therefore, shall not be the main focus of the review that follows. A comparison of their various characteristics is nevertheless provided in tbl. 2.1 for the sake of completeness.

...

Bibliography

[1] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A Foundation for Actor Computation. *JFP*, 7(1):1–72, 1997.

[2] D. P. Attard and A. Francalanza. A Monitoring Tool for a Branching-Time Logic. In *RV*, volume 10012 of *LNCS*, pages 473–481. Springer, 2016.

[3] D. P. Attard and A. Francalanza. Trace Partitioning and Local Monitoring for Asynchronous Components. In *SEFM*, volume 10469 of *LNCS*, pages 219–235. Springer, 2017.

[4] A. Bauer and Y. Falcone. Decentralised LTL Monitoring. *FMSD*, 48(1-2):46–93, 2016.

[5] A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *TOSEM*, 20(4):14:1–14:64, 2011.

[6] S. Berkovich, B. Bonakdarpour, and S. Fischmeister. Runtime verification with minimal intrusion through parallelism. *FMSD*, 46(3):317–348, 2015.

[7] I. Cassar and A. Francalanza. On Synchronous and Asynchronous Monitor Instrumentation for Actor-Based Systems. In *FOCLASA*, volume 175 of *EPTCS*, pages 54–68, 2014.

[8] I. Cassar and A. Francalanza. On Implementing a Monitor-Oriented Programming Framework for Actor Systems. In *IFM*, volume 9681 of *LNCS*, pages 176–192. Springer, 2016.

[9] D. Chappell. *Enterprise Service Bus: Theory in Practice.* O'Reilly Media, 2004.

[10] C. Colombo, A. Francalanza, and R. Gatt. Elarva: A Monitoring Tool for Erlang. In *RV*, volume 7186 of *LNCS*, pages 370–374. Springer, 2011.

[11] J. Dollimore, T. Kindberg, and G. Coulouris. *Distributed Systems: Concepts and Design.* Addison Wesley, 2005.

[12] A. El-Hokayem and Y. Falcone. Monitoring Decentralized Specifications. In *ISSTA*, pages 125–135. ACM, 2017.

[13] Y. Falcone, J. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[14] Y. Falcone, M. Jaber, T. Nguyen, M. Bozga, and S. Bensalem. Runtime Verification of Component-Based Systems in the BIP Framework with Formally-Proved Sound and Complete Instrumentation. *SoSyM*, 14(1):173–199, 2015.

[15] A. Francalanza and A. Seychell. Synthesising Correct Concurrent Runtime Monitors. *FMSD*, 46(3):226–261, 2015.

[16] S. Ghosh. *Distributed Systems: An Algorithmic Approach*. Chapman and Hall/CRC, 2014.

[17] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[18] N. M. Josuttis. *SOA in Practice: The Art of Distributed System Design: Theory in Practice*. O'Reilly Media, 2007.

[19] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[20] G. V. K. *Elements of Distributed Computing*. Wiley India, 2014.

[21] M. Leucker and C. Schallhart. A brief account of runtime verification. *JLAP*, 78(5):293–303, 2009.

[22] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. Wiley, 2011.

[23] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Decentralized Runtime Analysis of Multi-threaded Applications. In *IPDPS*. IEEE, 2006.

Appendix A

Trace Partitioning and Local Monitoring for Asynchronous Components

The paper entitled *"Trace Partitioning and Local Monitoring for Asynchronous Components"* was published in SEFM 2017 under Springer LNCS.

. . .