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

Lowering overheads for Runtime Enforcement

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

Choosing the Appropriate Enforcement mechanisms

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

References

1. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, pages 1–72, 1997.
2. J. Armstrong. *Programming Erlang*. The Pragmatic Bookshelf, 2007.
3. I. Cassar and A. Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. In *FOCLASA*, EPTCS, pages 54–68, 2014.
4. F. Cesarini and S. Thompson. *ERLANG Programming*. O'Reilly, 1st edition, 2009.
5. Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.
6. P. Haller and F. Sommers. *Actors in Scala*. Artima Inc., USA, 2012.
7. M. Leucker and C. Schallhart. A brief account of Runtime Verification. *JLAP*, 78(5):293 – 303, 2009.
8. I. A. Mason and C. L. Talcott. Actor languages their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 220:409 – 467, 1999.

Mobile Erlang Computations to Enhance Performance, Resource Usage and Reliability

Adrian Francalanza and Tyron Zerafa

University of Malta

1 Introduction

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

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

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

2 Control over Locality

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

Any computation of a functional nature is deterministic (i.e., its execution can be completely established from the computation's executed code and its initial inputs). For instance, the execution of the factorial algorithm is of a functional nature since it cannot be affected by any other computation; this is just a mathematical function. Thus, it is possible to application developers to determine its requirements (e.g., processing power) and **initialize** it over the best location (e.g., the most lightly-loaded location).