Computer Languages, Systems & Structures ■ (■■■■) ■■■–■■■

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Domains: Sharing state in the communicating event-loop actor model ☆

Joeri De Koster [a,*], Stefan Marr [b], Tom Van Cutsem [a], Theo D'Hondt [a]

[a] Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, Belgium
[b] Inria Lille, 40, avenue Halley, 59650 Villeneuve d'Ascq, France

ARTICLE INFO

Article history:
Received 10 May 2015
Received in revised form
16 October 2015
Accepted 11 January 2016

Keywords:
Actor model
Domains
Synchronization
Shared state
Race-free mutation

ABSTRACT

The actor model is a message-passing concurrency model that avoids deadlocks and low-level data races by construction. This facilitates concurrent programming, especially in the context of complex interactive applications where modularity, security and fault-tolerance are required. The tradeoff is that the actor model sacrifices expressiveness and safety guarantees with respect to parallel access to shared state.

In this paper we present *domains* as a set of novel language abstractions for safely encapsulating and sharing state within the actor model. We introduce four types of domains, namely immutable, isolated, observable and shared domains that each is tailored to a certain access pattern on that shared state. The domains are characterized with an operational semantics. For each we discuss how the actor model's safety guarantees are upheld even in the presence of conceptually shared state. Furthermore, the proposed language abstractions are evaluated with a case study in Scala comparing them to other synchronization mechanisms to demonstrate their benefits in deadlock freedom, parallel reads, and enforced isolation.

1. Introduction

In practice, the actor model is made available either via dedicated programming languages (actor languages) or via libraries in existing languages. Actor languages are mostly *pure*, in the sense that they usually enforce strict isolation of actors: the state of an actor is fully encapsulated, cannot leak, and asynchronous access to it is enforced. Examples of pure actor languages include Erlang [1], SALSA [2], E [3], AmbientTalk [4], and Kilim [5]. The major benefit of pure actor languages is that the developer gets strong safety guarantees: low-level data races are ruled out by design. The downside is that this strict isolation severely restricts the way in which access to shared resources can be expressed.

At the other end of the spectrum, we find actor libraries, which are often built on top of a shared-memory concurrency model. For Java alone, examples include ActorFoundry [6], Actor Architecture [7], ProActive [8], AsyncObjects [9], JavAct [10], Jetlang [11], and AJ [12]. Scala, which inherits shared-memory multithreading as its standard concurrency model from Java, features multiple actor frameworks, such as Scala Actors [13] and Akka [14]. These libraries have in common that they do not

☆ Note for reviewers: this paper builds on De Koster J, Marr S, D'Hondt T, Van Cutsem T. Tanks: Multiple reader, single writer actors. In Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 13, 2013; pp. 61–68.

* Corresponding author.
E-mail addresses: jdekoste@vub.ac.be (J. De Koster), stefan.marr@inria.fr (S. Marr), tvcutsem@vub.ac.be (T. Van Cutsem), tjdhondt@vub.ac.be (T. D'Hondt).

http://dx.doi.org/10.1016/j.cl.2016.01.003
1477-8424/

enforce actor isolation, i.e., they cannot guarantee that actors do not share mutable state. There exist techniques to circumvent this issue. For example, it is possible to extend the type system of Scala to guarantee data-race freedom [15]. However, it is easy for a developer to use the underlying shared-memory concurrency model as an "escape hatch" when direct sharing of state is the most natural or most efficient solution. Once the developer chooses to go this route, the benefits of the high-level actor model are lost and the developer typically has to resort to other ad hoc synchronization mechanisms to prevent data races.

A recent study [16] has shown that 56% of the examined Scala programs use actors purely for concurrency in a non-distributed setting. That same study has shown that in 68% of those applications, the programmers' mixed actor library constructs with other concurrency mechanisms. When asked for the reason behind this design decision, one of the main motivations' programmers brought forward was inadequacies of the actor model, stating that certain protocols are easier to implement using shared-memory than using asynchronous communication mechanisms without shared state.

In conclusion, pure actor languages are often more strict, which allows them to provide strong safety guarantees. The downside is that they often restrict the expressiveness when it comes to modeling access to a shared resource while impure actor libraries are more flexible at the cost of some of those safety guarantees.

The goal of this work is to enable safe and expressive state sharing among actors in a pure actor language. In this paper, we study the actor model only with the aim of applying it to improve shared-memory concurrency. We do not consider applications that require actors to be physically distributed across machines. In addition, while our approach applies to a wide variety of actor models, in this paper we focus on the Communicating Event-Loop Actor Model (CEL Actor Model) [3] specifically (see Section 2). To achieve this goal, we aim to relax the strictness of the event-loop model via the controlled use of novel language abstractions. We aim to improve state sharing among actors on two levels:

Safety The isolation between actors enforces a structure on programs and thereby facilitates reasoning about large-scale software. Consider for instance a plug-in or component architecture. By running plug-ins in their own isolated actors, we can guarantee that they do not violate certain safety and liveness invariants of the "core" application. Thus, as in pure actor languages, we seek an actor system that maintains strong language-enforced guarantees and prevents low-level data races and deadlocks by design.

Expressiveness Many phenomena in the real world can be naturally modeled using message-passing concurrency, for instance telephone calls, e-mail, digital circuits, and discrete-event simulations. Sometimes, however, a phenomenon can be modeled more directly in terms of shared state. Consider for instance the scoreboard in a game of football, which can be read in parallel by thousands of spectators. As in impure actor libraries, we seek an actor system in which one can directly express access to shared mutable state, without having to encode shared state as encapsulated state of a shared actor. Furthermore, by enabling direct *synchronous* access to shared state, we gain stronger synchronization constraints and prevent the inversion of control that is characteristic for interacting with actors using asynchronous message-passing.

This work extends and unifies the language abstractions presented in De Koster et al. [17] and De Koster et al. [18] in a novel programming model. This unified programming model was formalized in an operational semantics (see Appendix A) and serves as a platform for experimenting with new language abstractions expressing shared state within the actor model. Our validation shows the usefulness of our language abstractions based on a case study in Scala (see Section 5).

This paper is structured as follows. Section 2 introduces the Communicating Event-Loop Actor Model and SHACL [19], a communicating event-loop actor language developed as a platform for experimenting with new language features. Section 3 shows the lack of good abstractions to represent shared resources in modern actor systems by giving an overview of the drawbacks of common techniques to represent shared state in the communicating event-loop model. Section 4 presents the taxonomy that led to the design of four distinct types of domains, namely immutable, isolated, observable, and shared domains. Subsequently each of the different types of domains and their use in SHACL is discussed. Section 5 validates the usefulness of the domain model. We start with a survey of synchronization patterns used by developers in existing open-source Scala projects and then compare these patterns with our domain abstractions. Section 6 discusses related work and Section 7 concludes this paper.

2. SHACL: a communicating event-loop actor language

The communicating event-loop (CEL) model: The CEL model was originally intended as an object-oriented programming model for secure distributed computing [3]. The goal of our Domain Model is to increase the expressiveness of the CEL model in a shared-memory context by allowing safe access to shared state. In the CEL model an actor is represented by a *vat*. Throughout this paper the terms *actor*, *event-loop actor* and *vat* are used interchangeably and always refer to "actors" as defined by the CEL model. As shown in Fig. 1, each vat has a single thread of control (the event-loop), a heap of objects, a stack, and an event queue. Each object in a vat's object heap is *owned* by that vat and those objects all share the same event-queue and event-loop. This means that vats are strictly isolated from one another.

Please cite this article as: De Koster J, et al. Domains: Sharing state in the communicating event-loop actor model. Computer Languages, Systems & Structures (2016), http://dx.doi.org/10.1016/j.cl.2016.01.003