Science of Computer Programming 98 (2015) 140–158

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Domains: Safe sharing among actors ☆

Joeri De Koster *, Stefan Marr, Theo D'Hondt, Tom Van Cutsem

Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, Belgium

HIGHLIGHTS

- The actor model is not suited for expressing access to shared mutable state.
- The domain model enables safe, expressive and efficient sharing of mutable state.
- Our model retains the safety and liveness properties of the original actor model.
- We provide an operational semantics that serves as a reference work.

ARTICLE INFO

Article history:
Received 5 March 2013
Received in revised form 31 January 2014
Accepted 5 February 2014
Available online 18 February 2014

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

ABSTRACT

The actor model is a concurrency model that avoids issues such as deadlocks and data races by construction, and thus facilitates concurrent programming. While it has mainly been used for expressing distributed computations, it is equally useful for modeling concurrent computations in a single shared memory machine. In component based software, the actor model lends itself to divide the components naturally over different actors and use message-passing concurrency for the interaction between these components. The tradeoff is that the actor model sacrifices expressiveness and efficiency with respect to parallel access to shared state.

This paper gives an overview of the disadvantages of the actor model when trying to express shared state and then formulates an extension of the actor model to solve these issues. Our solution proposes *domains* and *synchronization views* to solve the issues without compromising on the semantic properties of the actor model. Thus, the resulting concurrency model maintains deadlock-freedom and avoids low-level data races.

1. Introduction

Traditionally, concurrency models fall into two broad categories: message-passing versus shared-state concurrency control. Both models have their relative advantages and disadvantages. In this paper, we explore an extension to a message-passing concurrency model that allows safe, expressive and efficient sharing of mutable state among otherwise isolated concurrent components.

A well-known message-passing concurrency model is the actor model [1]. In this model, applications are decomposed into concurrently running actors. Actors are isolated (i.e., have no direct access to each other's state), but may interact via asynchronous message passing. While originally designed to model open, distributed systems, and thus often used as

☆ This paper is an extension of: De Koster, J., Van Cutsem, T. and D'Hondt, T. (2012), Domains: safe sharing among actors, in: Proceedings of the 2nd edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions, pp. 11–22.

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

http://dx.doi.org/10.1016/j.scico.2014.02.008
0167-6423/

a distributed programming model, they remain equally useful as a more high-level alternative to shared-memory multithreading. Both component-based and service-oriented architectures can be modeled naturally using actors. It is important to point out that in this paper, we restrict ourselves to the use of actors as a concurrency model, not as a distribution model.

In practice, the actor model is either made available via dedicated programming languages (actor languages), or via libraries in existing languages. Actor languages are mostly *pure*, in the sense that they often strictly enforce the 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 [2], E [3], AmbientTalk [4], SALSA [5] and Kilim [6]. 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 drawback is that such pure actor languages make it difficult to express shared mutable state. Often, one needs to express shared state in terms of a shared actor encapsulating that state, which has several disadvantages, as will be discussed in Section 3.3. Actor libraries typically do not share these restrictions but also do not provide the strong safety guarantees because these libraries are often for languages whose concurrency models are based on shared-memory multithreading. Examples for Java include ActorFoundry [7] and AsyncObjects [8].

Scala, which inherits shared-memory multithreading as its standard concurrency model from Java, features multiple actor frameworks, such as Scala Actors [9] and Akka [10]. What these libraries have in common is that they do not enforce actor isolation, i.e., they do not guarantee that actors do not share mutable state. On the other hand, 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. However, 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 manual locking to prevent data races.

The goal of this work is to enable safe, expressive and efficient state sharing among actors:

Safety The isolation between actors structures 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 safety and liveness invariants such as deadlock and race-condition freedom of the "core" application. Thus, as in pure actor languages, we want 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 want an actor system in which one can directly express read/write 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, i.e., without requiring a message send, we gain stronger synchronization constraints and prevent the inversion of control that is characteristic for interacting with actors, as interaction is asynchronous.

Efficiency Today, multicore hardware is becoming the prevalent computing platform, both on the client and the server [11]. While multiple isolated actors can be executed perfectly in parallel by different hardware threads, shared access to a single actor can still form a serious sequential bottleneck. In pure actor languages, shared mutable state is modeled with a specific actor and all requests sent to it are serialized, even if some requests could be processed in parallel, e.g., requests to simply read or query some of the actor's state. Pure actors lack multiple-reader, single-writer access, which is required to enable truly parallel reads of shared state.

In this paper, we propose *domains*, an extension to the actor model that enables safe, expressive and efficient sharing among actors. Since we want to provide strong language-level guarantees, we present domains as part of a small actor language called SHACL.[1] In terms of sharing state, our approach strikes a middle ground between what is achievable in a pure actor language versus what can be achieved using impure actor libraries. An interpreter for the SHACL language is available online.[2] This paper also provides an operational semantics for a small subset of our language named SHACL-LITE. This operational semantics serves as a complete and detailed specification of our model.

The rest of this paper is structured as follows, Section 2 gives a short overview of the communicating event-loops model on which the SHACL model was based. In Section 3, we present a number of problems that occur when representing shared state in that model. Section 4 presents domain and view abstractions as an extension to actors. In Section 5, a formal specification of our model is given by means of an operational semantics. Section 6 lists a number of important additional features of SHACL. The related work section and our conclusion complete the paper.

2. Communicating event-loops

Before we explain the issues with state sharing in message-passing concurrency models based on actors, we first provide more details about the precise nature of the actor model upon which SHACL is based.

The concurrency model of SHACL is based on the communicating event-loops model of E [3] and AmbientTalk [4] where actors are represented by *vats*. Each vat/actor has a single thread of execution, an object heap and an event queue. Generally,

[1] Pronounce as "shackle", short for **sh**ared **ac**tor **l**anguage.

[2] http://soft.vub.ac.be/~jdekoste/shacl.