Journal of Logical and Algebraic Methods in Programming 90 (2017) 158–175

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

On detecting over-eager concurrency in asynchronously communicating concurrent object systems [☆]

Olaf Owe [a,*], Charlie McDowell [b]

[a] University of Oslo, Norway
[b] University of California, Santa Cruz, USA

ARTICLE INFO

Article history:
Received 3 March 2016
Received in revised form 21 February 2017
Accepted 21 February 2017

Keywords:
Active objects
Concurrent objects
Asynchronous communication
Futures
Static analysis
Concurrency handling
Object flooding
Scheduling

ABSTRACT

Modern software systems are often distributed, and object-orientation is a leading paradigm for system modeling and design. We consider an object-oriented concurrency model for distributed systems, based on active objects, asynchronous method calls, and shared futures. This approach is appealing in that it gives rise to massive parallelism while avoiding active waiting and explicit locks, and has a simple semantics.

In this paper we show that systems developed using active objects and asynchronous method calls can result in system failure due to over-eager concurrency, which we call flooding. A system may feed an object with more calls than it is able to handle, in some cases even regardless of its processing speed. We refer to this situation as *flooding* of the object. We distinguish between *strong and weak flooding*. In particular, the notion of strong flooding could lead to problems such as non-responsive objects, system crash, overfull buffers or massive amounts of lost messages, even in the presence of fair scheduling. We present an algorithm to statically detect strong and weak flooding, and prove the soundness of the algorithm.

1. Introduction

Distributed Systems play an import role in modern society. Conceptually, a distributed system consists of a number of autonomous components that are connected by a network and are cooperating by means of message passing [5]. Thus concurrency is a key aspect of such systems. Components that run in parallel on different computers or on the same multi-threaded computer, may achieve efficient overall performance, provided they are not blocking each other. Modularity of the components is desirable, as this helps in understanding and designing complex systems, and thereby enhances scalability. In particular synchronization mechanisms should be modular, avoiding inter-component signaling/notification.

The Actor model [16,2,1] has been acknowledged as a natural way of programming distributed and concurrent systems, and is based on a compositional semantics, as opposed to the thread-based concurrency model. It unifies concurrency with the view of actors as autonomous units communicating asynchronously through messages. The Actor model has been adapted to the object-oriented setting in the form of active concurrent objects, interacting by means of remote method calls. The notion of so-called asynchronous methods are implemented by asynchronous message passing, while *suspension* may be used to achieve non-blocking calls [20]. Method interaction inherently supports the notion of request and reply, which

[☆] This work was done in the context of the *IoTSec* project funded by the Norwegian Research Council (`http://cwi.unik.no/wiki/IoTSec:Home`) (grant IKTPLUSS). A research exchange period at UCSC 2014/2015 was supported by University of Oslo.
[*] Corresponding author.
E-mail addresses: olaf@ifi.uio.no (O. Owe), mcdowell@ucsc.edu (C. McDowell).

http://dx.doi.org/10.1016/j.jlamp.2017.02.005
2352-2208/© 2017 Elsevier Inc. All rights reserved.

provides more control than pure message passing languages, and is therefore attractive for modeling of distributed systems [2,4].

Shared futures enable even more efficient interaction, allowing objects to share computation results without waiting for the results [31,14,25]. A future is a reference to a globally accessible location where the result of a given method call will be stored (once computed), generated by the caller object when the call is made. By allowing futures to be passed as parameters, and thereby shared, several objects may get access to the same future, reading the result held in the future or checking if such a result has appeared. A caller that does not need the result of a called method may pass the associated future to other objects without waiting for the result to appear. Futures are used in several languages [28,8,4].

A core language based on this concurrency model is given in Sec. 2. Inter-object concurrency comes for free in the sense that each object can run concurrently with other objects. Synchronization is handled in a modular manner without the use of external notification, by means of conditional suspension. This gives a high degree of autonomy and concurrency, as well as scalability; and high performance is possible due to asynchronous method calls, suspension, and shared futures.

However, this unrestricted concurrency model comes with a price. This programming style may give rise to deadlocks, and it is easy to create programs that are class-wise semantically correct but that fail due to over-eager creation of method calls. A system may feed an object with more calls than it is able to handle, in some cases even regardless of its processing speed. We refer to this situation as *flooding* of the object. Flooding is caused by execution cycles that give rise to unbounded amounts of uncompleted method calls, as discussed further in Sec. 3. Flooding may lead to non-responsive objects, system crash, overfull buffers or massive amounts of lost messages. Thus, flooding may seriously affect the performance of the whole system.

Our work is motivated by experimentation with this concurrency model using an implementation in Java, given as a Java library package hiding the underlying Java thread-based concurrency model [26]. We experienced executions where certain objects were not able to do anything due to an unbounded number of incoming requests, and were surprised by discovering several cases of this kind of flooding, resulting from small changes to sample programs, see illustrations in Sec. 3. This was not expected since the implementation was fair with respect to the underlying threads. Our investigation includes an understanding of the issue of flooding, its seriousness, and a way to detect cases of flooding.

In this paper we define and exemplify the concept of flooding. As flooding may depend on the underlying scheduling, we distinguish between strong and weak flooding. Strong flooding is defined as flooding under "favorable" scheduling, and weak flooding is flooding under weak scheduling assumptions. Thus strong flooding represents the more critical flooding situations. The scientific contribution of the paper is to present a static analysis method to detect possible weak and strong flooding (Sec. 4) by means of a terminating algorithm, and prove its soundness (no false negatives, Sec. 5). Static analysis of flooding cannot be both sound and complete (and terminating), since that would imply static control of termination of cycles depending on Boolean conditions. Due to over-approximation, detection of flooding may not imply a real flooding situation. However, when no flooding is detected, this implies that there is no real flooding situation (soundness). We discuss improvements for tighter detection of flooding, and in particular of strong flooding (Sec. 6) based on assumptions on the underlying process scheduling.

Related work. Fairness of the underlying process scheduling inside an object, as well as load balancing between objects, may eliminate weak flooding. Thus the topics of process scheduling and load balancing are complementary to the current work. Detection of flooding naturally relies on underlying assumptions about these factors. Detection of weak flooding is valuable for systems with weak underlying scheduling conditions, while strong flooding is valuable for systems with favorable scheduling conditions. Also deadlock analysis is complementary, since a dead object may flood in a trivial sense since it cannot complete any incoming calls. Our results on strong flooding will rely on assumptions about fair scheduling and absence of deadlock.

While analysis of deadlock situations, scheduling, and load balancing for this concurrency model have been investigated in several ways [7,15,17,22], we are not aware of analysis of object flooding for this or similar concurrency models. In particular static analysis of the suspension, non-blocking call, and future mechanisms are challenging. These mechanisms are essential for efficient high-level programming of concurrent object systems, and investigations of these are therefore valuable.

Arvind and Nikhil [3] recognized a problem of "excessive parallelism" in the context of the functional data flow language Id and tagged-token data flow. More recently, there have been efforts to address scheduling and fairness issues with active objects, but none of that work discusses the issue of system failure due to flooding. Instead, scheduling has been proposed to improve performance, and in some cases as an essential part of the correctness of the algorithm [27].

The fundamental work of Ganty and Majumdar [12] considers asynchronous programs and clarifies the complexity of static verification of safety and liveness properties, using a translation to Petri Nets. However, their asynchronous programs are different from the ones considered here, in that blocking is avoided by passing control from one unit to another, instead of using unit-local suspension as in our model of asynchronously communicating active objects. Another difference is that our study provides insight in the considered concurrency model, rather than reducing the problems to reachability questions embedded in some other formalism. In particular, the connection between weak and strong flooding and the notions of underlying scheduling and fairness relate to the considered concurrency model.

In thread-based systems, programs may generate an unbounded number of threads, for instance as a result of incoming requests. This problem is considered in [30], and a static algorithm for detecting unbounded thread-instantiation loops is