Flooding Detection in Concurrent Object Systems*

Charlie McDowell†
University of California, Santa Cruz, USA
mcdowell@ucsc.edu

Olaf Owe‡
University of Oslo, Norway
olaf@ifi.uio.no

Abstract

Concurrency is today an essential component of computer systems. One approach to programming concurrent object oriented systems is the use of active objects and asynchronous method calls, based on the actor model. This model is attractive by offering efficient programming and a simple, compositional semantics. The model facilitates independent units with a high degree of concurrency, but may also lead to deadlock. 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*. If an concurrent unit is flooded it is not able to respond properly. We present an algorithm to statically detect flooding, and we prove the soundness of the algorithm.

Keywords: active objects, concurrent objects, asynchronous communication, futures, static analysis, concurrency faults

1 Introduction

Concurrency is today a key aspect of the computer systems forming our infra-structure. This aspect is essential in distributed systems and net-based service systems such as cloud computing, as well as multi-core computers. Since it is easier to reduce parallelism than to increase the amount of parallelism, it is a non-trivial challenge to design systems that allow the desired amount of concurrency – and in a correct manner. In practice many systems rely on centralized control or synchronization of blocks of code to make programs dealing with shared data work correctly, including thread-based object-oriented concurrency, which is the most common paradigm used to program distributed systems today. However, synchronization restricts parallelism and slows down overall performance. While synchronization primitives for notification/signaling may improve efficiency, they are difficult to use correctly because they break modular reasoning and understanding.

The Actor model has been acknowledged as a natural way of programming concurrent systems, and is based on a simple semantics allowing modular reasoning [7, 2, 1]. It has been extended to the object-oriented setting in the form of active concurrent objects, interacting by means of remote method calls. Asynchronous methods increase efficiency by allowing non-blocking calls [9, 8]; and shared futures enable even more efficient interaction, allowing objects to share computation results without waiting for the results [13, 6, 10, 12, 4]. For instance a caller who does not need the result of the called method may pass the future identity of the result to other objects without (itself) waiting for the result to appear.

We consider a high-level core language based on this concurrency model. The language includes a mechanism for asynchronous call, suspension of the active process, and blocking and non-blocking primitives for obtaining future values, similar to [5]. Inter-object concurrency comes for free in the sense that each object can run concurrently with other objects. Intra-object synchronization is handled in a modular manner without the use of external notification. The concurrency model allows unrestricted

*This work was done in the context of the EU projects FP7-610582 *Envisage: Engineering Virtualized Services* (`http://www.envisage-project.eu`) and FP7-ICT-2013-X *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations* (`http://www.upscale-project.eu`).

†

‡

concurrency with a compositional semantics. Thus it enables efficient programming, class-wise understanding, verification, and testing. However, this unrestricted concurrency model does not come without 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, regardless of its processing speed. We refer to this situation as *flooding* of the object.

In this paper we define and exemplify the concept of flooding, distinguishing between strong and weak flooding. Weak flooding is less serious than strong flooding and can be managed with the use of fair scheduling of the processes within an object. Strong flooding may eventually overwhelm a system, even in the presence of fair scheduling. The scientific contribution of the paper is to propose a static analysis method to detect possible flooding situations, and prove its soundness (no false negatives). Since static analysis of flooding cannot be both sound and complete, 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).

While analysis of deadlock situations for this concurrency model has been investigated in several ways, we are not aware of analysis of object flooding for this concurrency model. Arvind and Nikhil [3] recognized a problem of "excessive parallelism" in the context of the functional dataflow language Id and tagged-token dataflow. 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 [11].

References

[1] G. Agha, S. Frolund, W.Y. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(2):3 –14, may 1993.

[2] Gul A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.

[3] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. on Computers*, 39(3):300–318, March 1990.

[4] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer-Verlag, 2004.

[5] Crystal C. Din and Olaf Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27:1–22, 2014.

[6] Robert H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[7] Carl E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[8] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, March 2007.

[9] Einar Broch Johnsen, Olaf Owe, and Marte Arnestad. Combining active and reactive behavior in concurrent objects. In *Proc. of the Norwegian Informatics Conference (NIK'03)*, pages 193–204. Tapir, November 2003.

[10] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In David S. Wise, editor, *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages 260–267. ACM Press, June 1988.

[11] Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. Programming and deployment of active objects with application-level scheduling. In *SAC'12*, pages 1883–1888. ACM, March 2012.

[12] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala. A comprehensive step-by-step guide*. Artima Developer, 2008.

[13] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Sigplan Notices*, 21(11):258–268, November 1986.