Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 58 (2005) 344–365

Science of Computer Programming

www.elsevier.com/locate/scico

Exclusion requirements and potential concurrency for composite objects

Abdelsalam Shanneb[a,*], John Potter[a], James Noble[b]

[a] *Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia*

[b] *School of Mathematical and Computing Sciences, Victoria University of Wellington, Wellington, New Zealand*

Received 1 November 2004; received in revised form 15 January 2005; accepted 1 March 2005
Available online 13 June 2005

Abstract

Concurrent object-oriented systems must prevent the interference that may arise when multiple threads simultaneously access shared components. We present a simple approach for implementing flexible locking strategies in an object-oriented system, in which the components themselves may be *composite objects*. We express *exclusion requirements* as sets of conflict pairs on component interfaces. Given knowledge of the dependency between the interface of a composite object and its internal components, we show how external exclusion requirements can be calculated from internal requirements, and further, how any potential concurrent activity outside an object can be projected into *potential concurrency* for the internal components.

With our approach we can defer the distribution of locks in the system until deployment: the placement of locks and choice of lock type for a component can depend on its operating environment. A Galois connection between the outward mapping of exclusion requirements, and the inward mapping of potential concurrency, limits how many locks are worth considering. In this paper we only deal with exclusion control, including mutexes, read–write locks and read–write sets, and do not cover state-dependent locking or transaction-based approaches.

* Corresponding author.
E-mail addresses: shanneba@cse.unsw.edu.au (A. Shanneb), potter@cse.unsw.edu.au (J. Potter), kjx@mcs.vuw.ac.nz (J. Noble).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.03.004

Keywords: Concurrency control; Concurrent objects; Composite objects; Component-based systems; Locking granularity

1. Introduction

Whether by education, experience, or accident, most programmers habitually treat programs as sequential. When presented with a series of statements in almost any programming language, we are drawn to imagining the effect of executing these statements one step after another. Our assumptions about the correctness of code depend critically on this sequentiality. Unfortunately, our intuition does not model the execution of concurrent programs. In a multi-threaded environment, concurrent threads sharing resources can interfere with each other's execution.

To guarantee *thread-safety* for our systems, we need to prevent interference between concurrent threads potentially operating on the same data. In order to provide thread-safety for software components, the simplest approach is to force mutually exclusive access to the components' interface. For example, in Java, we can declare all the methods of a class as synchronised, serialising concurrent calls to each instance of that class so that each object acts as a monitor. COM's apartment model similarly allows entire components to be singly threaded. Single-threading entire high-level components or subsystems necessarily limits concurrent execution, thereby restricting system responsiveness and efficiency in multiprocessor environments.

To increase the potential concurrency in a system while maintaining thread-safety, we can adopt two complementary approaches. First, we can move monitor boundaries from high-level components down to subcomponents, so that rather than single-threading an entire subsystem, only the shared objects within that subsystem are single-threaded. Second, we can adopt a finer granularity of exclusion control, such as read–write locks, rather than simply single-threading entire components. We can of course adopt both of these approaches simultaneously, and provide finer grain locking internally rather than at the external interface.

This article contributes a novel approach for reasoning about concurrency and exclusion in component-based object-oriented systems. We provide a simple notation for recording the exclusion requirements of each component in a system. Programmers can associate fine-grained exclusion policies with any object in the composition. Then, using dependency relations between composite and subsidiary components, we show how to propagate internal exclusion requirements outward, and the potential for concurrency inward, thereby checking that all components exclusion requirements have been met, ensuring that the system as a whole will be thread-safe. Our notation only addresses *exclusion control*; this includes mutexes, read–write locks and read–write sets, but does not cover state-dependent locking or transaction-based approaches.

This work extends our earlier work on the *algebra of exclusion* [26,28], by introducing an explicit notion of *potential concurrency* that is complementary to exclusion. The earlier approach was unable to calculate the exclusion that a composite component provided for its subcomponents: rather, it relied on programmers guessing the exclusion first, and