Science of Computer Programming 163 (2018) 174–189

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Specification and verification of synchronization with condition variables

Pedro de C. Gomes[a,*], Dilian Gurov[a], Marieke Huisman[b,1], Cyrille Artho[a]

[a] KTH Royal Institute of Technology, Stockholm, Sweden
[b] University of Twente, Enschede, the Netherlands

A R T I C L E   I N F O

Article history:
Received 17 May 2017
Received in revised form 2 May 2018
Accepted 2 May 2018
Available online 8 May 2018

Keywords:
Concurrency
Formal verification
Java
Condition variables

A B S T R A C T

This paper proposes a technique to specify and verify the *correct synchronization* of concurrent programs with condition variables. We define correctness of synchronization as the liveness property: "every thread synchronizing under a set of condition variables eventually exits the synchronization block", under the assumption that every such thread eventually reaches its synchronization block. Our technique does not avoid the combinatorial explosion of interleavings of thread behaviours. Instead, we alleviate it by abstracting away all details that are irrelevant to the *synchronization behaviour* of the program, which is typically significantly smaller than its overall behaviour. First, we introduce SyncTask, a simple imperative language to specify parallel computations that synchronize via condition variables. We consider a SyncTask program to have a correct synchronization iff it terminates. Further, to relieve the programmer from the burden of providing specifications in SyncTask, we introduce an economic annotation scheme for Java programs to assist the *automated extraction* of SyncTask programs capturing the synchronization behaviour of the underlying program. We show that every Java program annotated according to the scheme (and satisfying the assumption mentioned above) has a correct synchronization iff its corresponding SyncTask program terminates. We then show how to transform the verification of termination of the SyncTask program into a standard reachability problem over Coloured Petri Nets that is efficiently solvable by existing Petri Net analysis tools. Both the SyncTask program extraction and the generation of Petri Nets are implemented in our STaVe tool. We evaluate the proposed framework on a number of test cases.

1. Introduction

Condition variables in concurrent programs. Condition variables (CV) are a commonly used synchronization mechanism to coordinate multithreaded programs. Threads *wait* on a CV, meaning they suspend their execution until another thread *notifies* the CV, causing the waiting threads to resume their execution. The signalling is asynchronous: the effect of the notification can be delayed. If no thread is waiting on the CV, then the notification has no effect. CVs are used in conjunction with

* Corresponding author.
E-mail addresses: pedrodcg@kth.se (P. de C. Gomes), dilian@kth.se (D. Gurov), m.huisman@utwente.nl (M. Huisman), artho@kth.se (C. Artho).
[1] Supported by ERC grant 258405 for the VerCors project.

https://doi.org/10.1016/j.scico.2018.05.001
0167-6423/© 2018 Elsevier B.V. All rights reserved.

```
01 class Utilizer extends Thread {        class Provider extends Thread {
     synchronized(lock) {               09  synchronized(lock) {
03    while (!resource_available) {           // prepare resource
       lock.wait();                     11    resource_available = true;
05    }                                       lock.notify();
     }                                  13  }
07 }                                        }
```

Fig. 1. A simple Java program using `wait`/`notify`.

locks; a thread must have acquired the associated lock for notifying or waiting on a CV, and if notified, must reacquire the lock.

Many widely used programming languages feature condition variables. In Java, for instance, they are provided both natively as an object's *monitor* [1], i.e., a pair of a lock and a CV, and in the `java.util.concurrent` library, as one-to-many `Condition` objects associated to a `Lock` object. C/C++ have similar mechanisms provided by the POSIX thread (Pthread) library, and C++ features CVs natively since 2011 [2] as the `std::condition_variable` class. The mechanism is typically employed when the progress of threads depends on the state of a shared variable, to avoid busy-wait loops that poll the state of this shared variable.

Example 1 *(Condition variables in Java).* Fig. 1 shows a simple example with two threads: The first thread, *Utilizer*, wants to use a shared resource. The resource is guarded with a common lock (line 2) to ensure that only one thread, the lock holder, can change the state of the resource. Because no high-level constructs like `await(resource_available)` exist in Java, the *Utilizer* thread has to check if the condition holds by using a conditional statement (line 3). If the condition is false, the *Utilizer* suspends itself by calling `wait` in line 4. This call implicitly relinquishes the lock, to allow another thread to access it and modify the condition variable. At some point, another thread may make the resource available. That thread then has to signal the state change to the condition variable. In our example, thread *Provider* uses the same lock to access the shared variable, and calls `notify` to signal a change in line 12.

As a result of that signal, one of the waiting threads is woken up. It has first to re-check the condition, since it might have been re-invalidated by another thread in the meantime. To do this, the lock is (implicitly) re-acquired. In case another thread has already consumed the resource, and `resource_available` is again *false*, the `while` loop in line 3 is re-entered. Otherwise, the waiting thread may proceed under the assumption that `resource_available` is *true*. This assumption holds if all accesses to the shared condition variable are protected by a common lock, i.e., if the whole program is data race free.

The `notify` method wakes up any one thread that is waiting at the time the notification is sent; there is no mechanism to ensure that a particular thread gets woken up. If multiple waiting threads may check or use shared conditions in different ways (for example, by using a function over multiple shared variables), the notifying thread should call `notifyAll`, to ensure each waiting thread gets woken up once and can re-check the condition variable to see if the "right" condition is true.

Waiting threads may get interrupted in real Java programs, so they have to guard any call to `wait` with a `try`/`catch` block, to catch an `InterruptedException`. Furthermore, the Java Specification [3, § 17.2] permits (but discourages) JVM implementations to perform spurious wake-ups, and reinforces the coding practice of invoking `wait` inside loops guarded by a logical condition necessary for thread progress. We elide these functionalities in our paper.

Writing correct programs using condition variables is challenging, mainly because of the complexity of reasoning about asynchronous signalling. Nevertheless, condition variables have not been addressed sufficiently with formal techniques, to no small part due to this complexity. For instance, Leino et al. [4] acknowledge that verifying the absence of deadlocks when using CVs is hard because a notification is "lost" if no thread is waiting on it. Thus, one cannot verify locally whether a waiting thread will eventually be notified. Furthermore, the synchronization conditions can be quite complex, involving both control-flow and data-flow aspects as arising from method calls; their correctness thus depends on the *global thread composition*, i.e., the type and number of parallel threads. All these complexities suggest the need for *programmer-provided annotations* to assist the automated analysis, which is the approach we are following here.

In this work, we present a formal technique for specifying and verifying that "every thread synchronizing under a set of condition variables eventually exits the synchronization", under the assumption that every such thread eventually reaches its synchronization block. The assumption itself is not addressed here, as it does not pertain to correctness of the synchronization, and there already exist techniques for dealing with such properties (see, e.g., [5]). Note that the above correctness notion applies to a *one-time synchronization* on a condition variable only; generalizing the notion to repeated synchronizations is left for future work. To the best of our knowledge, the present work is the first to address a *liveness* property involving CVs. As the verification of such properties is undecidable in general, we limit our technique to programs with bounded data domains and a bounded number of threads. Still, the verification problem is subject to a combinatorial explosion of thread interleavings. Our technique alleviates the state space explosion problem by *delimiting the relevant aspects of the synchronization*.