Title Text

Subtitle Text, if any

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

The text of the paper begins here.

Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.
More text.
Lots of text.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

More text.
Lots of text.
More text.
Lots of text.

A. Comparation

Most researchers focus on resource deadlock which is caused by circle locks, while not too much attention was paid on wait-notify type deadlocks. One of the related work is checkMate[1] which could detect potential wait-notify deadlocks from a test running which does not trigger a bug. Checkmate first observes a run of the original program and records the synchronization and relative condition semantics, e.g. synchronized, wait, notify and conditions for wait and notify. Checkmate then generates a trace program that removes all the business logics from original program and remains only the statements recorded at previous step. At last, by model checking the trace program with off-the-shell tool, checkmate reports potential wait-notify deadlocks. We choose 5 wait-notify deadlocks caused by different reasons from JaConTeBe[2] benchmark at SIR[3] to compare the performance of our approach and checkMate. The result is shown in table 1. Pool146 is from Apache pool 1.5, and its bug id in Apache pools Bugzilla repository is 146. This bug is caused by the improper usage of queue. When the thread related with the head element of the queue is blocked, the head element shall not be dequeued, and other threads that will notify the blocked thread have to wait because they cant access the required elements in the queue. Pool149 is from Apache pool 1.5, and its bug id in Apache pools Bugzilla repository is 149. This bug is triggered by a certain schedule. The wait and notify threads need to acquire the same lock, so when notify thread obtains the lock first, the notify will come before the wait, and the wait thread has to wait forever. Pool162 is from Apache pool 1.5, and its bug id in Apache pools Bugzilla repository is 162. This bug is caused by an unhandled

exception which leads to a memory leak and set the notify condition unsatisfied. The notify thread shall not reach the notify statement, and the wait thread will wait forever. Log4j38137 is from Apache log4j 1.2.13, and its bug id in Apache log4js Bugzilla repository is 38137. A certain schedule in this bug makes the notify condition unsatisfied, and all threads get into wait. Log4j50436 is from Apache log4j 1.2.14, and its bug id in Apache log4js Bugzilla repository is 50436. This bug is caused by an unhandled exception which leads to the dead of the notify thread. CheckMate could shrink the original program dramatically to make it efficient to be model checked, but it suffers several drawbacks which fail it from detecting all the deadlock bugs in table 1. 1. CheckMate heavily depends on the running schedule it observes. If a program has M schedules, checkMate has to observe each of them to generate trace programs for model checking all the possible deadlocks. The problem is, its not trivial to observe all M schedules of original program. So it does not mean there is no deadlock when checkMate reports so. 2. Its complicated and error-prone to insert condition control phrases into real world programs. CheckMate requires to insert condition control code into the original program before conditional wait/notify manually so the model checker can get into branches that are not reached in the observed execution. However, in the real world program, its complicated and error-prone to do so. There might be multiple exits for the conditional branch, e.g. break, continue, throw, catch and return. If the closure of inserted condition control code is missed before any of these exits, the generated trace program shall be wrong. 3. Assumptions for checkMates optimization does not always hold. CheckMate assumes the local variables do not get involved in deadlocks, so its safe to remove the synchronized blocks that monitor on local variables. This assumption does not always hold in real world program, because some local variables may refer to shared variables. Simply optimize local variables may lead to missing deadlocks. 4. CheckMate does not consider the unhandled exceptions. Unhandled exceptions may terminate notify thread or set notify condition unsatisfied and cause deadlock. But checkMate does not extend its vision to the field of exception handling, which may also miss some deadlocks. 5. CheckMate does not considered the daemon thread. Daemon threads in Java do not prevent the whole process terminating. Even a daemon thread is waiting, as long as other non-daemon threads terminate, the whole program terminates. But checkMates algorithm does not check if a thread is daemon. This means checkMate could report false positives.

Acknowledgments

Acknowledgments, if needed.

References

[1] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM, 2010.