Diss. ETH No. 16020

Combining Static and Dynamic Analysis to Find Multi-threading Faults Beyond Data Races

A dissertation submitted to the
Swiss Federal Institute of Technology Zurich
ETH Zürich

for the degree of
Doctor of Technical Sciences

presented by
Cyrille Artho
Dipl. Informatik-Ing. ETH
born June 17, 1976
citizen of St. Gallenkappel SG,
Switzerland

accepted on the recommendation of
Prof. Dr. Armin Biere, examiner
Prof. Thomas Gross, co-examiner
Dr. Klaus Havelund, co-examiner
Prof. Dr. Doron Peled, co-examiner

April 2005

Abstract

Multi-threaded programming gives rise to errors that do not occur in sequential programs. Such errors are hard to find using traditional testing. In this context, verification of the locking and data access discipline of a program is very promising, as it finds many kinds of errors quickly, without requiring a user-defined specification.

Run-time verification utilizes such rules in order to detect possible failures, which do not have to actually occur in a given program execution. A common such failure is a data race, which results from inadequate synchronization between threads during access to shared data. Data races do not always result in a visible failure and are thus hard to detect. Traditional data races denote direct accesses to shared data. In addition to this, a new kind of high-level data race is introduced, where accesses to sets of data are not protected consistently. Such inconsistencies can lead to further failures that cannot be detected by other algorithms. Finally, data races leave other errors untouched which concern atomicity. Atomicity relates to sequences of actions that have to be executed atomically, with no other thread changing the global program state such that the outcome differs from serial execution. A data-flow-based approach is presented here, which detects stale values, where local copies of data are outdated.

The latter property can be analyzed efficiently using static analysis. In order to allow for comparison between static and dynamic analysis, a new kind of generic analysis has been implemented in the JNuke framework presented here. This generic analysis can utilize the same algorithm in both a static and dynamic setting. By abstracting differences between the two scenarios into a corresponding environment, common structures such as analysis logics and context can be used twofold. The architecture and other implementation aspects of JNuke are also described in this work.

Kurzfassung

Programme mit mehreren Threads erlauben Fehler, die nicht in sequentiellen Programmen vorkommen. Solche Fehler sind mit traditionellem Testen schwer zu finden. Darum ist die Überprüfung der Locking- und Datenzugriffsdisziplin eines Programms sehr vielversprechend, da es viele Arten von Fehlern schnell findet, ohne eine benutzerdefinierte Spezifikation zu benötigen.

Run-time Verification benutzt solche Regeln, um mögliche Fehler zu finden, ohne dass diese tatsächlich in einer gegebenen Programmausführung auftreten müssen. Ein häufiger solcher Fehler ist ein Date Race, das aus unzureichender Synchronisation zwischen mehreren Threads mit gemeinsamen Daten resultiert. Data Races resultieren nicht immer in einem sichtbaren Fehler und sind darum schwer zu entdecken. Traditionelle Data Races beziehen sich auf direkte Zugriffe zu gemeinsamen Daten. Darüberhinaus wird eine neue Art von Data Race eingeführt, wo Mengen von Daten nicht konsistent geschützt sind. Solche Inkonsistenzen können zu weiteren Fehlern führen, welche nicht durch andere Algorithmen entdeckt werden können. Schlussendlich lassen Data Races andere Fehler ausser acht, welche die Atomicity betreffen. Atomicity bezieht sich auf Sequenzen von Aktionen, die atomar ausgeführt werden müssen, ohne dass ein anderer Thread den globalen Zustand auf eine Art ändert, so dass das Ergebnis einer Operation sich von serieller Ausführung unterscheidet. Ein data-flow-basierter Ansatz wird hier präsentiert, welcher Stale Values entdeckt, wo lokale Kopien von Daten veraltet sind.

Letztere Eigenschaft kann effizient statisch analysiert werden. Um einen Vergleich zwischen statischer und dynamischer Analyse zu erlauben, wurde im JNuke-Framework, das hier präsentiert wird, eine neue Art von generischer Analyse implementiert. Diese generische Analyse kann denselben Algorithmus in einem statischen oder dynamischen Umfeld benutzen. Indem man die Unterscheide zwischen den beiden Szenarien in ein entsprechendes Environment abstrahiert, können gemeinsame Strukturen wie die Analyse-Logik und der Kontekt doppelt genutzt werden. Die Architektur und andere Implementationsaspekte von JNuke werden auch in dieser Arbeit beschrieben.