AN ARCHITECTURE FOR SELF-HEALING OF DATA RACES AND ATOMICITY VIOLATIONS FOR JAVA

Zdeněk Letko
Master Degree Programme (3), FIT BUT
E-mail: xletko00@stud.fit.vutbr.cz

Supervised by: Tomáš Vojnar
E-mail: vojnar@fit.vutbr.cz

ABSTRACT

Data races and atomicity violation are a common problem in concurrent programming. This article describes a technology capable to detect atomicity violation and data races in Java programs and heal them at run-time. The architecture expects dynamic analysis to be used for detecting and healing data races and atomicity violations. Correct atomicity can be specified manually or obtained by static analysis.

1 INTRODUCTION

Data races and atomicity violation are difficult to find using traditional testing approaches because a multi-threaded program may execute differently from one run to another and problems may appear in a very rare situations only. This article describes an architecture and a prototype tool which uses dynamic analysis to detect and heal data races and atomicity violations at run-time. The combination of static and dynamic analysis is used for obtaining correct atomicity. The prototype is implemented on top of IBM ConTest, a concurrency testing tool, and uses FindBugs for static analysis of Java bytecode.

2 DATA RACES AND ATOMICITY VIOLATION

The traditional definition of a data race [1] is as follows: A data race occurs when two concurrent threads access a shared variable, and at least one of the accesses is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous. Therefore the value of the variable can be unpredictable and potentially wrong.

Atomicity is a property of a block of code. A block of code is atomic if it is serializable, i.e., if the effect of the execution of the block is the same regardless of the way it is executed in a concurrent environment with other blocks of code accessing the same data and running in parallel to it.

Usually, what developers want is atomicity, not freedom from data races [2]. Being data race-free does not necessarily indicate a correct synchronization. Many concurrency bugs still occur even when each access to each shared variable is protected by a lock. On the other hand, a data race is not always a bug. Many important synchronization mechanisms are actually

implemented in a way tolerating data races. The most popular are flag synchronization, barriers and producer-consumer queues.

3 DATA RACE AND ATOMICITY VIOLATION SELF-HEALING

Self-healing is a new approach based on the idea of a program which can automatically detect and heal its bugs [1]. The main idea can be described as follows. The program will watch its behavior and gather some other information related to the correctness of its execution. If a wrong behavior is detected, the program itself tries to change its activities or the way how they are done with the intention to avoid the problem.

Healing of data races or atomicity violations can be seen as removing them from the program. Commonly, data races are thought to be removed by using some synchronization mechanism. It really helps because we introduce an explicit mechanism which prevents the accesses from being simultaneous. But this solution has also two drawbacks.

First, we have removed the data races but usually not atomicity violation. This can be eliminated by introducing synchronization mechanism which follows the correct atomicity. Second, we can cause a possibly worse problem than a data race, e.g. a deadlock. This can be avoided by a further analysis which is able to answer the question if it is safe to heal the problem in the chosen way.

4 AN OVERVIEW OF THE PROPOSED ARCHITECTURE

The proposed architecture is depicted in Figure 1. The main part of the architecture consists of three modules. The *execution monitoring* module watches the program and triggers predefined events during the execution. Additional information describing the event are collected and the event is passed to the analysis engine. The *analysis engine* uses a suitable detection algorithm to decide if the event is legal for the predefined rules of a correct behavior of the program. The problem has been detected if some rule is violated. Finally, the *healing logic* can influence the behavior of the program to prevent the problem manifestation. Lets look at the modules in more detail.

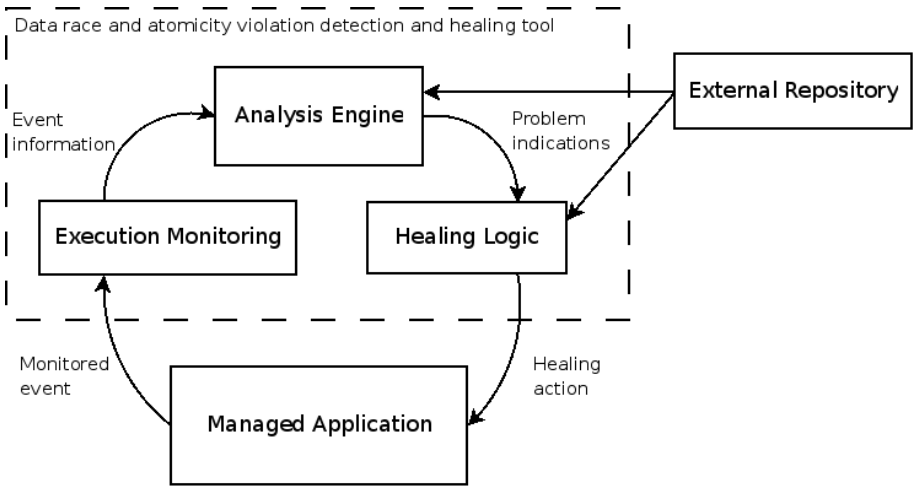

Figure 1: Architecture diagram.

The core of the execution monitoring and influencing is done by the IBM ConTest tool. It uses instrumentation to insert its code into the managed bytecode. This approach enables ConTest to track and influence the execution of the program. Some basic information such as the program location describing the position of the execution in the code, the active thread and the current operation are provided by ConTest. It also provides a listeners architecture which allows self-healing actions to take part in the execution of the program.
Analysis engine uses the stream of events provided by the monitoring module to analyze the behavior of the program. Two detection algorithms have been implemented sofar. The first is based on the well known lockset algorithm which detects race conditions by tracking the locking policy for each shared variable. An advantage of this algorithm is that it does not need any precomputed information, but on the other hand, it suffers from a higher number of false alarms. The other algorithm detects violations in a predefined atomicity of program blocks acquired by a static analysis and available from the external repository. An advantage of this algorithm is that it does not produce false alarms if a correct description of the needed atomicity of the program is available. The disadvantage is that it needs a definition of the expected atomicity of the program constructions to be available in advance. This can be done using static analysis or manually what could be costly.
Static analysis can be used in two ways. First, it is used to obtain an initial set of atomicity for the program which is then pruned by the dynamic and statistical analysis so it corresponds to the correct atomicity of the used program constructions. Another way is to statically detects a patterns which are known to be atomic. Such set of atomicity is smaller but does not need any further processing.
The healing logic module controls the influencing of the execution. It can be done by safe but not very efficient influencing of Java scheduler or by more effective but potentially dangerous adding a new synchronization lock. As was written in the previous section the healing has to follow the atomicity of the program.

5 CONCLUSIONS

This paper presents an architecture for self-healing. The prototype of the architecture has been implemented to prove the concept. The main subject of the future work is the detection of blocks expected to be executed atomically.

ACKNOWLEDGMENTS

This work is partially supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD – project SHADOWS contract IST-035157.

REFERENCES

[1] Bohuslav Křena, Zdeněk Letko, Rachel Tzoref, Shmuel Ur, and Tomáš Vojnar. Healing data races on-the-fly. In *PADTAD '07*, pages 54–64. ACM, 2007.

[2] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII*, pages 37–48. ACM, 2006.