# HEALING ASSURANCE IN JAVA PROGRAMS

**Pavel Vyvial**
Bachelor Degree Programme (1), FIT BUT
E-mail: xvyvia00@stud.fit.vutbr.cz

Supervised by: Bohuslav Křena
E-mail: krena@fit.vutbr.cz

## ABSTRACT

The project SHADOWS has started research which is developing software for automatic healing of concurrent bugs. We work with self-healing software, which looks for concurrent bugs. If the detection software finds a bug, the healing action will be performed. After every healing action, one would like to know whether this action has fixed the detected problem and, perhaps even more importantly, that it has not caused any other, possibly even more serious, problem.Therefore this paper describes a technique which gives the answer for this question after automatical healing. The prototype uses static analysis for this purpose.

## 1 INTRODUCTION

Concurrent programming brings with a several advantages (like efficient usage of high performance computers) possibility of new types of bugs. The programs which programmers created for single-CPU-core processors are now exhibiting problems. It is very difficult to find all concurrent bugs in programs due to the nature of concurrent programs – there is a huge number of possible interleavings. Concurrent bugs often remain in software. This is strong motivation for developing software with self-healing techniques.

In general, a self-healing approach consists of the following steps:

- problem detection – it is necessary to detect that somethig is wrong with the system
- problem localisation – finding the root cause of detected problem
- problem healing – fix the found problem
- healing assurance – one has to give result that healing action was success or not

The detection and automatic healing of bugs in Java programs is a complicated problem. Currently, we deal with data races that can be automatically healed by adding new locks. Wrong synchronization using locks, caused by healing action for instance, can lead to a deadlock. Within healing assurance we concentrate on deadlock detection.

## 2 BACKGROUND

A data race in concurrent system occurs when multiple threads access a shared memory location without proper synchronization and at least one of these acceses is a write access. Data races can lead to an unpredictable behaviour of program, therefore are usually considered as bug. Data race are hard to handle because they are non-deterministic.
For data races detection, static and dynamic analysis is used. A data race condition can be created while if is not. When programmer thinks that some part of concurrent code will be executed atomically. A block of code is atomic if for every interleaved execution of the program in which the block is executed, there is an equivalent run of the program where the block is executed sequentially (without interleaving with other threads). If programmer makes any possibly data race bug, which is based on bad conception of code atomicity, we can heal it by introducing additional locks[1].

## 3 FORMAL VERIFICATION

Algorithm of healing assurance which is described in this paper is based on analysis which finds every lock in the system and then checks whether the lock added by a healing action does not interract with other locks.
Formal verification is one of possibilities which is exploited to the verification of correctness system. It is better than inspection, simulation, or testing system for our analysis because it can be done fully automatically. This technique can give the answer that analysed system is correct. Formal verification is able to mathematically prove that analysed system is correct.

*In this paper we say that a system is correct if no deadlock is feasible from it is initial state.*

Formal verification can be performed by: theorem proving, model checking, or static analysis. Static analysis tries to get the most of information from source code without its execution. Disadvatage of this technique is that it reports false alarms (bugs which do not exist). But, if our analysis didn't find a lock it is true at all times. One part of static analysis is finding a bug patterns in program. We use such an approach in proposed analysis.
Our goal is to create a detector which will be able to find locks (in order to avoid deadlocks) in code. For this purpose we use *FindBugs*[2]. It is open source tool used for static analysis to find bugs in Java code via bug patterns. FindBugs works at byte-code level and abstraction of it created by *Byte Code Engineering Library (BCEL)*[3]. Tool has been successful used in projects like Javabeans, Swing Eclipse, Ant, SCA.

## 4 HEALING ASSURANCE BY STATIC ANALYSIS

Currently, self-healing action can introduce a new lock which ensures the atomicity of potentially dangerous unsynchronized code. Unfortunatelly, such a lock can cause new more dangerous bugs (e.g., a deadlock).
Checking of program correctness is done by *static analysis* and *FindBugs* tool. It is about establish if the lock which was inserted to a program can or can not make a deadlock, which comes from the idea, that the use of a new lock can not lead to a deadlock, if there is no other locking command within locked section of program code. Imporatant is the sequence of locking

and if a healing lock create loop in relation graph. Finding locks is provided by FindBugs tool under BCEL abstraction.
In practice, the *healing assurance programme (HAP)* takes the location of potential danger (characterised by name of method, class and line numbers) and tries to find there a potentially danger synchronization (e.g. locking). Healing assurance programme then takes a byte-code of methods from classes bring analysed and starts to search the interesting actions in it. If HAP finds a method call it marks the signature of the called method and the calling location for further investigation. Gathered information about method calls creates a *Call graph*. Healing assurance programme then get the scanning place (class, method and line numbers which represents atomic section created by healing lock) where it starts searching for lock. If HAP does not find a lock action in the given position it will start searching for locks over methods in a lower level of *Call graph* and are accesible from scanning location. The steps of searching for a lock actions are repeated until whole Control graph is scanned. If HAP do not find lock action, healing action is marked as save, otherwise the healing is considered as dangerous. Danger healing can not be used in a program, thus healing action by introducing a new lock is not taken. If HAP finds any method which it can not analyse, it marks them as potential danger.

## 5 CONCLUSIONS

In this paper, we have presented healing assurance in Java programs by static analysis. Currently, healing assurance programme is in the phase of finding locks and creating the Call graph over all methods of analysed program. The next step is to reduce the analysis of methods from the Call graph which are accesible from a location where planned the healing lock is to be used.

## ACKNOWLEDGEMENT

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. The authors are solely responsible for the content of this paper. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

## REFERENCE

[1] B. Křena et al.: Healing data races on-the-fly, ACM 2007 – PADTAD '07, s. 54-64, ISBN 978-1-59593-748-3

[2] FindBugs – a program which uses static analysis to look for bugs in Java code [online]. Last update 2008-02-22. Available URL: <http://findbugs.sourceforge.net/>

[3] BCEL – Byte Code Engineering Library [online]. Last update 2006-01-03. Available URL: <http://jakarta.apache.org/bcel/>