ACCELERATION OF HEALING ASSURANCE OF CONCURRENT JAVA PROGRAMS

Pavel Vyvial
Master 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. After every healing action, one would like to know whether this action has fixed the detected problem and that it has not caused any other, possibly even more serious, problem. This paper describes a technique which accelerates healing assurance. Because stable library could be use in lots of healing assurance analysis, acceleration of healing assurance of concurrent Java programs by library preprocessing based on script was created.

1. INTRODUCTION

Concurrent programming brings with a several advantages (like efficient usage of high performance computers) possibility of new types of bugs. 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 interleaving.

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

- problem detection – it is necessary to detect that something is wrong
- problem localisation – finding the root cause of detected problem
- problem healing – fix the found problem
- healing assurance – give result if healing action was successful or not

Data races can lead to an unpredictable behaviour of program, therefore are usually considered as bug. For data races detection, static and dynamic analysis is used. Currently, we deal with data races that can be automatically healed by adding new locks or by rescheduling. Within healing assurance, we concentrate on deadlock detection [1,2].

Currently, self-healing action can introduce a new lock which ensures the atomicity of potentially dangerous unsynchronized code. Unfortunately, such a lock can cause a deadlock. This paper describe acceleration for healing assurance after healing action based on adding new locks.

For healing assurance, we use two approaches. First technique is the strategy of recording the trace of program execution and on replaying it in the model checker − JavaPath Finder [3]. Some healing assurance purpose (as finding lock instruction in particular code block) do not need so robust technique and this is reason why we use second quicker technique − healing assurance by static analysis. Healing assurance based on static analysis (HABOSA) is less accurate then healing assurance based on model checking but if HABOSA tells that some healing action is safe it is true.

HABOSA is detector which is able to find locks (in order to avoid deadlocks) in code. For this purpose being used FindBugs [4]. FindBugs works at byte-code level and abstraction of it created by Byte Code Engineering Library (BCEL) [5].

Healing assurance is done by static analysis. Important is the sequence of locking and if a healing lock create loop in relation graph. Finding locks is provided by FindBugs under BCEL abstraction, Java byte-code which is instrumented by ConTest [6] and output of Python script (safeMethods file and dangerMethods file).

Acceleration is created for HABOSA which finds every lock in the system and then checks whether the lock added by a healing action does not interact with other locks.

2. LIBRARY PREPROCESSING FOR HEALING ASSURANCE PROGRAM

Because the first version of Healing Assurance Program (HAP) [7] can not find some analysed methods HAP marks a lot of methods as potential danger. Some of these methods which first HAP prototype marked as potential danger are safe. Often these methods include Java libraries or other libraries which are user available. We have two choices how to make less false alarm (mark potential danger methods). We take big libraries and instrument it by ConTest and start analyse. Advantage of this method is that user does not need more tools. Disadvantage of this technique is that analyse have to make huge Call Graph under all instrumented methods which user puts at HAP input. This technique is really slow for big libraries like Java library at HAP input. This is reason why was created second technique based on script (SK), which helps use smaller Call Graph then first technique.

If a user uses a library often, it is better to process the library (by executing SK) only once and so accelerate all the following usages of the library. SK is able to classify methods of the given library into two groups. First group include method's signature of methods which can be in location heal by adding lock (stored in the safeMethods file). Second group include method's signature of methods which can not be in location heal by adding lock (stored in dangerMethods file). This classification helps HAP accelerate healing assurance.

In practice, SK takes the classes which are in current directory or subdirectories and create text files by force of Javap. Then SK start with classes analyse. One start looks safe and danger methods. Danger methods are every method which have at least one of this patterns: (1) Synchronized or native flag are at method declaration. (2) Method's class has ancestor's class which has danger method with same signature. (3) Method's body include monitorenter instruction. (4) Method's body include invoke instruction which destination is some danger or unknown method. All the others methods are safe. When script analyse every method from classes which are at current directory or subdirectories SK create output files. This analysis could run long time for huge library but it is better spend this time once and after that analyse by HAP have faster. Table 1 shows some test statistics. Test machine has configuration: 8 core based on CPU Intel X5355 2.66GHz and 16GB RAM.

name	classes	methods	safe methods	danger methods	time
java library	17 307	133 981	61 746	72 235	69 hours
java.*	2 449	20 990	9 739	11 251	1 hour

Table 1: SK runs for all java libraries and java.*

3. CONCLUSIONS

In this paper, we have presented technique which accelerates healing assurance in Java programs by static analysis. The main part acceleration is build on script which is tackling problem with inaccessible methods from user's available libraries. Currently, script creates safeMethods file and dangerMethods file that accelerate HAP by reducing Call Graph. Healing assurance program uses script output and finds locks and creates the Call graph over all methods of analysed program. HAP analyses a location where healing action has planned new lock and decides about safety of healing action.

ACKNOWLEDGEMENT

This work is 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.

REFERENCES

[1] B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In Proceedings of the PADTAD '07. ACM, 2007.

[2] B. Křena, Z. Letko, and T.Vojnar. AtomRace: Data Race and Atomicity Violation Detector and Healer. In Proceedings of the PADTAD'08. ACM, 2008.

[3] B. Křena, V. Hrubá, and T. Vojnar. Self–healing Assurance Based on Bounded Model Checking. In Proceedings of the 12th International Workshop on computer Aided Systems Theory. Las Palamas de Gran Canaria, Spain, February 2009.

[4] 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/>

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

[6] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded Java program test generation. IBM Systems Journal, 41(1):111–125, 2002.

[7] P. Vyvial. Healing assurance in java programs. In Proceedings of the 14th conference Student EEICT 2008, volume 1. Vysoké učení technické v Brně, 2008.