BOUNDED MODEL CHECKING USING JAVA PATHFINDER

Vendula Hrubá
Master Degree Programme (2), FIT BUT
E-mail: xhruba03@stud.fit.vutbr.cz

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

ABSTRACT

This work describes the using of bounded model checking for verification of the true races in programs. The model checking of a real system is costly, thus there are some modification or alternations of model checking of features. This paper describes the search strategy for replaying a trace and for navigating through a state space to a suspicious state and a subsequent bounded model checking initiated from this state. The bounded model checking is implemented by the model checker Java Pathfinder.

1 INTRODUCTION

Software applications are almost in every profession at this time. There is systems which have to be safe, but nearly every system contains bugs. There are proposed a self-healing methodology for concurrency related problems in European project SHADOWS, however, the self-healing actions can react on false alarms, or even worse and more importantly create new flaws. The possible solution to reduce false alarms and increace confidence about healing action correctness is to use formal methods, e.g., well-known model checking.

Model checking is an approach of automated checking whether a system satisfies the behaviour given by a required specification. Model checking over real systems is costly unless there are some heuristics or alternatives. One of the alternatives is bounded model checking which is described in the following section. The specification can be written in one of temporal logics like LTL, CTL, CTL*, or in μ-calculus [2]. This specification is used for observing an interesting features of a system or for detecting some bugs or errors in a system. The crucial problem of applying the model checking in a real program is the state space explosion problem—the size of the state space grows exponentially with the size of a system description.

Bounded model checking assumes that bugs in a system can be found in a near surrounding of the suspicious state, i.e. within limited number of steps. This approach is not complete in the way of searching the whole state space it has been proven effective for finding flaws which occur near the suspicious location. The features which can be verified by bounded model checking are deadlocks, unhandled exceptions or data races.

This paper considers programs written in Java. As a model checker for bounded model checking the Java PathFinder (JPF) [1] has been chosen. JPF allows the verification of a concurrency, it is one of the main reason for choosing JPF as a model checker. JPF is implemented as a specialized Java Virtual Machine. JPF is a model checker for Java programs, which verifies systems at the bytecode level. JPF is easily extensible and it implements various available state space search strategies and state space reduction techniques.

In the next section, there is presented the implementation of navigation strategy through the state space into the suspected state. Then, a practial example of bounded model checking used for finding true race in the program is given. Finally, the results and the future work is discussed.

2 NAVIGATION STRATEGIES THROUGH STATE SPACE

For bounded model checking, there is a need to get into the concrete state, e.g., error state or into some previous state, and to apply model checking techniques close to the error state [2]. There are two main approaches—the so-called store&restore state strategy and the record&replay trace strategy. The advantage of record&replay trace strategy is that the model checking can be started from any of recorded states which precede the error state. The record strategy has been chosen for this work.

The selected strategy is based on recording of the trace during program execution and on the replaying it in the model checker. Recording of all the information necessary for replaying the trace consumes amount of memory and causes big CPU load. This method does not scale well for long running applications. For instance, replaying a two days long execution trace can take much more time than is available. This strategy allows to reduce the number of events to be recorded, for instance, by recording branching points only. The described strategy features are interesting for improvement of model checking techniques.

3 EXAMPLE OF BOUNDED MODEL CHECKING

In SHADOWS project, we use a simple example of a multi-threaded Java program: Bank Account [3], consisting of two classes. The main class is class `Bank`, which initializes and runs several threads that represent accounts—objects of the `Account` class. Each `Account` simulates operations with the account. The method `Service()` from `Bank` class contains a bug, which comes from missing proper synchronisation over an account balances. The model checking should confirm suspicion on data race in the `Service()` method (a data race causes inconsistences in data wrt. unexpected sequence of concurrent events).

The verification process receives a recorded trace leading to the erroneous state on its input. Currently, the trace contains pairs of current thread instructions that were executed, the suspected state and the suspected variable which could suffer from data race. The suspected state is a state which occurs somewhere before the error state.

The example is executed in the JPF using record&replay search strategy. Bounded model checking then starts searching suspicious state via a listener which is able to watch suspected variable. Such a listener records all the accesses to the selected shared variable, thus every possibly undesirable access is taken into account.

The bounded model checking generates limited number of states ahead and watch all the accesses to the suspected variable. The report, how the variable is accessed, is produced by bounded model checking. This report contains information on threads, which accessed to variable and sequences of this accesses. It is the possible to determine if the suspected program contains true race in the explored part of state space (a data race in a concurrent program occurs when two threads simultaneously access a shared memory location and at least one of these accesses is a write access).

The situation is illustrated in Figure 1. The upper part of the figure shows the replaying stage of the process. Whenever the suspected state is reached, bounded model checking starts to generate state space and watch the accesses to the suspected variable. The black nodes represent states where true race has been detected.

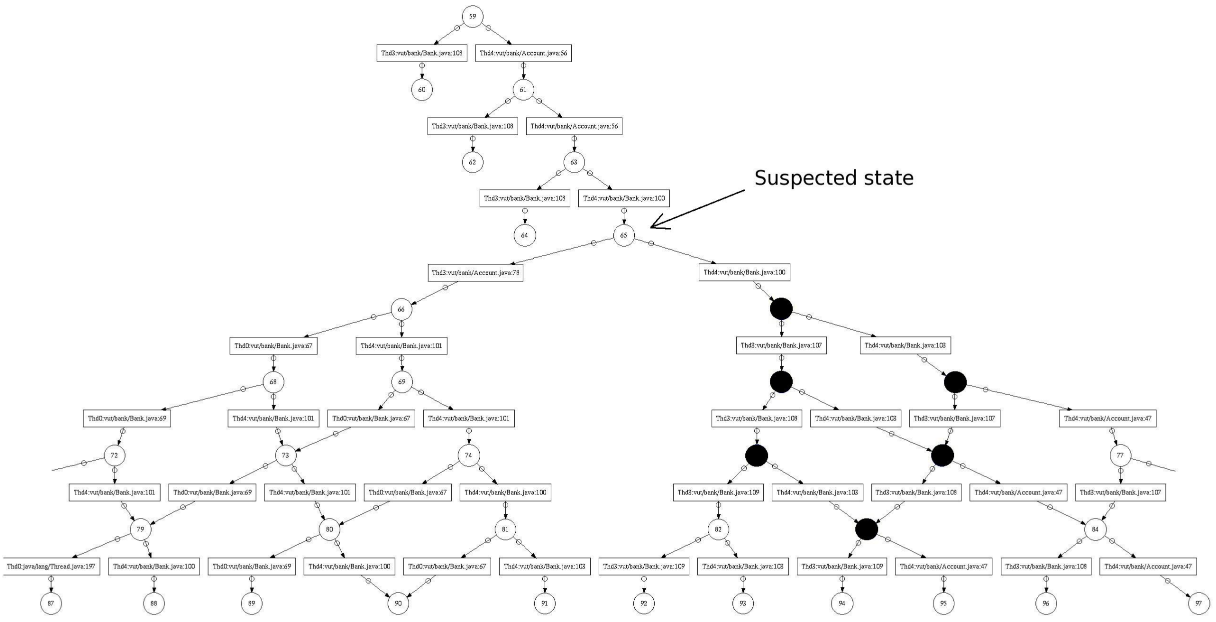
59
Thd3:vut/bank/Bank.java:108
Thd4:vut/bank/Account.java:56
60
61
Thd3:vut/bank/Bank.java:108
Thd4:vut/bank/Account.java:56
62
63
Suspected state
Thd3:vut/bank/Bank.java:108
Thd4:vut/bank/Bank.java:100
64
65
Thd3:vut/bank/Account.java:78
Thd4:vut/bank/Bank.java:100
66
Thd0:vut/bank/Bank.java:67
Thd4:vut/bank/Bank.java:101
Thd3:vut/bank/Bank.java:107
Thd4:vut/bank/Bank.java:103
68
69
Thd0:vut/bank/Bank.java:69
Thd4:vut/bank/Bank.java:101
Thd0:vut/bank/Bank.java:67
Thd4:vut/bank/Bank.java:101
Thd3:vut/bank/Bank.java:108
Thd4:vut/bank/Bank.java:103
Thd3:vut/bank/Bank.java:107
Thd4:vut/bank/Account.java:47
72
73
74
77
Thd4:vut/bank/Bank.java:101
Thd0:vut/bank/Bank.java:69
Thd4:vut/bank/Bank.java:101
Thd0:vut/bank/Bank.java:67
Thd4:vut/bank/Bank.java:100
Thd3:vut/bank/Bank.java:109
Thd4:vut/bank/Bank.java:103
Thd3:vut/bank/Bank.java:108
Thd4:vut/bank/Account.java:47
Thd3:vut/bank/Bank.java:107
79
80
81
82
84
Thd0:java/lang/Thread.java:197
Thd4:vut/bank/Bank.java:100
Thd0:vut/bank/Bank.java:69
Thd4:vut/bank/Bank.java:100
Thd0:vut/bank/Bank.java:67
Thd4:vut/bank/Bank.java:103
Thd3:vut/bank/Bank.java:109
Thd4:vut/bank/Bank.java:103
Thd3:vut/bank/Bank.java:109
Thd4:vut/bank/Account.java:47
Thd3:vut/bank/Bank.java:108
Thd4:vut/bank/Account.java:47
87
88
89
90
91
92
93
94
95
96
97

Figure 1: Exploration of the Bank example

4 CONCLUSION

The usage of the bounded model checking for detection of concurrency errors in programs has been presented. The example of data race detection was given. The future work is to implement the listeners and search strategies for verification of more errors and for obtaining more relevant information about them.

ACKNOWLEDGEMENTS

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.

REFERENCES

[1] P. Mehlitz et al.: Java PathFinder. Robust Software Engineering Group, NASA Ames Research Center. 2007. URL: http://javapathfinder.sourceforge.net.

[2] V. Hrubá, B. Křena, T. Vojnar: Using JavaPathFinder for Self-healing Assurance. In: Proceedings of MEMICS 2007, Znojmo, CZ, 2007, s. 67-73.

[3] B. Křena et al.: Healing Data Races On-The-Fly. In: Proceedings of PADTAD'07, London, GB, 2007, s. 54-64.