A Summary of Asserting and Checking Determinism for Multithreaded Programs

Jennifer Reinelt

Abstract

In their article "Asserting and Checking Determinism for Multithreaded Programs" [1], which was introduced at ESEC-FSE in 2009, Jacob Burnim and Koushik Sen present an assertion framework, that helps to find concurrency errors in multi-threaded programs. Their key idea is to check after every execution of a marked program block whether this block violates a given deterministic specification, i.e. whether the program block was already executed earlier with the same parameters, but the outcome was different.
Burnim et al. implemented this assertion framework as a library for Java. For evaluation purposes they used some benchmark programs from the Java Grande Forum and the Parallel Java Library (and others). It turned out, that all known concurrency errors where found and no additional program parts where marked as buggy by the implemented library.

1 Introduction

1.1 Problem Statement

In order to execute programs more efficient, more and more parallel code is needed. A very common possibility to parallelize code is to use threads. But making sure that for all possible thread interleavings the correctness of the program is still given turns out to be quite challenging.
Even if many executions of the program lead to the correct result, it might happen that others do not.

1.2 Approach

Burnim et al. try to solve this problem of unintentional non-determinism in multithreaded programs by checking whether a given deterministic specification was fullfilled. This deterministic specification is supposed to be given by the programers and gives them the possibility to express deterministic behaviour. If a programer expects deterministic behaviour for a program block `P` , he can use the following construct:

```
deterministic {
  P
}
```

This deterministic specification says that if the program block `P` is executed twice with the same state in the beginning of the block, then executing the block `P` should lead to the same state in the end of the block.

But since it is very unlikely in many applications to reach equivalent states in different executions of a program, programers have the possibility to give constraints that are checked as a replacement for checking whether two states are equivalent. In the following example two matrices `A` and `B` are multiplied in a parallel program, where the entries of the matrices are floats. Due to rounding errors, it is very unlikely to compute the same matrix for `AB` twice even if the matrices `A` and `B` are stay the same for the two executions of the program. Consequently the program is not strictly deterministic. Hence the above deterministic specification without the equivalence condition would be too conservative.
The following construct shows a deterministic specification for a parallel matrix multiplication given by the function `parallel_matrix_multiply_float`.

```
deterministic assume(|A - A'| < 10^-6
                 and |B - B'| < 10^-6){

  C = parallel_matrix_multiply_float(A, B);

} assert(|C - C'| < 10^-6);
```

Here the '-variables refer to a second execution of the program block. Consequently the given predicates relate program states from different executions of the program block. In contrast traditional assertions relate different states of a program from one execution. In the above example the deterministic specification expresses that "if `parallel_matrix_multiply_float`is called twice and the parameters (matrices `A` and `A'` and `B` and `B'` respectively) differ no more than $10^{\overline{-6}}$, then the difference between the resulting matrices `C` and `C'` is supposed to be less than $10^{\overline{-6}}$ as well.
In the following section 2 a more detailed introduction to the assertion framework is given. Thereafter section 3 presents a concept of checking whether a given deterministic specification is fullfilled. In section 4 a concrete implementation of the presented assertion framework is introduced. Thereafter in section 5 the evaluation carried out by Burnim et al. is summarized and finally this work concludes in section 6 with a short summary of the main contributions of "Asserting and Checking Determinism for Multithreaded Programs" [1].

2 Deterministic Specification

In general, parallel code is said to be deterministic if, "given any particular initial state, all executions of the code from the initial state produce the exact same final state" [1]. The following construct expresses this behaviour for program block `P` with a deterministic specification:

```
deterministic{
  P
}
```

In many cases this construct can help to specify the expected deterministic behaviour. For example when multiplying two matrices, whose entries are integers, in parallel:

```
deterministic {
  C = parallel_matrix_multiply_int(A, B);
}
```

Assuming that multiplying matrices `A` and `B` leads to matrix `C` , it can be expected that the multiplication of `A'=A` and `B'=B` leads to matrix `C'=C`. This above deterministic specification expresses this expectation.

In many other cases this specification is too conservative. For example when multiplying two matrices in parallel where the entries of the matrices are floating point numbers. Due to rounding errors, it is probable to obtain different states in the end of the program block, even if the multiplication is started from equivalent initial states.

In order to relax the deterministic specification, *bridge predicates* are used. Bridge predicates are predicates that relate program states from different executions of the same program. In

```
deterministic assume(|A - A'| < 10^-6
                 and |B - B'| < 10^-6){

  C = parallel_matrix_multiply_float(A, B);

} assert(|C - C'| < 10^-6);
```

For example `(|C - C'| < 10^-6)` is a bridge predicate. `C` and `C'` refer to the computed matrix resulting from multiplying `A` and `B` and `A'` and `B'` respectively. The primed variables `A' B'` and `C'` come from a different execution of the program block than `A B` and `C`.
With the help of bridge predicates expressing the relation between the values of one certain variable from different executions of a program and therefore asserting properties concerning determinism is possible.

3 Checking Determinism

So far only the deterministic specification part was considered. In this section the general idea of checking the specification will be given.
We assume that a programer gave a program block `P`, together with a `Pre` and a `Post` condition:

```
deterministic assume(Pre){
   P
} assert(Post);
```

In order to be sure that in P no error occured, it is necessary to check the deterministic specification. One simple and incomplete method to do so was presented in [1]. In this method the program states before (s_{pre}) and after (s_{post}) the certain program block P for every run of the program are recorded at runtime. Then for all pairs of pairs of states (s_{pre}, s_{post}) and ($s_{pre}\prime, s_{post}\prime$) the following implication is to be checked:
$\texttt{Pre}(s_{pre}, s_{pre}\prime) \Rightarrow \texttt{Post}(s_{post}, s_{post}\prime)$.
If this implication is false for at least one pair of pairs of states $((s_{pre}, s_{post}), (s_{pre}\prime, s_{post}\prime))$, then a determinism violation was found. Similarly, if this implication holds for all pairs of pairs, then no concurrency error can be found until now. However it is possible that a determinism violation exists, but the probability decreases with the number of executions of the program block P.
In order to improve the results it might be helpful to combine this determinism checking method with noise making [8] or other tools for exploring thread interleavings.

4 Library

In this section an implementation of the determinism checking method described in the previous section 3 is presented. This implementation was realized in Java. In Listing 1 a concrete example for the usage of the determinism checking library is given. The program renders images of the Mandelbrot Set and belongs to the Parallel Java Library [6]. In the beginning the parameters are read from the command-line and then several threads are used to compute the resulting image.
A call *Deterministic.assume*(o, p) refers to *assume*($p(o, o\prime)$) in the deterministic specification. It checks if the predicate p holds for object o and object $o\prime$ from a former execution of the program. If the current execution of the program is the first one, no check has to be done.
The predicate p can be implemented the way it is supposed to work. Therefore the interface *Predicate* can be used, which requires the implementation of a method *boolean apply*(*Object* $o1$, *Object* $o2$). If this method returns *true* for an object o and object $o\prime$ from a former run, then the assumption holds. As a consequence (and if there are no other assumptions or all assumptions hold), *assert*(o, $o\prime$) has to hold as well. In the library this assertion refers to *Deterministic.assert*(o, p). Here p is also a predicate.

Listing 1: Deterministic assertions for a Mandelbrot Set

```
main (String args[]) {
 //Read parameters from command-line.
 ...

 // Pre-predicate: equal-parameters
 Prediquate equals = new Equals();
 Deterministic.open();
 Deterministic.assume(width, equals);
 Deterministic.assume(heigt, equals);
 ...

 //compute matrix
 int matrix[][] = ...;

 Deterministic.assert(matrix, equals);
 Determinstic.close();
 ...
}
```

A quite intuitive usage of the predicates is to check if objects are equal. This Predicate *Equal* is a built-in predicate. It was already implemented by the authors and calling the function *apply*(*Object* $o1$, *Object* $o2$) simply returns the result of *equals*(*Object* o, *Object* $o\prime$). In Listing 1 this built-in predicate is used three times, e.g. in *Deterministic.assert*(*matrix*, *equals*). It states that if all assumptions (equality of *width* and *height* and maybe more) hold for the current execution of the program and a former one, then calling *equals*(*matrix*, *matrix*$\prime$) has to return *true*. For floating point numbers a similar predicate was already implemented: *ApproxEquals* checks whether the difference of two numbers lies in a given range.
In case that a determinism violation is detected, i.e. an assertion, where all previous assumptions passed,

fails, a message is printed and the application is halted.

5 Evaluation

This section summarizes the evaluation results presented by Burnim et al. in [1]. Two main issues were examined: Ease of Use and effectiveness.
In order to validate both arguments some benchmark programs from the Java Grand Forum (JGF) [7] and the Parallel Java (PJ) Library [6] (and others) where used. An overview is given in Table 1. For each benchmark program one deterministic specification was added manually.

Ease of Use
Although the authors did not know the code of the benchmark programs before, it took them no longer than five to ten minutes to add the deterministic specification for the majority of the programs. Moreover for all but one program the two built-in predicates were sufficient. Only for program *montecarlo* an *equals* and a *hashCode* method had to be implemented additionally. All in all the framework seems to be quite easy to use (especially if compared to asserting functional correctness [1])

Effectiveness
In order to test whether the presented framework really improves concurrency errors detection, a modified version of the tool CALFUZZER [4] [5] was used. CALFUZZER is a data race detection tool, i.e. it is used to find locations in the code where two threads could access a resource simultaneously and at least one of the accesses is writing.
Data races often cause unwished non-determinstic results. But actually the presence of data races does not always lead to non-determinism and the absence of data-races is not enough to ensure determinism [3][2]. Programs can contain high-level data races as well. "The notion of high-level data races refers to sequences in a program where each access to shared data is protected by a lock, but the program still behaves incorrectly because operations that should be carried out atomically can be interleaved with conflicting operations." [2]. CALFUZZER is able to find special kinds of high-level data races as well.

program from JGF	data races and higher level races	determinism violations
sor	2	0
sparsematmult	0	0
series	0	0
crypt	0	0
moldyn	2	0
lufact	1	0
raytracer	3	1
montecarlo	3	0
program from PJ	data races and higher level races	determinism violations
pi	10	1
keysearch3	3	0
mandelbrot	9	0
phylogeny	4	0
tsp	8	0

Table 1: Summary of experimental evaluation of deterministic assertions.

For evaluation purposes CALFUZZER was used to find data-races and high-level data races in the inspected benchmark programs. For each race marked by CALFUZZER 10 trials were run to create real executions and to generate different thread-interleavings. Tab. 1 shows a summary of the results. The second column shows for every program the number of data races and high-level races found by CALFUZZER (one known high-level race was confirmed by hand, because CALFUZZER was not able to find it). The third column shows the number of determinism violations found by the presented framework. Although 44 races were found, only two determinism violations were noticed and these are actually the only known ones for the benchmark programs. The 42 benign races are mostly due to "benign races on volatile variables used for synchronization–for example, to implement a tournament barrier or a custom lock" [1]. CALFUZZER marked these races as potential concurrency errors and the absence of errors has to be

verified by hand, which can be quite challenging. In contrast the presented framework did not mark benign races, it only found the two known determinism violations. I.e. no false alarms were triggered, but all known violations were found.

6 Conclusion

Although the presented approach for checking determinism specifications is incomplete–i.e. if no determinism violations are marked, there may exist some anyway–advantages of using the presented framework where shown: The introduced determinism specifications can be used easily and without much effort. Moreover in contrast to race finding tools bening races can be distinguished from harmful ones.

References

[1] J. Burnim, K. Sen: Asserting and Checking Determinism for Multithreaded Programs. In ESEC-FSE 2009, The 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)

[2] C. Artho, K. Havelund, A. Biere: High-level Data Races. In VVEIS 2003, The First International Workshop on Verification and Validation of Enterprise Information Systems, April 2003. Angers, France.

[3] C. Flanagan, S. N. Freund: Atomizer: A dynamic atomicity checker for multithreaded programs. In 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pages 256-267, 2004

[4] K. Sen: Race directed random testing of concurrent programs. In PLDI 2008, ACM SIGPLAN Conference on Programming Language Design and Implementation

[5] C.-S. Park, K. Sen: Randoized active atomicity violation detection in concurrent programs. In SIGSOFT 2008/FSE-16, Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering

[6] A. Kaminsky: Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java. In IPDPS 2007, 21st IEEE International Parallel and Distributed Processing Symposium

[7] Edinburough Parallel Computing Centre: Java Grande Forum benchmark suite. `www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html`

[8] S. D. Stoller: Testing Concurrent Java Programs using Randomized Scheduling. In RV 2002, Workshop on Runtime Verification