DA-BMC: A Tool Chain Combining Dynamic Analysis and Bounded Model Checking

FIT BUT Technical Report Series

Jan Fiedor, Vendula Hrubá, Bohuslav Křena, and Tomáš Vojnar

Technical Report No. FIT-TR-2011-06
Faculty of Information Technology, Brno University of Technology

Last modified: October 4, 2011

DA-BMC: A Tool Chain Combining Dynamic Analysis and Bounded Model Checking

Jan Fiedor, Vendula Hrubá, Bohuslav Křena, and Tomáš Vojnar

Brno University of Technology, Božetěchova 2,
612 66, Brno, Czech Republic
{ifiedor, ihruba, krena, vojnar}@fit.vutbr.cz
http://www.fit.vutbr.cz

Abstract. This paper presents the DA-BMC tool chain that allows one to combine dynamic analysis and bounded model checking for finding synchronisation errors in concurrent Java programs. The idea is to use suitable dynamic analyses to identify executions of a program being analysed that are suspected to contain synchronisation errors. Some points in such executions are recorded, and then the executions are reproduced in a model checker, using its capabilities to navigate among the recorded points. Subsequently, bounded model checking in a vicinity of the replayed execution is used to confirm whether there are some real errors in the program and/or to debug the problematic execution of the program.

1 Introduction

Despite the constantly growing pressure on quality of software applications, many software errors still appear in the field. One class of errors which can be found in software applications more and more frequently are concurrency-related errors, which is a consequence of the growing use of multi-core processors. Such errors are hard to find by testing since they may be very unlikely to appear. One way to increase chances to detect such an error is to use various *dynamic analyses* (such as Eraser [6] for detection of data races) that try to extrapolate the witnessed behaviour and give a warning about a possible error even if such an error is not really witnessed in any testing run. A disadvantage of such analyses is that they often produce false alarms. To avoid false alarms, one can use *model checking* based on a systematic search of the state space of the given program [1], but this approach is very expensive. In this paper, we describe a tool chain denoted as *DA-BMC*[1] that tries to combine advantages of both dynamic analysis and (bounded) model checking.

In our tool chain, implementing the approach proposed in [3], we use the infrastructure offered by the *Contest* tool [2] to implement suitable dynamic analyses over Java programs and to record selected points of the executions of the programs that are suspected to contain errors. We then use the *Java PathFinder (JPF)* model checker [5] to replay the partially recorded executions, using JPF's capabilities of state space generation to heuristically navigate among the recorded points. In order to allow the navigation, the JPF's state space search strategy, including its use of partial order reduction

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/da-bmc

to reduce the searched state space, is suitably modified. Bounded model checking is then performed in the vicinity of the replayed executions, trying to confirm that there is really some error in the program and/or to debug the recorded suspicious behaviour.

We illustrate capabilities of DA-BMC on several case studies, showing that it really allows one to benefit from advantages of both dynamic analysis and model checking.

2 Recording Suspicious Executions

The first step when using DA-BMC is to use a suitable *dynamic analysis* to identify executions suspected to contain an error and to record some information about them—recording the entire executions would typically be too costly. In DA-BMC, this phase is implemented on top of the *Contest tool* [2]. Contest provides a listener architecture (implemented via Java byte-code instrumentation) on top of which it is easy to implement various dynamic analyses. We further refer to two such analyses, namely, Eraser+ and AtomRace intended for detection of data races (and, in the second case, also atomicity violations), which have been implemented as Contest plugins in [4]. Further analyses can, of course, be added. Contest also provides a noise injection mechanism which increases the probability of manifestation of concurrency-related errors.

In order to record executions, we have implemented another specialised listener on top of Contest. We record information about an execution in the form of a *trace* which is a sequence of *monitored events* that contains partial information about some of the events that happen during the execution. In particular, Contest allows us to monitor the following events: *thread-related* events (thread creation, thread termination), *memory-access-related* events (before integer read, after integer read, before float write, after float write, etc.), *synchronisation-related* events (after monitor enter, before monitor exit, join, wait, notify, etc.), and some *control-related* events (basic block entry, method enter, and method exit). The user can choose only some of such events to be monitored. As shown in our case studies, one should mainly consider synchronisation-related and memory-access-related events, which help the most when dealing with the inherent non-determinism of concurrent executions.

Each monitored event contains information about the source-code location from which it was generated (class and method name, line and instruction number) and the thread which generated it. The recorded trace also contains information produced by the applied dynamic analysis which labels some of the monitored events as suspicious from the point of view of causing an error.

3 Replaying Recorded Traces

The second step when using DA-BMC is to reproduce suspicious executions recorded as traces of monitored events in a model checker. More precisely, there is no guarantee that the same execution as the one from which the given trace was recorded will be reproduced. The tool will simply try to generate some execution whose underlying trace corresponds with the recorded trace. It is also possible to let the model checker generate more executions with the same trace.

In DA-BMC, we, in particular, use the *Java PathFinder (JPF)* model checker [5]. JPF provides several state space search strategies, but also allows one to add new user-specific search strategies. Moreover, it provides a listener mechanism which is useful for performing various analyses of the searched state space and/or for guiding the search strategies to a specific part of the state space. JPF uses several state space reduction techniques, including partial order reduction (POR), which out of several transitions that lead from a certain state may explore only some [1].

A recorded trace is replayed by navigating JPF through the state space of a program such that the monitored events encountered on the search path correspond with the ones in the recorded trace. The states being explored during the search are stored in a priority queue. The priority of the inserted states depends on the chosen search strategy (DFS and BFS are supported). In each step, the next parent state to be processed is obtained from the queue. After that, all relevant children of the parent state are generated. Here, we should note that, in JPF, a transition between a parent and child states represents, in fact, a sequence of events happening in a running program. This sequence is chosen by the POR to represent all equivalent paths between the two states. Into the priority queue, we only save the child states that may appear on a path corresponding to the recorded trace. In other words, each program event encountered within the JPF's transition between the parent and child states must either be an event which is not monitored (and hence ignored), or an event which corresponds with the one stored in the recorded trace at an appropriate position. This correspondence is checked during the generation of a transition in JPF.

Sometimes, it is also necessary to influence the POR used by JPF. That happens when the POR decides to consider another permutation of the events than the one actually present in the trace. Then, the POR is forced to use the needed permutation as follows. If the generation of the sequence of events that the POR wants to compose into a single transition encounters some monitored event, and this event differs from the one expected in the recorded trace, then we force JPF to finish the generation of the sequence of events to be put under a single transition and to create a new state. The navigation algorithm then searches the transitions enabled in this state that correspond with the recorded trace (if there is none, the search backtracks).

Since the replaying is driven by a sequence of monitored events generated from the Contest's instrumentation of the given program, we run the instrumented byte-code in JPF. We, however, make JPF skip all the code that is a part of Contest in order not to increase the size of the state space being searched. Moreover, Contest not only adds some instructions into the code, but also replaces some original byte-code instructions. This applies, e.g., for the instructions `wait`, `notify`, `join`, etc. In this case, when such an instruction is detected in JPF, we dynamically replace it with the original instruction.

As the JPF's implementation of `sleep()` ignores interruption of sleeping threads, we provide a modified implementation of the `interrupt()` and `sleep()` methods which correctly generate an exception if a thread is interrupted by another thread when sleeping. For that to work correctly, the possibility of branching of the execution after `sleep()` must be enabled in JPF.

Still, it might not be possible to replay a trace if the program depends on input or random data or if it uses some specific dynamic data structures like hash tables where, e.g., objects might be iterated in a different order in each run of the program. In these cases, it is necessary to modify the source code of the analysed program, e.g., by adding JPF data choice generators to eliminate these problems.

4 Bounded Model Checking

As we have already said above, the trace recorded from a suspicious execution does not identify the execution from which it was generated in a unique way. Moreover, even the original suspicious execution based on which the applied dynamic analysis generated a warning about the possibility of some error needs not contain an actual occurrence of the error (even if the error is real). To cope with such situations, apart from possibly exploring several paths through the state space corresponding with the recorded trace, we use bounded model checking that starts from the states from which an event that is marked as suspicious is enabled, or from some of its predecessors. The latter is motivated by the fact that once a suspicious event is reached, it may already be too late for a real error to manifest.

To be able to use bounded model checking to see whether an error really appears in the program, it is expected that the user supplies a JPF listener capable of identifying occurrences of the error (in our experiments, which concentrate on data races, we, e.g., use a slight modification of the `PreciseRaceDetector` listener available in JPF). The listeners looking for occurrences of errors may be activated either at the very beginning of replaying of a trace, or they may be activated at the beginning of each application of bounded model checking. The user is allowed to control both the depth of the bounded model checking as well as the number of backward steps to be taken from a suspicious event before starting bounded model checking.

5 Experiments

To demonstrate capabilities of DA-BMC, we consider four case studies. The first two, *BankAccount* and *Airlines*, are simple programs (with 2 or 8 classes, respectively) in which a data race over a single shared variable can happen. The *DiningPhilosophers* case study is a simple program (3 classes) implementing the problem of dining philosophers with a possibility of a deadlock. Finally, our last case study, *Crawler*, is a part of an older version of an IBM production software (containing 19 classes) with a data race manifesting more rarely and further in the execution. All the tests were performed on a machine with 2 Dual-Core AMD Opteron processors at 2.8GHz.

First, we measured the slowdown of program executions when recording various types of events. An analysis of the overhead associated with trace recording is presented in Table 1. The first column of the table gives the average time needed for executing the particular case studies while they are monitored by the Eraser dynamic analysis implemented as a Contest plugin. The second column gives the average slowdown when recording all the events that can be monitored by Contest and hence DA-BMC (cf. Section 2). Finally, the third column gives the average slowdown when the thread and memory-access events are recorded only.

Table 1. Overhead generated by trace recording

	Execution time with Eraser in seconds	Slowdown due to recording (number of events in the trace)	
Bank	1.62	44% (2 035)	34% (1 211)
Airlines	0.85	77% (969)	58% (609)
Crawler	4.45	31% (3 288)	23% (1 859)
DinPhil	0.49	43% (110 035)	25% (55 257)

Table 2. Finding real errors in traces produced by Eraser

No. of traces	Error discovery ratio (traces found / BMC runs)				Time/memory consumption (sec/MB)			
	DFS		BFS		DFS		BFS	
	1	5	1	5	1	5	1	5
Bank	46%(1/1)	49%(2/2)	46%(1/1)	46%(2/2)	2/517	4/633	3/522	5/659
Airlines	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)	1/482	1/482	1/482	1/482
DinPhil	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)	11/417	20/411	20/414	22/413
Crawler	7%(0.8/15)	7%(1.8/34)	2%(0.5/49)	2%(1.2/50)	122/1312	268/1479	311/2857	321/3020

To sum up, when recording all the possible types of events mentioned above, the slowdown was about 30-40 %. When recording only thread and memory-access-related events, the slowdown was just about 20-30 % but the number of corresponding paths found by JPF increased by about 50 %.

Note, however, that the overhead differs quite significantly from one example to another. Therefore, the types of events to be recorded should be chosen depending on the program being analysed, taking into account which kind of events and how often it can generate. For instance, since the DiningPhilosophers case study is a program which frequently switches threads, but minimally accesses shared variables, it is sufficient to record only thread-related events in order to precisely navigate through the state space.

Next, we performed a series of tests in which we measured how often a real error is identified when replaying a trace and performing bounded model checking (BMC) in its vicinity. We let JPF to always backtrack 3 states from the state before a suspicious event and to use the maximum BMC depth of 10. The results are shown in Table 2. We distinguish whether 1 or up to 5 paths corresponding to the recorded trace were explored, using either DFS or BFS. For each of these settings and each case study, the left part of Table 2 gives the percentage of recorded traces based on which a real error was found. Further, in brackets, it is shown how many corresponding paths were on average found by JPF for a single trace, and how many times BMC was on average applied when analysing a single trace. The right part of Table 2 then gives the corresponding time and memory consumption. Clearly, BFS has higher time and memory requirements than DFS (mainly because it performs significantly more runs of BMC). It is also less successful in finding an error if the error manifests later in the execution (like in Crawler). It can also be seen that the number of corresponding paths searched has a little contribution to the overall success of finding a real error.

Table 3. Efficiency of finding errors using DFS in traces of Crawler produced by AtomRace

Traces searched	No. of backtracked states / Max depth of BMC				
	3/10	5/15	10/30	15/45	20/60
1	66%	71%	78%	84%	90%
5	71%	73%	80%	85%	90%

States	Max depth of bounded model checking					
	30	40	50	60	70	80
10	78%(0)	78%(1)	78%(3)	-	-	-
20	-	90%(0)	-	90%(2)	90%(2)	90%(2)
30	-	-	92%(2)	-	94%(0)	-
40	-	-	-	-	-	89%(5)

The low percentage of real errors found in traces of Crawler is mostly due to the number of false alarms produced by Eraser that were eliminated by DA-BMC, which nicely illustrates one of the main advantages of using DA-BMC. Further, note that classical model checking as offered by JPF did not find any error in this case since it ran of our deadline of 8 hours (DFS) or ran out of the 24GB of memory available to JPF (BFS). To analyse how successful DA-BMC is in finding real errors in traces recorded in Crawler and how the success ratio depends on the various settings of DA-BMC, we have then done experiments with traces recorded using the AtomRace analysis, which does not produce any false alarms. The results can be seen in Table 3. Its left part shows how the percentage of real errors found depends on the number of explored paths corresponding to the recorded trace, the number of states backtracked from the state before a suspicious event, and the maximum BMC depth. The right part analyses in more detail how the percentage depends on the number of backtracked states and the maximum BMC depth (a single path corresponding to a recorded trace is analysed). The numbers in brackets express the percentage of replays which reached a 10 minute timeout. We can see that while increasing the number of searched corresponding paths has some influence on the error detection, it is evident that the BMC settings have a much greater impact. Moreover, the number of backtracked states increases the chances to find an error much more than the increased maximum depth of BMC.

6 Conclusion

We have presented DA-BMC—a tool chain combining dynamic analysis and bounded model checking for finding errors in concurrent Java programs (and also for debugging them). We have demonstrated on several case studies that DA-BMC allows one to combine the lower price of dynamic analysis with the higher precision of model checking.

Acknowledgement

This work was supported by the Czech Science Foundation (project no. P103/10/0306 and doctoral project no. 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528) and the internal BUT project FIT-11-1.

References

1. C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

2. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, S. Ur. Framework for Testing Multi-threaded Java Programs. *Concurrency and Computation: Practice and Experience*, 15(3-5), 2003.
3. V. Hrubá, B. Křena, and T. Vojnar. Self-healing assurance using bounded model checking. In *Proc. of EUROCAST'09*, LNCS 5717, Springer, 2009.
4. B. Křena, Z. Letko, Y. Nir-Buchbinder, R. Tzoref, S. Ur, T. Vojnar. A Concurrency Testing Tool and its Plug-ins for Dynamic Analysis and Runtime Healing. In *Proc. of RV'09*, LNCS 5779, Springer, 2009.
5. W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. Model Checking Programs. Automated Software Engineering Journal, 10(2), 2003.
6. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of SOSP'97*, ACM Press, 1997.