Dynamic Validation of Contracts in Concurrent Code

Jan Fiedor[1], Zdeněk Letko[1], João Lourenço[2], and Tomáš Vojnar[1]

[1] IT4Innovations Centre of Excellence, FIT, Brno University of Technology, Czech Republic
{ifiedor, iletko, vojnar}@fit.vutbr.cz
[2] NOVA LINCS, Dep. Informática, FCT – Universidade Nova de Lisboa, Portugal
joao.lourenco@fct.unl.pt

Abstract. Multi-threaded programs allow one to achieve better performance by doing a lot of work in parallel using multiple threads. Such parallel programs often contain code blocks that a thread must execute atomically, i.e., with no interference from the other threads of the program. Failing to execute these code blocks atomically leads to errors known as atomicity violations. However, frequently it not obvious to tell when a piece of code should be executed atomically, especially when that piece of code contains calls to some third-party library functions, about which the programmer has little or no knowledge at all. One solution to this problem is to associate a contract with such a library, telling the programmer how the library functions should be used, and then check whether the contract is indeed respected. For contract validation, static approaches have been proposed, with known limitations on precision and scalability. In this paper, we propose a dynamic method for contract validation, which is more precise and scalable than static approaches.

1 Introduction

With multi-core processors present in all the newest computers, multi-threaded programs are becoming increasingly common. However, multi-threaded programs require proper synchronisation to restrict the thread interleavings and make the program produce correct results. Failing to do so often leads to various critical errors, which occur under some very specific timing scenarios only, and standard testing and debugging techniques are less effective or even useless for their detection.

Atomicity violations are a class of errors which result from an incorrect definition of the scope of an atomic region. Such errors are usually hard to localise and diagnose, which becomes even harder when using (third-party) software libraries where it is unknown to the programmer how to form the atomic regions correctly when accessing the library. Even new synchronisation techniques, such as transactional memories, designed to ease the process of writing concurrent programs, do not entirely avoid this problem and suffer from atomicity violations as well [3].

One way to address the problem of proper atomicity is to associate a *contract* with each program module/library and then check whether the contract is indeed respected. In fact, the notion of contract is, in general, not restricted to concurrent programs. In the general case, a contract [8] regulates the use of methods of an object by specifying a set of pre-conditions the program must meet before calling the object methods. For the particular case of concurrent programs, Sousa et al. proposed in [10] the concept of the so-called *contracts for concurrency*. A contract for concurrency contains a set of clauses where each clause defines a (finite) set of sequences of method calls that must be executed atomically whenever they are executed on the same object. Contract clauses may be written by the software module/library developer or inferred automatically from the program (based on its typical usage patterns) [10].

In this paper, assuming that the appropriate contracts for concurrency have been obtained, we propose a method for dynamically verifying that such contracts are respected at program run time. In particular, our method belongs among the so-called *lockset-based* dynamic analyses whose classic example is the Eraser algorithm for data race detection [9] and whose common feature is that they track sets of locks that are held by various threads and used for various synchronization purposes. The tracked lock sets are used to extrapolate the synchronization behaviour seen in the witnessed test runs, allowing one to warn about possible errors even when they do not directly appear in the witnessed test runs. We have implemented our approach in a prototype tool, and we present some encouraging experimental results obtained with our implementation.

2 Related Work

A notion of *contract* was first introduced by Meyer [8] in 1992 as a sequence of tasks (commands) with defined pre- and post-conditions. If this sequence was executed without meeting these conditions, the contract was violated. Contracts in the form of regular expressions were used to specify protocols for accessing objects in sequential [1] as well as concurrent scenarios [2, 7, 10]. Hurlin in [7] proposes a technique to validate the correctness of contracts by checking contracts on a set of artificially generated programs that use a particular object. Both Demeyer in [2] and Sousa in [10] propose to use a static approach to address the contract validation.

The static approaches of [2, 10] can formally prove that no contract violation is possible. For that, however, they assume that properly handled contracts must appear in code blocks declared as atomic (with the atomicity assured by the run-time support). If a different way of guarding the contracts is used, a false alarm is issued. Moreover, the approaches scale to relatively small programs only. For more complex programs, one has to restrict the analysis to program fragments, e.g., individual methods, in order to achieve a reasonable performance. This leads to a loss of precision as contracts may span across several methods and thus be missed by the analysis.

Another problem with the static approach is related to the fact that contracts for concurrency are required to operate atomically only when all the involved method calls operate on the same object. This is a natural requirement since the atomic execution is critical only when working with data elements that are mutually related, which is assumed to be reflected in that they are stored within one object. However, static validation does not have precise information on which objects the methods are called on. Hence, calls of methods on different objects are mixed together, leading to possible false alarms. Classic alias and escape analyses can be used to infer this information from the source code of the program, but these analyses provide only approximate information and may still lead to false alarms.

Our dynamic approach of contract validation avoids the above false alarms since it has precise run-time information about the objects that particular methods are executed on. Moreover, it also scales quite well. On the other hand, despite the lockset-based method that we use extrapolates to some degree the behaviour of the witnessed test runs, our approach can miss some contract violations that do not happen in the witnessed test runs nor they can be deduced from the locking patterns used in these traces. In order to minimize the number of possibly missed contract violations, one can combine our approach with *noise injection* techniques [6] that maximize the number of thread interleavings witnessed in a set of test runs.

3 Contracts for Concurrency

A *contract for concurrency* [10] (or simply *contract* herein) is a protocol for accessing the public services of a module, i.e., the methods of its public API, in a concurrent setting. Each module shall have its own contract, which contains one or more sequences of tasks (methods). The condition to be met here is that the sequences of methods must be executed atomically whenever executed on the same object.

Formally, let Σ_c be a set of all public method names (API) of a module (or library). A *contract* is a set S of *clauses* where each clause $s \in S$ is a star-free regular expression over Σ_c. A contract is violated if any of the sequences represented by the contract is not executed atomically when executed on the same object o, meaning that it is interleaved with an execution of some method from Σ_c on the object o.

4 Dynamic Validation of Contracts

In order to detect atomicity violations in more complex programs and to reduce the number of false alarms, we propose a dynamic approach to check whether a contract is violated or not. Our dynamic validation looks for contract violations based on concrete program executions. Possible violations not witnessed during the execution of the program may be missed, but all of the methods encountered during the execution are taken into account, and so contract violations caused by method calls from all over the program are detected. Since all of the threads are running and all objects are known when the program is executing, we know precisely whether all of the methods called in a sequence use the same object, and we do not report any false alarms due to mixing calls on different objects as is common in static analysis.

Since we look for contract violations based on concrete executions, we can avoid some false alarms, but on the other hand, we can miss some errors. In order to minimise this possibility, we employ one of the dynamic analysis techniques—namely, the *lock sets* [9]—to extrapolate the actually witnessed behaviour and hence detect possible contract violations even when they were not actually witnessed. Moreover, we utilise noise injection techniques [6] to enforce synchronisation scenarios, which normally appear only rarely, leading to behaviours (and possibly contract violations) that would not be covered by extrapolation of the common synchronisation scenarios only.

4.1 Detection of Contracts

In order to validate a contract, we first need to detect the sequences it contains in the execution of a program. To do that, we encode each contract, i.e., all of its sequences, as a single finite state automaton. As each clause of the contract represents a regular expression, we use standard methods for transforming (star-free) regular expressions[3] into finite automata to perform the conversion and then merge all these automata into a single one. The transitions of the automaton represent method calls and the accepting states represent situations where a contract sequence was detected.

[3] Star-free regular expressions are used in the static contract validation approach [10]. We can, however, easily generalize our approach to general regular expressions.

Each thread manages a list of finite state automata instances which represent the currently encountered incomplete contract sequences. Whenever a method $m \in \Sigma_c$ is encountered, we try to advance each of these instances using the method m. If we cannot advance the instance, the contract sequence is invalid and we discard it. If we successfully advanced the instance to the next state, call it q, we check if q is an accepting state. If yes, a contract sequence is detected; if not, we leave the instance in q and go on. Moreover, we check if we can advance any of the finite state automata from their starting state using the method m. If yes, then the beginning of another contract sequence was detected and we create a new instance of the automaton which will monitor the execution of this contract sequence to check if it can be accepted or not.

4.2 Checking the Atomicity Condition

When a contract sequence is detected, the next step is to check if the atomicity condition is met, i.e., if the program ensures that all methods of this contract sequence are executed atomically. The static approach does this by checking if all of the methods of the detected contract sequence are enclosed in code blocks declared as atomic, which can be done by analysing the source code of the program.

We propose a lockset-based method, inspired by [9], to perform these checks which is more suited for dynamic analysis. This method checks if at least one lock is held during the execution of a contract sequence by monitoring the lock acquisitions and releases during the execution of a contract sequence. If this condition is not satisfied, i.e., no locks are held throughout the execution, then the contract is being violated.

The method works online, i.e., it performs the contract validation during the execution of a program, and is based on the analysis state $\sigma = (A, H, R)$ where:

- $A : T \to 2^L$ records the set of locks acquired by a thread.
- $H : T \times S \to 2^L$ records the set of locks held by a thread when a contract sequence starts.
- $R : T \times S \to 2^L$ records the set of locks released by a thread during the execution of a contract sequence.

In the initial analysis state, all sets of locks are empty, reflecting that at the beginning of the execution, no locks are held by any thread, i.e., $\sigma_0 = (\emptyset, \emptyset, \emptyset)$. Fig. 1 shows rules according to which the analysis state is updated for each operation of the target program.

The rule [CONTRACT SEQUENCE START] records that a thread t is starting an execution of a contract sequence s by remembering the locks which are currently held by the thread. It also clears the set of locks released by the thread as no locks could have been released yet.

The rule [CONTRACT SEQUENCE END] records that a contract sequence s

[CONTRACT SEQUENCE START]

$$\frac{\begin{array}{l} H' := H_t[s := A_t] \\ R' := R_t[s := \emptyset] \end{array}}{(A, H, R) \Rightarrow^{seq\_start(t,s)} (A, H', R')}$$

[CONTRACT SEQUENCE END]

$$\frac{\mathtt{\textbf{if}}\ H_t(s) \setminus R_t(s) = \emptyset\ \mathtt{\textbf{then}}\ \text{ERROR}}{(A, H, R) \Rightarrow^{seq\_end(t,s)} (A, H, R)}$$

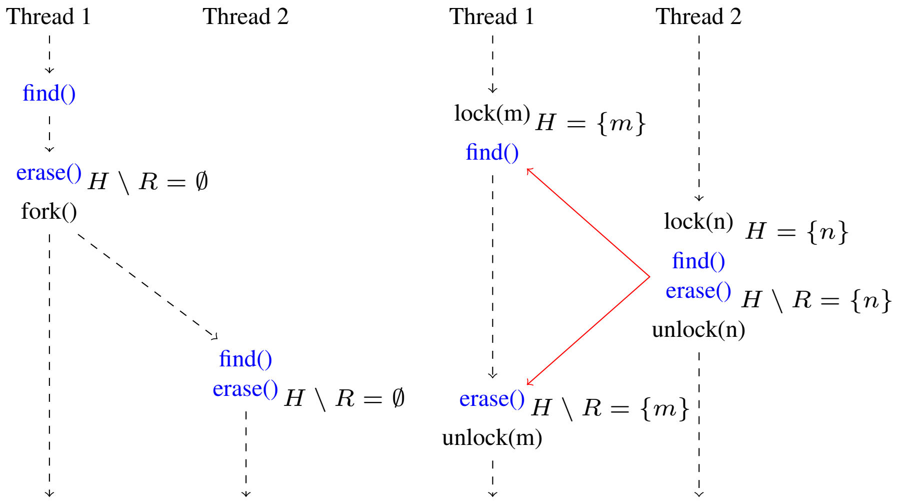
[LOCK ACQUIRED]

$$\frac{A' := A[t := A_t \cup \{m\}]}{(A, H, R) \Rightarrow^{acq(t,m)} (A', H, R)}$$

[LOCK RELEASED]

$$\frac{\forall s \in S : R' := R_t[s := R_t(s) \cup \{m\}]}{(A, H, R) \Rightarrow^{rel(t,m)} (A, H, R')}$$

Fig. 1: Analysis rules.


Thread 1
Thread 2
find()
erase() $H \setminus R = \emptyset$
fork()
find()
erase() $H \setminus R = \emptyset$
Thread 1
Thread 2
lock(m) $H = \{m\}$
find()
lock(n) $H = \{n\}$
find()
erase() $H \setminus R = \{n\}$
unlock(n)
erase() $H \setminus R = \{m\}$
unlock(m)

(a) An atomically executed contract reported as errorneous.

(b) A not reported contract violation.

Fig. 2: Examples of situations where the contract validation fails.

was detected in a thread t and checks the atomicity condition by comparing the set of locks held when the contract sequence started its execution with the set of locks released during its execution. If at least one lock was held all the time the contract sequence was executed, the contract is valid, and no error is issued. If all locks held at the beginning of the execution of the contract sequence were released before its execution finished, a contract violation is reported.

The rule [LOCK ACQUIRED] records that a thread t acquired a lock m, and it updates the set of locks currently acquired by this thread. Finally, the rule [LOCK RELEASED] records that a thread t released a lock m, and it updates the set of locks released by the thread for each contract sequence currently executed by this thread.

4.3 Discussion of the Proposed Approach

The above method may produce both false positives (i.e., false alarms) as well as false negatives. False positives may be caused by the fact that not guarding an execution of a contract sequence with a single lock throughout its entire duration does not mean that it will not be executed atomically. Take the situation shown in Fig. 2(a) as an example. Not a single one of the contract sequences is guarded by a lock, yet there is no contract violation as the synchronisation ensures that the contract sequence in `Thread 1` is always executed before the contract sequence in `Thread 2`. Therefore there is no interference between these two contract sequences, and hence no contract violation. Yet the method reports both of the contract sequences being violated.

False negatives may happen since holding a lock when executing a contract sequence does not always ensure that no other thread interferes with it. Take the situation in Fig. 2(b) as an example. The executions of the contract sequence in both `Thread 1` and `Thread 2` are guarded by a lock. However, these locks are different and thus the execution of the contract sequence in `Thread 1` may be interleaved with the execution of the contract sequence in Thread 2, violating the contract sequence in Thread 1. Yet the method does not report any error.

To solve the above problems, we need to take into account thread interleavings. When guarding the same contract sequence with two different locks in two different threads, we should issue an error only when these two threads may interleave each other. Conversely, when a contract sequence is not guarded by a lock, we should report an error only when this thread may be interleaved by another thread executing the same contract sequence. When using the static approach, this information is hard to obtain as one would need to infer it from the source code of the program where the scheduling of threads is unknown. On the other hand, the dynamic approach actually sees the concrete thread interleavings, and so it is easier to get the needed information. Unfortunately, the lockset method does not work with it in any way. Moreover, incorporating this information into the lockset method would be counterproductive as it would kill the extrapolation which increases chances to detect errors. A way to go here seems to be a use of dynamic analysis based on the *happens-before relation* as used, e.g., in the GoldiLock data race detector [5], which is a part of our future work.

5 Experiments

This section presents an experimental comparison of the proposed dynamic validation of contracts with the static approach of [10]. To compare the approaches, we implemented the method described in Section 4 as a plug-in for the IBM Concurrency Testing Tool (ConTest) [4]. The ConTest infrastructure provides a fully automatic Java bytecode instrumentation and a listeners architecture that facilitated the implementation of the proposed method as well as execution and dynamic analysis of the benchmarks.

The comparison of the static and dynamic approaches is done using a subset of the small benchmark programs which were previously used to evaluate the static approach [10], namely, the Account, Allocate Vector, Arithmetic DB, Jigsaw, Store, and Vector fail test cases. All these benchmark programs had to be slightly modified in order to allow us to execute them and use ConTest to analyse their runs. Namely, we did the following modifications by hand: (1) test arguments were provided if missing; (2) infinite loops (which are not a problem for the static approach, but cannot be present during a dynamic analysis) were transformed to finite loops with a small number of iterations to avoid infinite executions; (3) exceptions generation and handling (commented out due to limits of the static approach) were uncommented; (4) the Atomic annotations preferred by the static approach were turned back to synchronized blocks; and (5) all assertions and correctness checks already present in the test cases were extended to send notifications to our ConTest plug-in. The dynamic analysis tests were executed on a Linux machine with an i5-4200M CPU (i.e., comparable with the machine used to evaluate the static approach in [10]), running Linux 3.16, and OpenJDK 1.6 JVM.

Table 1 summarises results of the comparison between our dynamic approach and the static approach of [10]. The table is divided into three sections. In the leftmost part, basic characteristics of the benchmark programs are provided. In particular, the test case name, the number of effective lines of the original Java code (without our modifications, which added only a few extra lines of code), and the number of contract clauses for the benchmark program as manually identified by the authors of the static approach.

The middle part of Table 1 characterizes results of the static analysis obtained in [10]. Namely, the average analysis duration in seconds is provided with the num-

Table 1: An experimental comparison of static and dynamic contract validation.

Program			*Static analysis*			*Dynamic analysis*		
Benchmark	LOC	Contract Clauses	Duration (sec.)	CFG Nodes	Detected Violations	Duration (sec.)	Detected Violations	Failed Assertions
Account	68	2	0,041	158	2	0,011	2	0,96
Allocate Vector	167	1	0,120	882	1	0,099	1	0,00
Arithmetic DB	325	2	0,272	2256	2	0,010	2	0,06
Jigsaw	147	1	0,044	125	1	0,009	1	0,44
Store	769	1	0,090	559	1	0,303	1	1,00
VectorFail	100	2	0,048	244	2	0,009	2	0,09

ber of control flow graph (CFG) nodes generated and processed. Finally, the number of detected violations of contract clauses for each benchmark program is shown.

The rightmost part of the table shows the average results (from 1000 test executions) obtained with our dynamic approach. In particular, the average execution time of the instrumented test in seconds is provided, followed by the average number of detected violations of contract clauses. Finally, the average ratio of failed assertions (usually implemented as conditions checking memory consistency) provided by the authors of the tests is reported. The standard deviations of the execution times as well as failed assertions were quite low. The standard deviation of the number of violated contract was zero (i.e., the algorithm always detected all the violations).

Concerning both the considered static as well as dynamic approach, there are two interesting aspects we would like to emphasize: (i) the ability of both approaches to detect contract violations; and (ii) the very low execution time taken by both approaches (of course, a further evaluation on larger test cases remains to be done).

In both approaches, all violations were *always* correctly reported. Such a good result of the dynamic approach depends on the quality of the test that executes the problematic part of the code and on the ability of the lockset approach to extrapolate other behaviours from the witnessed execution, and therefore to detect possible violations even from executions where the problem did not occur. This can, in particular, be demonstrated on the Allocate Vector, Arithmetic DB, and VectorFail benchmark programs where the assertion-based detection reported the problem in less than 10 % of executions while the dynamic approach always detected a possible violation.

Let us now get back to the time consumed by the analyses. In both cases, the analysis itself took less than one second for the considered simple test programs. However, there was a significant difference in the overhead of the underlying infrastructures. The initialisation of the static approach within the Soot analysis environment [10] took nearly 40 seconds for each benchmark. The dynamic approach was much faster. The bytecode instrumentation took about 0.5 seconds. The slowdown of the test execution was within 5 % because only the *method entry* and *lock operation* events were instrumented (i.e., most of the code was executed with no instrumentation and hence no overhead).

6 Conclusion

We presented a method for dynamic validation of contracts in concurrent code. When compared with previously proposed static approaches, our approach can suppress some of the false alarms produced by these approaches, and it is also more scalable. Since we build on observing concrete runs, our approach can miss some errors that would not be missed by static analysis. To detect as many contract violations as possible, our approach employs a lockset-based extrapolation of the synchronization behaviour observed in performed test runs, which allows the method to warn about possible contract violations even when they were not seen in a concrete execution. Moreover, noise injection may be used to increase the number of observed thread interleavings, and hence chances to see interleavings containing a contract violation or at least symptoms that such a violation is possible.

The extrapolation we use can suffer from both false positives and negatives due to the fact that the lockset method used does not utilise any information about thread interleavings. In order to solve this problem, we plan to use extrapolation methods based on the happens-before relations, which do reflect thread interleavings. Another interesting subject for future work is then generalization of the notion of contracts (e.g., by considering full regular expressions), exploiting the fact that such generalizations seem to be much easier in the context of dynamic analysis.

Acknowledgements. This work was supported by the ESF COST Action IC1001 (Euro-TM), the COST project LD14001 and the Kontakt II project LH13265 of the Czech ministry of education, the BUT project FIT-S-14-2486, the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, the EU/Czech Interdisciplinary Excellence Research Teams Establishment project CZ.1.07/2.3.00/30.0005, and the Portuguese National Science Foundation in the strategic project FCT/MEC NOVA LINCS PEst UID/CEC/04516/2013.

References

1. Y. Cheon and A. Perumandla. Specifying and Checking Method Call Sequences of Java Programs. *Software Quality Control*, 15(1):7–25, Mar. 2007.
2. R. Demeyer and W. Vanhoof. Static Application-Level Race Detection in STM Haskell using Contracts. In *Proc. of PLACCES*. Open Publishing Association, 2013.
3. R. J. Dias, V. Pessanha, and J. M. Lourenço. Precise detection of atomicity violations. In A. Biere, A. Nahir, and T. Vos, editors, *Hardware and Software: Verification and Testing*, volume 7857 of *Lecture Notes in Computer Science*, pages 8–23. Springer Berlin / Heidelberg, Nov. 2013. HVC 2012 Best Paper Award.
4. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.
5. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*. ACM, 2007.
6. J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in noise-based testing. *STVR*, 24(7):1–38, 2014.
7. C. Hurlin. Specifying and checking protocols of multithreaded classes. In *Proc. of SAC'09*, pages 587–592. ACM, 2009.
8. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.
9. 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, 1997.
10. D. G. Sousa, R. J. Dias, C. Ferreira, and J. M. Lourenço. Preventing atomicity violations with contracts. *eprint arXiv:1505.02951*, May 2015.