Verifying Real-World Software with Contracts for Concurrency

João M. Lourenço(✉)

NOVA LINCS—NOVA Laboratory for Computer Science and Informatics,
Departamento de Informática, Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
joao.lourenco@fct.unl.pt

Abstract. In this paper we present *Contracts for Concurrency*. A contract for concurrency specifies the protocol to access the services provided by a software module or library. A program that respects a (well-defined and complete) contract for a module is safe from high-level atomicity violations with respect to that module. On the other hand, violations of a contract may denote errors in the program, and the application of contracts for concurrency to some real-world open source software packages did uncover a few latent bugs.

1 Introduction

The encapsulation of a set of services as software module offers strong advantages in the software development process, since it promotes the reuse of code and eases the maintenance efforts. If unacquainted with the implementation details of a particular set of services provided by a software module, the programmer may fail to identify correlations that exist across those services, such as data and code dependencies, and misuse the services and introduce bugs in the program. This is particularly relevant in a concurrent setting, where it is hard to account for all the possible interleavings between threads and their effects in the module's internal state when its services are used.

When using a (third-party) software module, a program must follow its *protocol*, which defines the legal sequences of invocations of its methods. For instance, a module that offers an abstraction to deal with files will typically demand the programmer to start by calling the method `open()`, followed by an arbitrary number of `read()` and `write()` operations, and concluding with a call to `close()`. A program that does not follow this protocol is incorrect and should be fixed. A way to enforce a program to conform to such well defined behaviours is to use the design by contract methodology [5], and specifying contracts that regulate the module's protocol. Contracts serve as useful documentation and may be automatically verified, thus ensuring the program obeys the protocol [1,3].

Concurrency brings new challenges to the definition of module protocols, as it also brings additional requirements such as ensuring the atomic execution of sequences of calls that are susceptible of causing atomicity violations. Some of these (high-level) atomicity violations are possible even when the individual methods of the module are protected by some concurrency control mechanism.

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11247, pp. 70–73, 2018.
https://doi.org/10.1007/978-3-030-03427-6_9

Figure 1 shows part of a program that suffers from a high-level atomicity violation. The `schedule()` method gets a task from a queue and executes it only if it is ready to run. This program contains a potential high-level atomicity violation since the method may end executing a task that was not marked as ready. This may happen when another thread concurrently schedules the same task, despite the fact the methods of `Task` are atomic. In this case the `isReady()` and `run()` methods should be executed in the same atomic context to avoid atomicity violations. Atomicity violations are one of the most common source of bugs in concurrent programming [4], and are particularly susceptible to occur when composing calls to an external module whose implementation details are unknown.

```
void schedule() {
    Task t=taskQueue.next();

    if (t.isReady())
        t.run();
}
```

Fig. 1. Program with a atomicity violation.

In this paper we describe *Contracts for Concurrency* [2,6], which improves the correctness of concurrent programs by extending module usage protocols with a specification of the sequences of calls that should be executed atomically.

2 Contract Specification

The contract of a module is a protocol, and this protocol specifies which sequences of calls of the non-private methods of that module are to executed atomically. In the spirit of the *programming by contract* methodology, the specification of the terms of the contract, including the identification of the sequences of methods that should be executed atomically, is a responsibility of the module's developer.

The *Contract for Concurrency* of a module with public methods $m_1, \cdots, m_n$ is a set of clauses, each clause c_i is described by e_i, a star-free regular expression over the alphabet $\{m_1, \cdots, m_n\}$. Star-free regular expressions are regular expressions without the Kleene star, using only the alternative (|) and the (implicit) concatenation operators.

Each sequence defined in e_i must be executed atomically by the program using the module, otherwise there is a violation of the contract. Our verification analysis assumes that the contract defines a finite number of call sequences.

Example. Consider the array implementation as offered by the *Java* standard library, `java.util.ArrayList`. For simplicity we will only consider the methods `add(obj)`, `contains(obj)`, `indexOf(obj)`, `get(idx)`, `set(idx,obj)`, `remove(idx)`, and `size()`.

The following contract defines some of the clauses for this class.

1. `contains indexOf`
2. `indexOf (remove | set | get)`
3. `size (remove | set | get)`
4. `add (get | indexOf)`

Clause 1 of the above contract denotes that the execution of `contains()` followed by `indexOf()` should be atomic, otherwise the client program may confirm the existence

of an object in the array, but fail to obtain its index due to a concurrent modification. Clause 2 represents a similar scenario where, in addition, the position of the object is modified. Clause 3 deals with the common situation where the program verifies if a given index is valid before accessing the array. To make sure the size obtained by `size()` is valid when accessing the array, these calls must be atomic. Clause 4 maps scenarios where an object is added to the array and then the program tries to obtain information about that object by querying the array.

Another relevant clause is "`contains indexOf (set | remove)`", but the contract's semantic already enforces the atomicity of this clause as a consequence of the composition of clauses 1 and 2, as they overlap in the `indexOf()` method.

2.1 Extending Contracts with Parameters

Basic contracts may be too restrictive because they enforce methods to always be executed atomically. However, frequently two methods may have to be executed atomically if they refer to the same data and my be executed concurrently otherwise. To incorporate this concept, Contracts were extended with parameters [2,6].

Example. When including symbolic parameters, the above contract may be refined as:

1. `contains(X) indexOf(X)`
2. `X=indexOf(_) (remove(X) | set(X,_) | get(X))`
3. `X=size() (remove(X) | set(X,_) | get(X))`
4. `add(X) ( get(X) | indexOf(X))`

This contract uses the underscore as a free variable, and captures in detail the dependencies between method calls, expressing the relations that are problematic and excluding those that do not cause atomicity violations, hence allowing for more concurrency without compromising correctness.

2.2 Extending Contracts with Spoilers

Some sequences of method calls from the module's API may need to be executed atomically with respect to some well identified methods, while with some others they do not. For example, the clause `contains indexOf` states that this sequence of calls must always be executed atomically (w.r.t. methods of the given module), regardless of which methods the other threads are executing. Interleaving a thread executing this sequence with another one is thus a contract violation regardless of whether the other thread executes `remove` or `get`, not distinguishing that the former is harmful while the latter is not.

Contract spoilers [2] address the above problem by allowing to express in which context the contract clauses shall be enforced. For that, each clause of the basic contract (a *target*) is coupled with a set of *spoilers* that restrict its application. A spoiler represents a set of sequences of methods that may violate its target. Client programs must then ensure that each target is executed atomically w.r.t. its spoilers, whenever executed on the same object. For the target clause `contains indexOf`, a possible spoiler is `remove`, and the extended clause would be: `contains indexOf` $\rightsquigarrow$ `remove`.

Example. The basic contract for `java.util.ArrayList` extended with parameters and spoilers is:

1. `contains(X) indexOf(X) ⇜ remove(_)`
2. `X=indexOf(_) ( remove(X) | set(X,_) | get(X) ) ⇜ remove(_) | add(_) | set(X,_)`
3. `X=size() ( remove(X) | set(X,_) | get(X) ) ⇜ remove(_)`
4. `add(X) ( get(X) | indexOf(X) ) ⇜ remove(_) | set(X,_)`

3 Conclusions

The *Contracts for Concurrency* may be validated statically or dynamically [2,6]. Experiments with applying these Contracts in real-world programs resulted in identifying multiple atomicity violations in production open-source softwares such as Tomcat[1], Derby[2], Cassandra[3], Chromium-1 (version 6.0.472.35). Both, the automatic generation of Contracts for Concurrency and their extension to address other concurrency issues are current ongoing work.

Acknowledgments. This paper describes work that was developed in collaboration with other colleagues from NOVA University Lisbon and Brno University of Technology [2,6]. This work was partially supported by NOVA LINCS (UID/CEC/ 04516/2013) and the National Science Foundation (FCT/MEC) in the framework of the HiPsTr research project (02/SAICT/2017–032456).

References

1. Cheon, Y., Perumandla, A.: Specifying and checking method call sequences of Java programs. Softw. Qual. Control. **15**(1), 7–25 (2007)
2. Dias, R.J., et al.: Verifying concurrent programs using contracts. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 196–206, March 2017
3. Hurlin, C.: Specifying and checking protocols of multithreaded classes. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC 2009, pp. 587–592. ACM, New York (2009)
4. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. SIGPLAN Not. **43**(3), 329–339 (2008)
5. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992)
6. Sousa, D.G., Dias, R.J., Ferreira, C., Lourenço, J.M.: Preventing atomicity violations with contracts. arXiv preprint arXiv:1505.02951, May 2015

[1] https://issues.apache.org/bugzilla/show_bug.cgi?id=56784.
[2] https://issues.apache.org/jira/browse/DERBY-6679.
[3] https://issues.apache.org/jira/browse/CASSANDRA-7757.