Violat: Generating Tests of Observational Refinement for Concurrent Objects

Michael Emmi[1] and Constantin Enea[2]

[1] SRI International, New York, NY, USA, michael.emmi@sri.com
[2] IRIF, Univ. Paris Diderot & CRNS, Paris, France, cenea@irif.fr

Abstract. High-performance multithreaded software often relies on optimized implementations of common abstract data types (ADTs) like counters, key-value stores, and queues, i.e., *concurrent objects*. By using fine-grained and non-blocking mechanisms for efficient inter-thread synchronization, these implementations are vulnerable to violations of ADT-consistency which are difficult to detect: bugs can depend on specific combinations of method invocations and argument values, as well as rarely-occurring thread interleavings. Even given a bug-triggering interleaving, detection generally requires unintuitive test assertions to capture inconsistent combinations of invocation return values.

In this work we describe the Violat tool for generating tests that witness violations to atomicity, or weaker consistency properties. Violat generates self-contained and efficient programs that test *observational refinement*, i.e., substitutability of a given ADT with a given implementation. Our approach is both sound and complete in the limit: for every consistency violation there is a failed execution of some test program, and every failed test signals an actual consistency violation. In practice we compromise soundness for efficiency via random exploration of test programs, yielding probabilistic soundness instead. Violat's tests reliably expose ADT-consistency violations using off-the-shelf approaches to concurrent test validation, including stress testing and explicit-state model checking.

1 Introduction

Many mainstream software platforms including Java and .NET support multithreading to enable parallelism and reactivity. Programming multithreaded code effectively is notoriously hard, and prone to data races on shared memory accesses, or deadlocks on the synchronization used to protect accesses. Rather than confronting these difficulties, programmers generally prefer to leverage libraries providing *concurrent objects* [29, 19], i.e., optimized thread-safe implementations of common abstract data types (ADTs) like counters, key-value stores, and queues. For instance, Java's concurrent collections include implementations which eschew the synchronization bottlenecks associated with lock-based mutual exclusion, opting instead for non-blocking mechanisms [28] provided by hardware operations like *atomic compare and exchange*.

Concurrent object implementations are themselves vulnerable to elusive bugs: even with effective techniques for exploring the space of thread interleavings, like stress testing or model checking [7, 30, 47], bugs often depend on specific combinations of method invocations and argument values. Furthermore, even recognizing whether a given execution is *correct* is non-trivial, since recognition generally requires unintuitive test assertions to identify inconsistent combinations of return values. Technically, correctness amounts to *observational refinement* [18, 21, 32], which captures the substitutability of an ADT with an implementation [23]: any combination of values admitted by a given implementation is also admitted by the given ADT specification.

In this work we describe an approach to generating tests of observational refinement for concurrent objects, as implemented by the Violat tool, which we use to discover violations to atomicity (and weaker consistency properties) in widely-used concurrent objects [9, 10, 12]. Unlike previous approaches based on *linearizability* [20, 46, 4], Violat generates self-contained test programs which do not require enumerating linearizations dynamically *per execution*, instead statically precomputing the ADT-admitted return-value outcomes *per test program*, once, prior to testing. Despite this optimization, the approach is both sound and complete, i.e., in the limit: for every consistency violation there is a failed execution of some test program, and every failed test witnesses an actual consistency violation. In practice, we compromise soundness for efficiency via random exploration of test programs, achieving probabilistic soundness instead.

Besides improving the efficiency of test execution, Violat's self-contained tests can be validated by both stress testers and model checkers, and double as regression and conformance tests. Our previous works [9, 10, 12] demonstrate that Violat's tests reliably expose ADT-consistency violations in Java implementations using the Java Concurrency Stress testing tool [42]. In particular, Violat has uncovered atomicity violations in over 50 methods from Java's concurrent collections; many of these violations seem to correspond with their documentations' mention of *weakly-consistent* behavior, while others indicate confirmed implementation bugs, which we have reported.

Previous work used Violat in empirical studies, without artifact evaluation [9, 10, 12]. This article is the first to consider Violat itself for evaluation, the first to describe its implementation and usage, and includes several novel extensions. For instance, in addition to stress testing, Violat now includes an integration with Java Pathfinder [47]; besides enabling complete systematic coverage of a given test program, this integration enables the output of the execution traces leading to consistency violations, thus facilitating diagnosis and repair. Furthermore, Violat is now capable of generating tests of any user-provided implementation, in addition to those distributed with Java.

2 Overview of Test Generation with Violat

Violat generates self-contained programs to test the observational refinement of a given concurrent object implementation with respect to its abstract data type (ADT), according to Figure 1. While its methodology is fairly platform agnostic, Violat currently integrates with the Java platform. Accordingly, its input includes the fully-qualified name of a single Java class, which is assumed to be available either on the system classpath, or in a user-provided Java archive (JAR); its output is a sequence of Java classes which can be tested with off-the-shelf back-end analysis engines, including the Java Concurrency Stress testing tool [42] and Java Pathfinder [47]. Our current implementation integrates directly with both back-ends, and thus reports test results directly, signaling any discovered consistency violations.

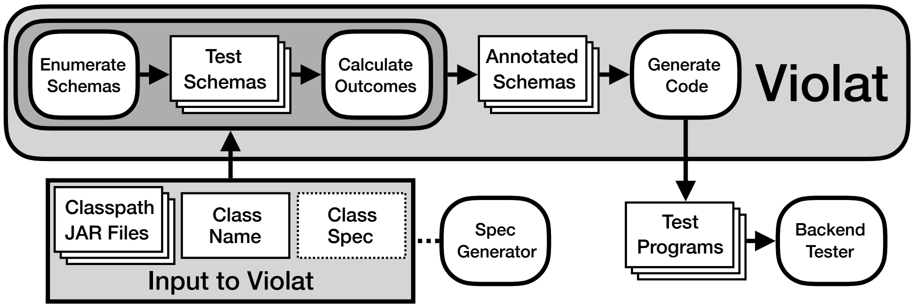

Fig. 1. Violat generates tests by enumerating program schemas invoking a given concurrent object, annotating those schemas with the expected outcomes of invocations according to ADT specifications, and translating annotated schemas to executable tests.

Violat generates tests according to a three-step pipeline. The first step, described in Section 3, enumerates test program *schemas*, i.e., concise descriptions of programs as parallel sequences of invocations of the given concurrent object's methods. For example, Figure 2 lists several test schemas for Java's ConcurrentHashMap. The second step, described in Section 4, annotates each schema with a set of expected *outcomes*, i.e., the combinations of return values among the given schema's invocations which are admitted according to the given object's ADT specification. The final step, described in Section 5, translates each schema into a self-contained[3] Java class.

Technically, to guide the enumeration of schemas and calculation of outcomes, Violat requires a specification of the given concurrent object, describing constructor and method signatures. While this could be generated automatically from the object's bytecode, our current implementation asks the user to input this specification in JSON format. By additionally indicating whether methods are read-only or weakly-consistent, the user can provide additional hints to improve schema enumeration and outcome calculation. For instance, excessive generation of programs with only read-only methods is unlikely to uncover consistency violations, and weakly-consistent ADT methods generally allow additional outcomes — see Emmi and Enea [12]. Furthermore, Violat attempts to focus the blame for discovered violations by constructing tests with a small number of specified *untrusted* methods, e.g., just one.

3 Test Enumeration

To enumerate test programs effectively, Violat considers a simple representation of program *schemas*, as depicted in Figure 2. We write schemas with a familiar notation, as parallel compositions `{...}||{...}` of method-invocation sequences. Intuitively, schemas capture parallel threads invoking sequences of methods of a given concurrent object. Besides the parallelism, these schemas include only trivial control and data flow. For instance, we exclude conditional statements and loops, as well as passing

[3] The generated class imports only a given concurrent object, and a few basic `java.util` classes.

java.util.ConcurrentHashMap	
Schema / **Method**	Outcome
{ put(0,0); put(1,1); put(1,1)} || { put(0,1); **clear**() }	N,N,N,N,**()**
{ put(0,0); remove(1) } || { put(1,0); **contains**(0) }	N,0,N,**F**
{ get(1); **containsValue**(1) } || { put(1,1); put(0,1); put(1,0) }	1,**F**,N,N,1
{ put(0,1); put(1,0) } || { **elements**() }	N,N,**[0]**
{ put(0,1); put(1,0) } || { **entrySet**() }	N,N,**[1=0]**
{ put(1,1) } || { put(1,2); **isEmpty**() }	N,1,**T**
{ put(0,1); put(1,1) } || { **keySet**() }	N,N,**[1]**
{ **keys**()} || { put(0,1); put(1,1) }	**[1]**,N,N
{ put(1,0); put(1,1); **mappingCount**()} || { remove(1) }	N,N,**2**,0
{ put(1,0); put(1,1); **size**()} || { remove(1) }	N,N,**2**,0
{ put(0,1); put(1,1) } || { **toString**() }	N,N,**1=1**
{ put(0,1); put(1,0) } || { **values**() }	N,N,**[0]**

Fig. 2. Program schemas generated by Violat for Java's ConcurrentHashMap class, along with outcomes which are observed in testing, yet not predicated by Violat.

return values as arguments, in favor of straight-line code with literal argument values. Nevertheless, this simple notion is expressive enough to capture any possible *outcome*, i.e., combination of invocation return values, of programs with arbitrarily complex control flow, data flow, and synchronization. To see this, consider any outcome $\boldsymbol{y}$ admitted by some execution of a program with arbitrarily-complex control and data flow in which methods are invoked with argument values $\boldsymbol{x}$, collectively. The schema in which each thread invokes the same methods of a thread of the original program with literal values $\boldsymbol{x}$, collectively, is guaranteed to admit the same outcome $\boldsymbol{y}$.

For a given concurrent object, Violat enumerates schemas according to a few configurable parameters, including bounds on the number of threads, invocations, and (primitive) values. By default, Violat generates schemas with exactly 2 threads, between 3 and 6 invocations, and exactly 2 values. While our initial implementation enumerated schemas systematically according to a well-defined order, empirically we found that this strategy spends too much time in neighborhoods of uninteresting schemas, i.e., which do not expose violations. Ultimately we adopted a pseudorandom enumeration which constructs each schema independently by randomly choosing the number of threads, invocations, and values, within the given parameter bounds, and randomly populating threads with invocations. Methods are selected according to a weighted random choice, in which the weights of read-only and untrusted methods is 1; trusted mutator methods have weight 3. The read-only and trusted designations are provided by class specifications — see Section 2. Integer argument values are chosen randomly between 0 and 1, according to the default value bound; generic-typed arguments are assumed to be integers. Collection and map values are constructed from randomly-chosen integer values, up to size 2. In principle, all of these bounds are configurable, but we have found these defaults to work reasonably well.

Note that while the manifestation of a given concurrency bug can, in principle, rely on large bounds on threads, invocations, and values, recent studies demonstrate that the majority (96%) can be reproduced with just 2 threads [25]. Furthermore, while our current implementation adheres to the simple notion of schema in which all threads are execute in parallel, Violat can easily be extended to handle a more complex notion of schema in which threads are partially ordered, thus capturing arbitrary program synchronization. Nevertheless, this simple notion seems effective at exposing violations without requiring additional synchronization — see Emmi and Enea [12, §5.2].

4 Computing Expected Outcomes

To capture violations to observational refinement, Violat computes the set of *expected outcomes*, i.e., those admitted by a given concurrent object's abstract data type (ADT), for each program schema. Violat essentially follows the approach of Line-Up [4] by computing expected outcomes from sequential executions of the given implementation. While this approach assumes that the sequential behavior of a given implementation does adhere to its implicit ADT specification — and that the outcomes of concurrent executions are also outcomes of sequential executions — there is typically no practical alternative, since behavioral ADT specifications are rarely provided.

Violat computes the expected outcomes of a given schema once, by enumerating all possible shuffles of threads' invocations, and recording the return values of each shuffle when executed by the given implementation. For instance, there are 10 ways to shuffle the threads of the schema

```
{ get(1); containsValue(1) } || { put(1,1); put(0,1); put(1,0) }
```

from Figure 2, including the sequence

```
get(1); put(1,1); put(0,1); put(1,0); containsValue(1).
```

Executing Java's ConcurrentHashMap on this shuffle yields the values null, null, null, 1, and true, respectively. To construct the generated outcome, Violat reorders the return values according to the textual order of their corresponding invocations in the given schema; since containsValue is second in this order, after get, the generated outcome is null, true, null, null, 1. Among the 10 possible shuffles of this schema, there are only four unique outcomes — later given explicitly in Figures 3 and 4.

Note that in contrast to existing approaches based on *linearizability* [20], including Line-Up [4], which enumerate linearizations *per execution* of a given program, Violat only enumerates linearizations once *per schema*. This is made possible for two reasons. First, by considering simple test programs in which all invocations are known *statically*, we know the precise set of invocations (including argument values) to linearize even before executing the program. Second, according to sequential happens-before consistency [12], we consider the recording of real-time ordering among invocations infeasible on modern platforms like Java and C++11, which provide only weak ordering guarantees according to a platform-defined happens-before relation. This enables the static prediction of ordering constraints among invocations. While this static enumeration is also exponential in the number of invocations, it becomes an additive rather than multiplicative factor, amounting to significant performance gains in testing.

```
public class Test {
  public static class StringResult5 {
    @sun.misc.Contended public String r1;
    @sun.misc.Contended public String r2;
    ...
    public String toString() {
      return r1 + ", " + ... + ", " + r5;
    }
  }

  static StringResult5 results;
  static HashSet<String> expected;
  static ConcurrentHashMap obj;
  static {
    obj = new ConcurrentHashMap();
    results = new StringResult5();
    expected = new HashSet<String>();
    expected.add("0, true, null, null, 1");
    expected.add("1, true, null, null, 1");
    expected.add("null, true, null, null, 1");
    expected.add("null, false, null, null, 1");
  }
```

```
  // ...continued from the column to the left

  static String stringify(Object object) { ... }

  public static void main(String[] args) {
    Thread thread1 = new Thread(() -> {
      results.r1 = stringify(obj.get(1));
      results.r2 = stringify(obj.containsValue(1));
    });

    Thread thread2 = new Thread(() -> {
      results.r3 = stringify(obj.put(1, 1));
      results.r4 = stringify(obj.put(0, 1));
      results.r5 = stringify(obj.put(1, 0));
    });

    thread1.start(); thread2.start();
    thread1.join(); thread2.join();

    assert expected.contains(results.toString());
  }
}
```

Fig. 3. Code generated for the containsValue schema of Figure 2 for use with Java Pathfinder. Code generation for the Java Concurrency Stress testing tool is similar, but conforms to the tool's idiomatic test format using decorators, and built-in thread and outcome management.

5 Code Generation and Back-End Integrations

Once schemas are annotated with expected outcomes, the translation to actual test programs is fairly straightforward. Note that until this point, Violat is mainly agnostic to the underlying platform for which tests are being generated. The only exception is in computing the expected outcomes for schema linearizations, which executes the given concurrent object implementation as a stand-in oracle for its implicit ADT specification.

Figure 3 lists a slight simplification of the code generated for the containsValue schema of Figure 2. The test program initializes a concurrent-object instance and a hash table of expected outcomes, then runs the schema's threads in parallel, recording the results of each invocation, and checks, after threads complete, whether the recorded outcome is expected. To avoid added inter-thread interference and the masking of potential weak-memory effects, each recorded result is isolated to a distinct cache line via Java's *contended* decorator. The actual generated code also includes exception handling, which we have omitted here for brevity.

Our current implementation of Violat integrates with two analysis back-ends: the Java Concurrency Stress testing tool [42] (jcstress) and Java Pathfinder [47]. Figure 4 demonstrates the results of each tool on the code generated from the containsValue schema of Figure 2. Each tool observes executions with the 4 expected outcomes, as well as executions yielding an outcome that Violat does not predict, thus signaling a violation to observational refinement (and atomicity). Java Pathfinder explores 18 program paths in a few seconds — achieving exhaustiveness via partial-order reduction [16] — while jcstress explores nearly 4 million executions in 1 second, observing the unpredicted outcome only twice. Aside from this example, Violat has uncovered consistency violations in over 50 methods of Java's concurrent collections [9, 10, 12].

ConcurrentHashMap: containsValue			
`{ get(1); containsValue(1) } || { put(1,1); put(0,1); put(1,0) }`			
outcome	atomic?	paths via JPF	frequency via jcstress
`0, true, null, null, 1`	✓	3	13,287
`1, false``, null, null, 1`	×	3	2
`1, true, null, null, 1`	✓	3	16,417
`null, false, null, null, 1`	✓	6	3,638,600
`null, true, null, null, 1`	✓	3	9,504

Fig. 4. Observed outcomes for the size method, as recorded by Java Pathfinder and jcstress. Outcomes list return values in program-text order, e.g., the return value of get is listed first.

6 Usage

Violat is implemented as a Node.js command-line application, available from GitHub and the npm package manager.[4] Its basic functionality is provided by the command:

```
$ violat-validator ConcurrentHashMap.json
...
violation discovered
---
{ put(0,1); size(); contains(1) } || { put(0,0); put(1,1) }
---
outcome                 OK  frequency
----------------------  --  ---------
0, 0, true, null, null  X   7
0, 1, true, null, null  ✓   703
0, 2, true, null, null  ✓   94,636
null, 1, false, 1, null ✓   2,263
null, 1, true, 1, null  ✓   59,917
null, 2, true, 1, null  ✓   4
...
```

which reports violations among 100 generated programs. Alternatively, user-provided classes, individual schemas, program limits, and particular back-ends can be specified:

```
$ violat-validator MyConcurrentHashMap.json \
  --jar MyCollections.jar \
  --schema "{get(1); containsValue(1)} || {put(1,1); put(0,1); put(1,0)}" \
  --max-programs 1000 \
  --tester "Java Pathfinder"
```

A full selection of parameters is available from the usage instructions:

```
$ violat-validator --help
```

[4] https://github.com/michael-emmi/violat, https://www.npmjs.com/package/violat

7 Related Work

Many recent works propose approaches to test generation for concurrent object implementations; Terragni and Pezzè outline an excellent survey [45]. Like Violat, Ballerina [31] and ConTeGe [33] enumerate tests randomly, while ConSuite [43], AutoConTest [44], and CovCon [6] exploit static analysis to compute potential shared-memory access conflicts to reduce redundancy among generated tests. Similarly, Omen [35, 37, 36, 38], Narada [40], Intruder [39], and Minion [41] reduce redundancy by anticipating potential concurrency faults while executing tests sequentially. While Ballerina [31] and ConTeGe [33] do compute linearizations, all of these works are limited to identifying concurrency-related faults like data races, deadlocks, and exceptions only raised in concurrent executions. Unlike Violat, which recognizes ADT-admitted return-value combinations, these approaches are neither sound nor complete for testing observational refinement: executions with un-admitted return-value combinations which do not trigger faults are false negatives, while executions with admitted return-value combinations which do trigger faults are generally false positives — many non-blocking concurrent object implementations exhibit data races by design. We consider the key innovations of these works, particularly redundancy elimination, orthogonal to ours; reducing redundancy in generated tests could also improve Violat. While Pradel and Gross do consider subclass substitutability [34], they only consider programs with two concurrent invocations, and require exhaustive enumeration of the superclass's thread interleavings to calculate admitted outcomes. In contrast, Violat computes expected outcomes without interleaving method implementations, i.e., considering them atomic.

Other recent works propose approaches to test generation for memory consistency models. TSOtool [17] generates random tests to validate memory systems against the total-store order (TSO) model, while LCHECK [5] employs genetic algorithms. Mador-Haim et al. [26, 27] generate litmus tests to distinguish several memory models, including TSO, partial-store order (PSO), relaxed-memory order (RMO), and sequential consistency (SC). CppMem [2] considers the C++ memory model, while Herd [1] considers release-acquire (RA) and Power in addition to the aforementioned models. McVerSi [8] employs genetic algorithms to enhance test coverage, while Wickerson et al. [48] leverage the Alloy model finder [22]. In some sense, these works generate tests of observational refinement for hardware platforms implementing memory-system ADTs, i.e., providing read and write operations, whereas Violat generates tests for arbitrary ADTs, including collections with arbitrarily-rich sets of operations.

Violat more closely follows work on *linearizability* checking. Herlihy and Wing [20] established the soundness of linearizability for observational refinement, and Filipovic et al. [14] established completeness. Wing and Gong [49] developed a linearizability-checking algorithm, which was later adopted by Line-Up [4] and optimized by Lowe [24]; while Violat pays the exponential cost of enumerating linearizations once *per program*, these approaches pay that cost *per execution* — an exponential quantity itself. Gibbons and Korach [15] established NP-hardness of per-execution linearizability checking for arbitrary objects, while Emmi and Enea [11] demonstrate tractability for collections. Bouajjani et al. [3] propose polynomial-time approximations, and Emmi et al. [13] demonstrate efficient symbolic algorithms. Finally, Emmi and Enea [9, 10, 12] apply Violat to checking atomicity and weak-consistency of Java concurrent objects.

References

[1] Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36(2), 7:1–7:74 (2014), https://doi.org/10.1145/2627752

[2] Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 55–66. ACM (2011), https://doi.org/10.1145/1926385.1926394

[3] Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 651–662. ACM (2015), https://doi.org/10.1145/2676726.2677002

[4] Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Zorn, B.G., Aiken, A. (eds.) Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010. pp. 330–340. ACM (2010), https://doi.org/10.1145/1806596.1806634

[5] Chen, Y., Lv, Y., Hu, W., Chen, T., Shen, H., Wang, P., Pan, H.: Fast complete memory consistency verification. In: 15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA. pp. 381–392. IEEE Computer Society (2009), https://doi.org/10.1109/HPCA.2009.4798276

[6] Choudhary, A., Lu, S., Pradel, M.: Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In: Uchitel, S., Orso, A., Robillard, M.P. (eds.) Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017. pp. 266–277. IEEE / ACM (2017), https://doi.org/10.1109/ICSE.2017.32

[7] Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001), http://books.google.de/books?id=Nmc4wEaLXFEC

[8] Elver, M., Nagarajan, V.: Mcversi: A test generation framework for fast memory consistency verification in simulation. In: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016. pp. 618–630. IEEE Computer Society (2016), https://doi.org/10.1109/HPCA.2016.7446099

[9] Emmi, M., Enea, C.: Exposing non-atomic methods of concurrent objects. CoRR abs/1706.09305 (2017), http://arxiv.org/abs/1706.09305

[10] Emmi, M., Enea, C.: Monitoring weak consistency. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 487–506. Springer (2018), https://doi.org/10.1007/978-3-319-96145-3_26

[11] Emmi, M., Enea, C.: Sound, complete, and tractable linearizability monitoring for concurrent collections. PACMPL 2(POPL), 25:1–25:27 (2018), https://doi.org/10.1145/3158113

[12] Emmi, M., Enea, C.: Weak-consistency specification via visibility relaxation. PACMPL 3(POPL), 60:1–60:28 (2019), https://dl.acm.org/citation.cfm?id=3290373

[13] Emmi, M., Enea, C., Hamza, J.: Monitoring refinement via symbolic reasoning. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 260–269. ACM (2015), https://doi.org/10.1145/2737924.2737983

[14] Filipovic, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. 411(51-52), 4379–4398 (2010), https://doi.org/10.1016/j.tcs.2010.09.021

[15] Gibbons, P.B., Korach, E.: Testing shared memories. SIAM J. Comput. 26(4), 1208–1244 (1997), https://doi.org/10.1137/S0097539794279614

[16] Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, Lecture Notes in Computer Science, vol. 1032. Springer (1996), https://doi.org/10.1007/3-540-60761-7

[17] Hangal, S., Vahia, D., Manovit, C., Lu, J.J., Narayanan, S.: Tsotool: A program for verifying memory systems using the memory consistency model. In: 31st International Symposium on Computer Architecture (ISCA 2004), 19-23 June 2004, Munich, Germany. pp. 114–123. IEEE Computer Society (2004), https://doi.org/10.1109/ISCA.2004.1310768

[18] He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 86, European Symposium on Programming, Saarbrücken, Federal Republic of Germany, March 17-19, 1986, Proceedings. Lecture Notes in Computer Science, vol. 213, pp. 187–196. Springer (1986), https://doi.org/10.1007/3-540-16442-1_14

[19] Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)

[20] Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990), https://doi.org/10.1145/78969.78972

[21] Hoare, C.A.R., He, J., Sanders, J.W.: Prespecification in data refinement. Inf. Process. Lett. 25(2), 71–76 (1987), https://doi.org/10.1016/0020-0190(87)90224-9

[22] Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11(2), 256–290 (2002), https://doi.org/10.1145/505145.505149

[23] Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16(6), 1811–1841 (1994), https://doi.org/10.1145/197320.197383

[24] Lowe, G.: Testing for linearizability. Concurrency and Computation: Practice and Experience 29(4) (2017), https://doi.org/10.1002/cpe.3928

[25] Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Eggers, S.J., Larus, J.R. (eds.) Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008. pp. 329–339. ACM (2008), https://doi.org/10.1145/1346281.1346323

[26] Mador-Haim, S., Alur, R., Martin, M.M.K.: Generating litmus tests for contrasting memory consistency models. In: Touili, T., Cook, B., Jackson, P.B. (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6174, pp. 273–287. Springer (2010), https://doi.org/10.1007/978-3-642-14295-6_26

[27] Mador-Haim, S., Alur, R., Martin, M.M.K.: Litmus tests for comparing memory consistency models: how long do they need to be? In: Stok, L., Dutt, N.D., Hassoun, S. (eds.) Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011. pp. 504–509. ACM (2011), https://doi.org/10.1145/2024724.2024842

[28] Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Burns, J.E., Moses, Y. (eds.) Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996. pp. 267–275. ACM (1996), https://doi.org/10.1145/248052.248106

[29] Moir, M., Shavit, N.: Concurrent data structures. In: Mehta, D.P., Sahni, S. (eds.) Handbook of Data Structures and Applications. Chapman and Hall/CRC (2004), https://doi.org/10.1201/9781420035179.ch47

[30] Musuvathi, M., Qadeer, S.: CHESS: systematic stress testing of concurrent software. In: Puebla, G. (ed.) Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4407, pp. 15–16. Springer (2006), https://doi.org/10.1007/978-3-540-71410-1_2

[31] Nistor, A., Luo, Q., Pradel, M., Gross, T.R., Marinov, D.: Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. pp. 727–737. IEEE Computer Society (2012), https://doi.org/10.1109/ICSE.2012.6227145

[32] Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. 5(3), 223–255 (1977), https://doi.org/10.1016/0304-3975(77)90044-5

[33] Pradel, M., Gross, T.R.: Fully automatic and precise detection of thread safety violations. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 521–530. ACM (2012), https://doi.org/10.1145/2254064.2254126

[34] Pradel, M., Gross, T.R.: Automatic testing of sequential and concurrent substitutability. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. pp. 282–291. IEEE Computer Society (2013), https://doi.org/10.1109/ICSE.2013.6606574

[35] Samak, M., Ramanathan, M.K.: Multithreaded test synthesis for deadlock detection. In: Black, A.P., Millstein, T.D. (eds.) Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014. pp. 473–489. ACM (2014), https://doi.org/10.1145/2660193.2660238

[36] Samak, M., Ramanathan, M.K.: Omen+: a precise dynamic deadlock detector for multithreaded java libraries. In: Cheung, S., Orso, A., Storey, M.D. (eds.) Proceed-

ings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. pp. 735–738. ACM (2014), https://doi.org/10.1145/2635868.2661670

[37] Samak, M., Ramanathan, M.K.: Omen: a tool for synthesizing tests for deadlock detection. In: Black, A.P. (ed.) Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '14, Portland, OR, USA, October 20-24, 2014 - Companion Volume. pp. 37–38. ACM (2014), https://doi.org/10.1145/2660252.2664663

[38] Samak, M., Ramanathan, M.K.: Trace driven dynamic deadlock detection and reproduction. In: Moreira, J.E., Larus, J.R. (eds.) ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014. pp. 29–42. ACM (2014), https://doi.org/10.1145/2555243.2555262

[39] Samak, M., Ramanathan, M.K.: Synthesizing tests for detecting atomicity violations. In: Nitto, E.D., Harman, M., Heymans, P. (eds.) Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015. pp. 131–142. ACM (2015), https://doi.org/10.1145/2786805.2786874

[40] Samak, M., Ramanathan, M.K., Jagannathan, S.: Synthesizing racy tests. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 175–185. ACM (2015), https://doi.org/10.1145/2737924.2737998

[41] Samak, M., Tripp, O., Ramanathan, M.K.: Directed synthesis of failing concurrent executions. In: Visser, E., Smaragdakis, Y. (eds.) Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016. pp. 430–446. ACM (2016), https://doi.org/10.1145/2983990.2984040

[42] Shipilev, A.: The java concurrency stress tests (2018), https://wiki.openjdk.java.net/display/CodeTools/jcstress

[43] Steenbuck, S., Fraser, G.: Generating unit tests for concurrent classes. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013. pp. 144–153. IEEE Computer Society (2013), https://doi.org/10.1109/ICST.2013.33

[44] Terragni, V., Cheung, S.: Coverage-driven test code generation for concurrent classes. In: Dillon, L.K., Visser, W., Williams, L. (eds.) Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. pp. 1121–1132. ACM (2016), https://doi.org/10.1145/2884781.2884876

[45] Terragni, V., Pezzè, M.: Effectiveness and challenges in generating concurrent tests for thread-safe classes. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 64–75. ACM (2018), https://doi.org/10.1145/3238147.3238224

[46] Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P.B. (eds.) Computer Aided Verification, 22nd International Conference, CAV

2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6174, pp. 450–464. Springer (2010), https://doi.org/10.1007/978-3-642-14295-6_40

[47] Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with java pathfinder. In: Avrunin, G.S., Rothermel, G. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004. pp. 97–107. ACM (2004), https://doi.org/10.1145/1007512.1007526

[48] Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 190–204. ACM (2017), http://dl.acm.org/citation.cfm?id=3009838

[49] Wing, J.M., Gong, C.: Testing and verifying concurrent objects. J. Parallel Distrib. Comput. 17(1-2), 164–182 (1993), https://doi.org/10.1006/jpdc.1993.1015