Toward New Unit-Testing Techniques for Shared-Memory Concurrent Programs

Sung-Shik Jongmans
Department of Computer Science, Open University of the Netherlands
Centrum Wiskunde & Informatica (CWI), the Netherlands Foundation of Scientific Research Institutes (NWO-I)
Heerlen, the Netherlands
ssj@ou.nl

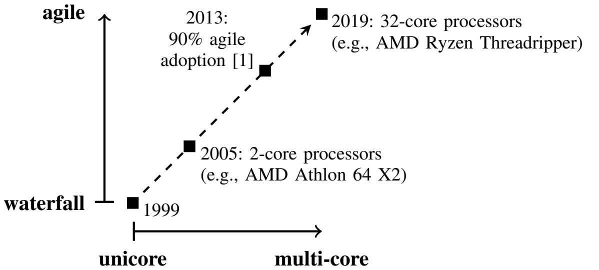

Fig. 1. Paradigm shifts in computing: hardware eng. (x) vs. software eng. (y)

***Abstract*—Following advances in hardware engineering (multi-core processors) and software engineering (agile practices), there is now a large demand for unit-testing techniques for concurrent code. This paper presents the motivation, problem, proposed solution, first results, and open challenges of an early-stage research project (2019–2022) that aims to develop innovative such techniques. Founded on existing work on coordination models and languages, the project's idea is to use a combination of domain-specific language, compilation, and model-checking to build a fully automated framework for unit-testing concurrency.**

***Index Terms*—coordination, domain-specific language, compilation, model-checking**

I. MOTIVATION

Figure 1 illustrates two major—seemingly unrelated—paradigm shifts in computing of the past decades: in **hardware engineering** (x-axis), chip manufacturers shifted from producing unicore processors to developing multi-core architectures; meanwhile, in **software engineering** (x-axis), development teams shifted from abiding by waterfall practices to adopting agile methods. These shifts impact a vast population of "common" software engineers: the shift to multi-core means they now need to write *concurrent code*; the shift to agile means they now need to write *unit tests*.[1] As a result, there is now a large demand for unit-testing techniques for concurrent code.

A substantial amount of research has been conducted on testing techniques for concurrent code in general, and unit-testing techniques in particular (e.g., [3]–[8]). Testing of concurrent code is substantially more complex than testing of non-concurrent code due to the exponentially many interleavings in which threads may be scheduled. The difficulty is to attain the highest coverage (i.e., check many schedules) with the lowest effort (i.e., spend few resources); accordingly, existing unit-testing techniques for concurrent code "mainly focus on re-executing existing test cases with different schedules" [8].

The premise of this project is that *typical concurrent code*, written in a *typical general-purpose language* (GPL), actually exhibits **too little structure** to effectively unit-test a significant class of functionality; and moreover, that **more structure** can make the complicated approach of "re-executing existing test cases with different schedules" avoidable. The objectives of this project are therefore to develop: (1) declarative *programming techniques* to add more structure to concurrent code, along with (2) complementary *unit-testing techniques* to exploit that structure. The guiding principle is to raise the level of abstraction as far as possible (e.g., beyond P/P# [9], [10]), to maximize programmability and testability; then, the key challenge is to achieve competitive performance—of both code and unit tests—at the resulting high level of abstraction.[2]

II. PROBLEM

To illustrate the lack of structure in typical concurrent code, and why it is problematic, imagine a concurrent chess program that consists of two threads, W (White, the human player) and B (Black, the computer player), and two message queues through which W and B communicate (one in every direction). Each thread has a local copy of the board and runs a loop until it reaches a final configuration. In each iteration:

- W receives a move from B; then updates (its local copy of) the board accordingly; then writes the board to the terminal (output to the user); then reads the next move from the terminal (input from the user); then updates the board accordingly; then sends its move to B.
- B analyses (its local copy of) the board to precompute possible next moves (*before* learning about W's move); then receives a move from W; then updates the board

---

Funded by the Netherlands Organisation of Scientific Research (NWO): 016.Veni.192.103

[1] "The idea that any new code must be accompanied by new tests [...] is one of the major contributions of agile methods." [2]

[2] The focus of this project is on shared-memory concurrency; challenges that pertain specifically to distribution and networking (e.g., dealing with failures [11], [12]), are beyond the scope of this project. However, the project's aim and techniques to avoid "re-executing existing test cases with different schedules" seem novel (cf., for instance, model-based testing of networked systems [13]) and applicable to distributed and networked applications as well.

```
 1 public class Board { ... }
 2 public class Move  { ... }

 3 public static void runWhite(BlockingQueue fromBlack,
 4                             BlockingQueue   toBlack) {
 5   Board b = new Board();
 6   while (!board.final()) {
 7     if (!board.initial()) {
 8       Move mBlack = (Move) fromBlack.take(); // blocking
 9       b.update(mBlack);
10       if (board.final()) break;
11     }
12     b.writeTo(System.out);
13     Move mWhite = b.readMoveFrom(System.in);
14     b.update(mWhite);
15     toBlack.put(mWhite);
16 } }

17 public static void runBlack(BlockingQueue fromWhite
18                             BlockingQueue   toWhite) {
19   Board b = new Board();
20   while (!board.final()) {
21     b.analyse(); // long-running call
22     Move mWhite = (Move) fromWhite.take(); // blocking
23     b.update(mWhite);
24     if (board.final()) break;
25     Move mBlack = b.decide();
26     b.update(mBlack);
27     toWhite.put(mBlack);
28 } }

29 public static void main(String[] args) {
30   BlockingQueue q1 = new LinkedBlockingQueue();
31   BlockingQueue q2 = new LinkedBlockingQueue();
32   new Thread(() -> runWhite(q1, q2)).start();
33   new Thread(() -> runBlack(q2, q1)).start();
34 }
```

Fig. 2. Concurrent chess program (gray fragments: turn-taking code)

accordingly; then decides its move based on the preceding analysis (*after* learning about W's move); then updates the board accordingly; then sends its move to W.

Figure 2 shows an implementation of this program in Java.[3] Intuitively, White's turn is between lines 8–15, while Black's turn is between lines 22–27. Notably, Black's board analysis on line 21 can run already during White's turn, concurrently.

The **non-concurrent code** of the program can readily be unit-tested with traditional techniques, because it is adequately structured using traditional abstractions (i.e., classes and methods). For instance, JUnit can be used to test that method `Board.update` updates the board as expected, or that method `Board.readMoveFrom` returns only legal moves.

In contrast, it is problematic to unit-test the **concurrent code** that coordinates W and B toward proper *turn-taking*, simply because—as the gray fragments show—it has not been isolated in a separate module. Because of this lack of structure, turn-taking can be tested only indirectly (e.g., test if W's and B's local copies of the board reach expected configurations after n moves; if they do, then probably turn-taking is fine), but this requires many different schedules to be checked (e.g., re-execute the test case for each interleaving of W and B), and it is too imprecise for debugging (e.g., if the test case fails, it remains unclear if the bug is in the turn-taking or elsewhere).

Turn-taking is an example of a *protocol* among threads. Protocols codify the rules of interaction (i.e., synchronization and communication) that threads must abide by, and they are an essential ingredient of any non-trivial concurrent program (e.g., if turn-taking is implemented incorrectly, the concurrent chess program is fundamentally flawed). Despite the importance of protocols, however, typical GPLs do not provide abstractions to adequately structure concurrent code and isolate protocol implementations in separate modules. Thus, this significant class of functionality cannot be unit-tested effectively, which is problematic: protocols are notoriously hard to get right, while deadlocks and data races continue to plague software engineers [14], so unit-testing is all the more important.

III. PROPOSED SOLUTION

The aim of this project is to enable software engineers to add more structure to concurrent code by developing declarative, high-level abstractions for programming and unit-testing of protocols. The main components to achieve this are: a *domain-specific language* (DSL) to offer the abstractions; a *compiler* to translate code and unit tests from the DSL to the GPL; and an adaptation-based software *model checker* [15] to efficiently execute the resulting unit tests in the GPL.

A. Programmming

The envisioned programming workflow is that software engineers continue to write all code and unit tests in the GPL, *except* all code and unit tests that pertain to synchronization or communication; those should be **initially** written in the DSL, **subsequently** compiled to the GPL, and **finally** integrated with the rest of the concurrent program and unit test suite. To make it practically feasible to really separate the *actions* that threads perform (i.e., computations; GPL) from their *interactions* (i.e., synchronizations and communications; DSL), and to minimize the final integration effort, the workflow must be supported by the programming model as well. It works as follows.

The idea is that every thread runs in an opaque *environment*. Threads are aware of, and can exchange messages with, their environments, but they are oblivious to their environments' contents. Specifically, when threads exchange messages with their environments, they know neither where received messages comes from, nor where sent messages go to. The only thing threads can do, is indicate to their environments **that** they want to interact, but not **when**, **how**, or with **whom**; it is left to the environments to decide which interactions are enacted, in accordance with the protocols (exogenous coordination [16]).

Thus, threads *request* environments to enact interactions, while environments *respond* by enacting interactions among threads. Since these responsibilities are clearly divided in the programming model, their implementations can be clearly separated, too: threads can be implemented in terms of actions (incl. requests) in the GPL, while environments can be implemented in terms of interactions, as protocols, in the DSL.

A notable instance of this programming model is the one where each environment is a *channel* with queue-like behavior. Several modern GPLs support such channel-based concurrency over shared memory (e.g., Go, Rust, Clojure).

---

[3] `Thread`, `BlockingQueue`, `LinkedBlockingQueue`, and `System` are part of the standard Java libraries.

B. Unit-Testing

Following the envisioned programming workflow and programming model, after the final integration effort, the full concurrent program in the GPL consists of three types of modules of code (e.g., classes and methods): *pure action modules*, hand-written, perform only computations; *impure action modules*, hand-written, perform both computations and requests; and *interaction modules*, from-DSL-to-GPL-compiled, perform only synchronizations/computations in response to requests. The first generic observation is that all modules, **including interaction modules that implement protocols**, can be unit-tested. Two more specific observations follow next.

First, the execution of a pure action module in one thread *cannot* affect, nor be affected by, the concurrent execution of any module in another thread (i.e., its result is interleaving-independent); otherwise, it must contain a form of synchronization or communication (e.g., use of shared locks or data), but this is precluded by definition.[4] Notably, this rules out harmful interference that would otherwise cause deadlocks or data races. Thus, unit tests for pure action modules do not need to be re-executed with different schedules; pure action modules are not only structurally separate, but also behaviorally.

Second, and in contrast, the execution of an interaction module in one thread *can* affect, and be affected by, the concurrent executions of impure action modules in other threads (i.e., its result is interleaving-dependent: the order in which requests are made may affect the order in which interactions can be enacted). Thus, unit tests for interaction modules must be re-executed to attain high coverage; interaction modules are structurally separate, but not behaviorally. To facilitate this, the compiler includes a custom model checker in its output: when an interaction unit test is executed, the model checker efficiently verifies that each execution of the interaction module yields the expected result. Importantly, the model checker does *not* verify a formal state space built from abstract DSL code (pre-compilation), but the actual state space built from concrete GPL code (post-compilation). Thus, the code tested in development, is the same code run in production.

IV. FIRST RESULTS

Development of an initial proof-of-concept DSL, compiler, and model checker has started (in progress), to explore the design/implementation space *in a basic setting*; the next section discusses future plans, built upon this preliminary effort. The presentation in this section is example-driven; to save space, formal definitions and other details appear in Sects. A–B.

A. DSL

Inspired by support in several modern GPLs (e.g., Go, Rust, Clojure), the DSL is based on channel-based message-passing between threads; it offers declarative, high-level abstractions for programming and unit-testing of protocols in terms of communications. Figure 3 shows an example.

```
1  protocol Chess {
2    repeat {
3      Move from W to B;
4      Move from B to W;
5    }
6    opt {
7      Move from W to B;
8  } }

9  unittest TurnTaking {
10   []((Move to B) -> X((!(Move to B)) W (Move to W))) /\
11   []((Move to W) -> X((!(Move to W)) W (Move to B)))
12 }
```

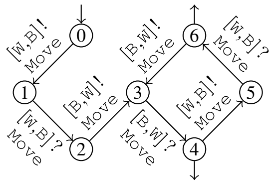

Fig. 3. Chess protocol (syntax; semantics) and turn-taking unit test (syntax)

```
public interface Env {
  Optional       exch(Optional box);
  default void   send(Object m) { exch(Optional.of(m)); }
  default Object recv() {
    return exch(Optional.empty()).get();
} }

public interface Pr {
  Env      env(String threadName);
  String[] threadNames(); // helper method to get all
                          // participating thread names
  Object[] dummies();     // helper method to get a dummy
                          // value of each relevant type
  Pr       deepClone();   // helper method to deep-clone
}
```

Fig. 4. API to interface interaction modules and (impure) action modules

Lines 1–8 implement the chess protocol: it prescribes a finite repetition of (asynchronous, reliable, FIFO-ordered) communications of messages of type `Move` from W to B and from B to W (if Black makes the final move), optionally followed by a communication from W to B (if White makes the final move). Lines 9–12 implement the turn-taking unit test: line 10 asserts that always if a move is sent to Black (White made a move), no next move is sent to Black either until a move is sent to White (Black makes a move) or never (White made the final move); line 11 asserts the symmetric case. The *labeled transition system* (LTS) shows the semantics of the protocol code: labels $[p,q]!t$ and $[p,q]?t$ prescribe the send and receive of a message of type t through the channel from thread p to thread q. Notably, the syntax for protocols is declarative to enhance programmability (it states *what* interactions transpire: "White and Black communicate"), while its semantics is imperative to support compilation (it states *how* interactions transpire: "first White sends to Black; then Black receives from White).

The syntax for protocols is inspired by Scribble [17] and its semantics by Basic Process Algebra with unary iteration; the syntax for unit tests is inspired by Spin [18] and its semantics by Linear Temporal Logic. The calculus that formalizes this initial proof-of-concept DSL appears in Sect. A.

B. Compiler

The behavior of the compiler depends on whether it compiles protocol code or unit test code.

For **protocol code**, the compiler first builds the LTS and then generates a corresponding state machine in Java; the result is an interaction module that implements the protocol in the GPL. The key concern is the interface between this interaction module and the (impure) action modules that use

---

[4] It is a separate issue to ensure/check that an action module is indeed pure.

```
public static void runWhite(Env e) {
  Board b = new Board();
  while (!board.final()) {
    if (!board.initial()) {
      Move mBlack = (Move) e.recv();
      b.update(mBlack);
      if (board.final()) break;
    }
    b.writeTo(System.out);
    Move mWhite = b.readMoveFrom(System.in);
    b.update(mWhite);
    e.send(mWhite);
} }

public static void runBlack(Env e) { ... }

public static void main(String[] args) {
  Pr p = new ChessPr();
  new Thread(() -> runWhite(p.env("W"))).start();
  new Thread(() -> runBlack(p.env("B"))).start();
}
```

Fig. 5. Concurrent chess program, modified (API calls greyed)

```
public class ChessPr implements Pr {
  private volatile int state = 0;
  private Object monitor = this;
  private BlockingQueue q1 = new LinkedBlockingQueue();
  private BlockingQueue q2 = new LinkedBlockingQueue();

  public Env env(String threadName) {
    switch (threadName) {
      case "W": return new Env() {
        public Optional exch(Optional box) {
          synchronized (monitor) {
            while (true) {
              switch (state) {
                case 0: monitor.notifyAll(); state = 1;
                        q1.put(box.get());
                        return new Optional.empty();
                case 1: monitor.wait();
                        break;
                case 2: monitor.wait();
                        break;
                case 3: monitor.notifyAll(); state = 4;
                        Object m = q2.take();
                        return new Optional.of(m);
                ...
      } } } } };
      case "B": return new Env() { ... };
  } }

  public String[] threadNames() {
    return new String[] { "W", "B" }; }
  public Object[] dummies() {
    return new Object[] { new Move() }; }

  public Pr deepClone() { ... }
}
```

Fig. 6. Generated interaction module (for the protocol code in Figure 3)

it. For this, the API shown in Figure 4 is used. It is designed to closely mimic the programming model: interface `Env` represents environments; method `Env.exch` represents requests to enact interactions (*exch*anges with the environment); methods `Env.send` and `Env.recv` are "macros" for common requests; and interface `Pr` represents protocols as sets of environments.

Now, the idea is that: (a) the compiler encapsulates the state machine in an implementation of `Pr`, (b) each thread gets access to its own custom `Env` that implements for each state if/how requests of that particular thread can be responded to, and (c) when `Env.exch` is called, the state machine makes a transition if it can, or the call remains pending and the thread becomes blocked if it cannot. Figures 5–6 show an example.

```
public class ChessPrTest {
  private boolean check(ChessPr p, String formula) { ... }

  @Test // JUnit annotation
  public void testTurnTaking {
    ChessPr p = new ChessPr();
    String formula =
"[]((Move to B) -> X((!(Move to B)) U (Move to W))) /\" +
"[]((Move to W) -> X((!(Move to W)) U (Move to B)))"
    assertTrue(check(p, formula)); // JUnit call
} }
```

Fig. 7. Generated JUnit test case (for the unit test code in Figure 3)

For **unit test code**, the compiler generates a JUnit test case (straightforward to add to an existing JUnit test suite); it uses the model checker to verify the actual LTS built from concrete Java code (i.e., not the formal LTS built from abstract DSL code by the compiler). Figure 7 shows an example.

C. Model Checker

Conceptually, the model checker works in two main steps: (i) build the LTS; (ii) run an algorithm for automata-theoretic model-checking, based on *nested depth-first search* [19].[5] Step (ii) is straightforward to implement; step (i) less so.

The key observation is that at any point in time, a `Pr`-object `p` comprehensively represents a state s in the LTS (i.e., it has a state id and channel contents as its attributes). Moreover, to compute `Pr`-objects for the successors of s, the model checker just needs to deep-clone `p` for each possible send/receive and perform that operation on the clone via an `Env`-object; each time this succeeds, a successor state is discovered. The whole LTS can be generated in this way as `Pr`-objects, noninvasive, by running the same code as the code run in production.[6]

Because interactions are formulated in terms of (finitely many) message types instead of (possibly infinitely many) message values, no further data abstractions are needed; every possible send/receive can be enumerated in finite time.

V. OPEN CHALLENGES

There are two fundamental research challenges. The first one concerns **expressiveness and performance**: to be actually useful in practice, the DSL should support at least parallel composition, parametrization, and data dependencies. Such extensions require research and development of more powerful calculi and new protocol-based reduction techniques to achieve competitive performance of both code and unit tests; here, the main difficulty is the high level of abstraction (i.e., there is a large gap for the compiler to cross from DSL to GPL).

The second challenge concerns **integration-testing**: while unit-testing of protocols as proposed in this paper can guarantee *safety* (i.e., "bad" interactions never transpire: if a request is made, then the response is fine), it cannot guarantee *liveness* (i.e., "good" interactions eventually transpire: requests are made often enough). To guarantee liveness, complementary

---

[5] These steps are often intertwined to yield an *on-the-fly* model-checking approach; it is omitted here for simplicity, but straightforward to implement.

[6] Checking whether a send or receive succeeds is actually non-trivial; the problem and a solution are clarified in Sect. B

integration-testing techniques need to be researched and developed; here, the main difficulty is the fact that liveness pertains not to a single module (cf. safety), but to many. A promising direction is the use of linearity (cf. session types [20]).

REFERENCES

[1] "Agile is the new normal," Hewlett Packard Enterprise, Tech. Rep. 4AA5-7619ENW, 2017.

[2] B. Meyer, *Agile! – The Good, the Hype, and the Ugly*. Springer, 2014.

[3] B. Long, D. Hoffman, and P. A. Strooper, "Tool support for testing concurrent java components," *IEEE Trans. Software Eng.*, vol. 29, no. 6, pp. 555–566, 2003.

[4] W. Pugh and N. Ayewah, "Unit testing concurrent software," in *ASE*. ACM, 2007, pp. 513–516.

[5] K. Coons, S. Burckhardt, and M. Musuvathi, "GAMBIT: effective unit testing for concurrency libraries," in *PPOPP*. ACM, 2010, pp. 15–24.

[6] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov, "Improved multithreaded unit testing," in *SIGSOFT FSE*. ACM, 2011, pp. 223–233.

[7] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code," in *ICSE*, 2012, pp. 727–737.

[8] S. Steenbuck and G. Fraser, "Generating unit tests for concurrent classes," in *ICST*. IEEE Computer Society, 2013, pp. 144–153.

[9] A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey, "P: safe asynchronous event-driven programming," in *PLDI*. ACM, 2013, pp. 321–332.

[10] P. Deligiannis, A. F. Donaldson, J. Ketema, A. Lal, and P. Thomson, "Asynchronous programming, analysis and testing with state machines," in *PLDI*. ACM, 2015, pp. 154–164.

[11] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems," in *OSDI*. USENIX Association, 2014, pp. 399–414.

[12] T. Leesatapornwongsa and H. S. Gunawi, "SAMC: a fast model checker for finding heisenbugs in distributed systems (demo)," in *ISSTA*. ACM, 2015, pp. 423–427.

[13] C. Artho, Q. Gros, G. Rousset, K. Banzai, L. Ma, T. Kitamura, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Model-based API testing of apache zookeeper," in *ICST*. IEEE Computer Society, 2017, pp. 288–298.

[14] S. A. Asadollah, D. Sundmark, S. Eldh, and H. Hansson, "Concurrency bugs in open source software: a case study," *J. Internet Services and Applications*, vol. 8, no. 1, pp. 4:1–4:15, 2017.

[15] P. Godefroid and K. Sen, "Combining model checking and testing," in *Handbook of Model Checking*. Springer, 2018, pp. 613–649.

[16] K. Lau, P. V. Elizondo, and Z. Wang, "Exogenous connectors for software components," in *CBSE*, ser. Lecture Notes in Computer Science, vol. 3489. Springer, 2005, pp. 90–106.

[17] R. Hu and N. Yoshida, "Hybrid session verification through endpoint API generation," in *FASE*, ser. Lecture Notes in Computer Science, vol. 9633. Springer, 2016, pp. 401–418.

[18] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[19] R. Gerth, D. A. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *PSTV*, ser. IFIP Conference Proceedings, vol. 38. Chapman & Hall, 1995, pp. 3–18.

[20] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," *J. ACM*, vol. 63, no. 1, pp. 9:1–9:67, 2016.

[21] J. C. M. Baeten, F. Corradini, and C. Grabmayer, "A characterization of regular expressions under bisimulation," *J. ACM*, vol. 54, no. 2, p. 6, 2007.

[22] A. Pnueli, "The temporal logic of programs," in *FOCS*, 1977, pp. 46–57.

$$\frac{}{\alpha \xrightarrow{\alpha} 1} \qquad \frac{}{1\downarrow}$$

$$\frac{P_i \xrightarrow{\alpha} P' \textbf{ and } i \in \{1,2\}}{(P_1+P_2)\xrightarrow{\alpha} P'} \qquad \frac{P_i\downarrow \textbf{ and } i\in\{1,2\}}{(P_1+P_2)\downarrow}$$

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{(P_1\cdot P_2)\xrightarrow{\alpha}(P_1'\cdot P_2)} \qquad \frac{P_1\downarrow \textbf{ and } P_2\xrightarrow{\alpha}P_2'}{(P_1\cdot P_2)\xrightarrow{\alpha}P_2'} \qquad \frac{P_1\downarrow \textbf{ and } P_2\downarrow}{(P_1\cdot P_2)\downarrow}$$

$$\frac{P\xrightarrow{\alpha}P'}{P^*\xrightarrow{\alpha}P'\cdot P^*} \qquad \frac{}{P^*\downarrow}$$

Fig. 8. Transition relation for protocols

$$\frac{}{\mathcal{P},i\models\top} \qquad \frac{\alpha=\beta \textbf{ for all } \mathcal{P}(i)\xrightarrow{\beta}\mathcal{P}(i{+}1)}{\mathcal{P},i\models\alpha} \qquad \frac{\textbf{not } \mathcal{P},i\models T}{\mathcal{P},i\models\neg T}$$

$$\frac{\mathcal{P},i\models T_1 \textbf{ or } \mathcal{P},i\models T_2}{\mathcal{P},i\models T_1\vee T_2} \qquad \frac{\mathcal{P},i{+}1\models T}{\mathcal{P},i\models \mathsf{X}T}$$

$$\frac{\big[[\mathcal{P},j\models T_1 \textbf{ for all } i\le j<n] \textbf{ and } \mathcal{P},n\models T_2\big] \textbf{ for some } n\ge i}{\mathcal{P},i\models T_1\,\mathsf{U}\,T_2}$$

Fig. 9. Satisfaction relation for unit tests

APPENDIX A
DSL

The calculus that formalizes the DSL is defined as follows. Let $\mathbb{P}$ denote a set of *thread names*, ranged over by p, q, r. Let $\mathbb{T}$ denote a set of *message types*, ranged over by t. Let $\mathbb{P}\text{ROT}$ and $\mathbb{T}\text{EST}$ denote the sets of *protocols* and *unit tests*, ranged over by P and T, generated by the following grammar:

$$\begin{aligned} \alpha &::= [p,q]!t \mid [p,q]?t \\ P &::= 0 \mid 1 \mid \alpha \mid P_1+P_2 \mid P_1\cdot P_2 \mid P^* \\ T &::= \alpha \mid \neg T \mid T_1\vee T_2 \mid \mathsf{X}T \mid T_1\,\mathsf{U}\,T_2 \mid \top \end{aligned}$$

Actions $[p,q]!t$ and $[p,q]?t$ represent the send/receive of a message typed t through the channel from thread p to thread q.

Protocols are essentially $\text{BPA}^*_{0,1}$ processes [21] over sends and receives: informally, protocol 0 prescribes deadlock; protocol 1 prescribes skip; protocol α prescribes a send or receive; protocols P_1+P_2 and $P_1\cdot P_2$ prescribe the alternative and sequential composition of P_1 and P_2; protocol P^* prescribes a finite iteration of P. Formally, $P\xrightarrow{\alpha}P'$ means P performs α and makes a *transition* to P', while $P\downarrow$ means P *terminates*; they are defined as the smallest relations induced by the rules in Figure 8. *Run* $\mathcal{P}$ of protocol P is a sequence $\mathcal{P}=P_0P_1P_2\cdots P_n$ such that $P=P_0$, and $P_i\xrightarrow{\alpha_i}P_{i+1}$ for all $0\le i<n$, and $P_n\downarrow$, for some $\alpha_0,\ldots,\alpha_{n-1}$; each P_i is a *state* of P, denoted as $\mathcal{P}(i)$.

Unit tests are essentially LTL formulas [22] over sends and receives, and their meaning is defined relative to run $\mathcal{P}$ and time index i: informally, test α asserts that next state $\mathcal{P}[i{+}1]$ is reached from current state $\mathcal{P}[i]$ by performing α; tests $\top$, $\neg T$, and $T_1\vee T_2$ assert tautology, the negation of T, and the disjunction of T_1 and T_2 in the current state; test $\mathsf{X}T$ asserts that T holds in the next state; test $T_1\,\mathsf{U}\,T_2$ asserts that T_1 holds until T_2 holds. In particular, $\top\,\mathsf{U}\,T$ asserts that T *eventually* holds, while $\neg(\top\,\mathsf{U}\,\neg T)$ asserts that T *always* holds. Formally, $\mathcal{P},i\models T$ means test T holds on run $\mathcal{P}$ at time index i; it is defined as the smallest relation induced by the rules in Figure 9. Run $\mathcal{P}$ *passes* test T iff $\mathcal{P},0\models T$; protocol P passes T if each of P's runs passes T. A run that does not pass T is a *counterexample*.

TABLE I
FROM DSL TO CALCULUS (PROTOCOLS)

DSL	Calculus
t `from` p `to` q `;`	$[p,q]!t \cdot [p,q]?t$
`{` P_1 `}` `or` `{` P_2 `}`	$P_1 + P_2$
P_1 P_2	$P_1 \cdot P_2$
`loop` `{` P `}`	P^*
`opt` `{` P_1 `}` P_2	$(P_1 \cdot P_2) + P_2$
P_1 `opt` `{` P_2 `}`	$P_1 + (P_1 \cdot P_2)$

TABLE II
FROM DSL TO CALCULUS (UNIT TESTS)

DSL	Calculus
t `from` p	$[p,r_1]!t \vee \cdots \vee [p,r_n]!t$
t `to` q	$[r_1,q]?t \vee \cdots \vee [r_n,q]?t$
`!` T	$\neg T$
T_1 `\/` T_2	$T_1 \vee T_2$
T_1 `/\` T_2	$\neg(\neg T_1 \vee \neg T_2)$
T_1 `->` T_2	$\neg T_1 \vee T_2$
`X` T	$\mathbf{X}\, T$
T_1 `U` T_2	$T_1 \,\mathbf{U}\, T_2$
T_1 `W` T_2	$(T_1 \,\mathbf{U}\, T_2) \vee (\neg(\top \,\mathbf{U}\, \neg T_1))$
T_1 `V` T_2	$\neg(\neg T_1 \,\mathbf{U}\, \neg T_2)$
`<>` T	$\top \,\mathbf{U}\, T$
`[]` T	$\neg(\top \,\mathbf{U}\, \neg T)$

Tables I–II show the mapping from protocols and unit tests in the DSL to those in the calculus. In Table II, cases t `from` p and t `to` q, thread names $r_1, \ldots, r_n$ are those that occur in the protocol for which the unit test is written.

APPENDIX B
MODEL CHECKER

The non-trivial bit of building the actual LTS from the concrete Java code is *detecting* whether a send or receive has succeeded. The problem is that methods `send` and `recv` return only once the underlying `put` or `take` (on the internal message queue) is actually done: if the `Pr`-object does not permit this (because it would violate the protocol), the thread becomes blocked and basically gets stuck. One way to solve this, is to add non-blocking send/receive methods to the generated state machine code for the sole purpose of model-checking. However, this would violate the principle of building the LTS by running the same code as the code run in production.

Instead, to detect if a send or receive has succeeded, the code in Figure 10 is used. It works as follows for sends (lines 8-24); it works similarly for receives. For each thread name and dummy value of a relevant type (i.e., a type that occurs in the protocol code in the DSL), a deep clone is created of the `Pr`-object, as a tentative successor. Then, the current thread (`curThread`) acquires a lock and spawns an auxiliary thread (`auxThread`) that tries to acquire the same lock. Then, the current thread performs a send, and two things can happen:

- If the send is permitted, the underlying `put` is performed, so the send succeeds, and a successor is found (and added to the list). The current thread then releases the lock, and waits until the auxiliary thread has terminated. The auxiliary thread can then acquire the lock and interrupts the current thread; however, this interrupt can safely be ignored, because the current thread already knows the send has succeeded (and added the successor to the list).
- If the send is not permitted, the underlying `put` is not performed. Instead, inside method `send` (line 18), the current thread calls `wait` on a monitor backed *by the same lock* that it already acquired previously. Thus, the lock is now free for the auxiliary thread to acquire, and after having done so, the auxiliary thread interrupts the current thread. The current thread accordingly unblocks, catches the corresponding exception, and then knows that the send was not permitted and will never succeed. Accordingly, it does not add `successor` to the list.

```
1  public class ModelChecker {
2    ...
3    public static List<Pr> getSuccessors(Pr p) {
4      List<Pr> successors = new ArrayList<>();
5      for (String r : p.threadNames()) {
6        // try sends:
7        for (Object d : p.dummies()) {
8          Pr successor = p.deepClone();
9          synchronized (successor) {
10           Thread curThread = Thread.currentThread();
11           Thread auxThread = new Thread(() -> {
12             synchronized (successor) {
13               thread.interrupt();
14             }
15           });
16           auxThread.start();
17           try {
18             successor.env(r).send(d);
19             successors.add(successor);
20           } catch (InterruptedException e) {}
21         }
22         try {
23           auxThread.join();
24         } catch (InterruptedException e) {}
25       } }
26       // try receives (similar to lines 8-24):
27       ...
28     }
29     return successors;
30 } }
```

Fig. 10. Method to generate successors