Prut4j: Protocol Unit Testing fo(u)r Java

Florian Joost Slob* and Sung-Shik Jongmans*†
*Department of Computer Science, Open University of the Netherlands
†Centrum Wiskunde & Informatica (CWI), Netherlands Foundation of Scientific Research Institutes (NWO-I)

***Abstract*—This paper presents Prut4j: a tool to simplify unit testing of channel/queue-based communication protocols in concurrent Java programs. Prut4j offers two domain-specific languages to write, compile (to Java), and execute (with JUnit) high-level "protocol modules" and accompanying unit tests. Our first evaluation provides evidence for Prut4j's expressiveness (network topologies, games, scientific kernels) and efficiency (Prut4j-based programs perform well in a third-party benchmark).**

***Index Terms*—concurrency, communication, DSLs**

I. Introduction

Background: To take advantage of modern multi-core processors, concurrent programming with shared memory—a notoriously hard enterprise—is becoming increasingly important. To alleviate some of the complexities, besides low-level *synchronisation primitives*, many languages now offer core support for higher-level *communication primitives* as well, in the guise of message passing through asynchronous, reliable, order-preserving "channels" (e.g., Go, Clojure) or "queues" (e.g.,. Java, C#). The idea is that, beyond their usage in distributed computing, channels/queues can also serve as a *programming abstraction* for shared memory, supposedly less prone to concurrency bugs than locks, semaphores, etc.

However, evidence suggests that channel/queue-based programming abstractions have their issues, too. For instance, a study of 171 concurrency bugs in major open source programs shows: "message passing does not necessarily make multi-threaded programs less error-prone than shared memory" [1].

Our vision: The long-term aim of the Prut4j project [2] is to simplify *unit testing* of concurrent programs that use channel/queue-based programming abstractions.

Motivating example: Imagine that we need to write a concurrent chess program that consists of threads White (human) and Black (computer), plus two channels/queues. Every thread has a local copy of the board and runs a loop. In each iteration:

- White receives a move from Black; then updates (its local copy of) the board; then writes the board to the terminal; then reads the next move from the terminal; then updates the board; then sends its move to Black.
- Black pre-analyses (its local copy of) the board for possible next moves (*before* knowing White's move); then receives a move from White; then updates the board; then selects a move based on the pre-analysis (*after* knowing); then updates the board; then sends its move to White.

Funded by the Netherlands Organisation of Scientific Research (NWO): 016.Veni.192.103. This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

```
1  public static BlockingQueue wb = ...;
2  public static BlockingQueue bw = ...;

4  public static void runWhite() {
5    Board b = new Board();
6    while (!board.final()) {
7      if (!board.initial()) {
8        Move mBlack = (Move) bw.take();
9        b.update(mBlack);
10       if (board.final()) break; }
11     b.writeTo(System.out);
12     Move mWhite = b.readMoveFrom(System.in);
13     b.update(mWhite);
14     wb.put(mWhite); } }

16 public static void runBlack() {
17   Board b = new Board();
18   while (!board.final()) {
19     b.pre_analyse(); // long-running method call
20     Move mWhite = (Move) wb.take();
21     b.update(mWhite);
22     if (board.final()) break;
23     Move mBlack = b.decide();
24     b.update(mBlack);
25     bw.put(mBlack); } }

27 public static void main(String[] args) {
28   new Thread(() -> runWhite()).start();
29   new Thread(() -> runBlack()).start(); }
```

Note: `Board` and `Move` are custom Java classes (details unimportant); `Thread`, `BlockingQueue`, and `System` are Java classes in the standard library.

Fig. 1. Concurrent chess program (highlights: turn-taking protocol)

Fig. 1 shows a Java version of the program: White's turn is on lines 8–14, while Black's turn is on lines 20–25. We note that Black's pre-analysis can run already *during* White's turn.

The highlighted method calls in Fig. 1 indicate communication primitives for sending (`put`) and receiving (`get`) of messages through channels/queues (`BlockingQueue`). Collectively, they realise the *turn taking protocol* between White and Black. This protocol is crucial: if its code is buggy (e.g., White takes two turns in a row), the whole program is useless (i.e., it does not simulate chess). Therefore, it seems clear that we *must* unit test the protocol code; moreover, since it forms a logically distinct piece of functionality, this *should* be easy.

However, it is not: the protocol code has not been isolated in a separate module. As a result, it can be tested only indirectly (e.g., test if White's and Black's local copies of the board reach expected configurations after some moves; if so, then *probably* turn-taking is fine), but this has two disadvantages:

- It requires many thread interleavings to be investigated (time-consuming and unsupported by mainstream tools).
- It is too imprecise for debugging (if the test fails, it is unclear if the bug is in the protocol code or elsewhere).

Sadly, non-isolation is common: in fact, the protocol code in Fig. 1 even has an *idiomatic* producer–consumer structure [3].

Our contribution: Prut4j offers a novel approach to write, compile, and execute high-level "protocol modules" and accompanying unit tests to avoid the listed disadvantages.

Organisation: In Sect. II, we present an overview of Prut4j. In Sect. III, we present design/implementation details. In Sect. IV, we evaluate Prut4j. In Sect. V, we conclude with lessons learned, related work, and future work. Prut4j is available at https://github.com/prut4j.

II. Overview of Prut4j

A. Foundations, Workflow, and Scope

Our research is founded on the following two observations:

- It is crucial to isolate protocol code in separate modules. Otherwise, protocol code simply cannot be unit tested.
- **In theory**, mainstream *general-purpose languages* (GPL) and testing frameworks are expressive enough to write "protocol modules" and accompanying unit tests. **In practice**, however, it requires too much effort and discipline to really do this (Sect. II-B, final paragraph)

Prut4j is a research prototype to investigate if *domain-specific languages* (DSL) can help to address this issue.

Regarding workflow, the general approach is that the developer continues to write *most* code of their concurrent programs and unit tests in a mainstream GPL and testing framework. There is one exception: protocol modules and accompanying unit tests. Instead of writing these in a low-level GPL (this paper: Java), they are written in two higher-level DSLs (this paper: *Discourje* [4] and *Owl* [5]); tooling for these DSLs (this paper: Prut4j) is subsequently responsible for ensuring that the DSL code can be effortlessly integrated into the GPL code.

We currently consider Prut4j a scientific proof-of-concept tool to investigate the feasibility of our approach in terms of **expressiveness** and **efficiency**; our evaluation provides first evidence (Sect. IV). We have many ideas for improvements (Sect. V), but their development is beyond this paper's scope.

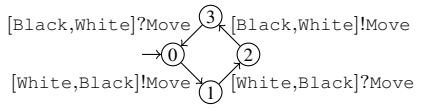
B. Motivating Example, Revisited

To demonstrate the workflow/usage of Prut4j, we revisit the concurrent chess program (Sect. I): first, we present the turn taking protocol in Discourje; then, we present a selection of unit tests in Owl; finally, we show how the two are integrated into the program in Java. (The first two steps are actually independent.) Here, we cover only the minimum material to explain the basic workings of Prut4j; Sect. III contains details.

Turn taking protocol in Discourje: Discourje is a DSL for channel/queue-based protocols, based on *Lisp*.

Fig. 2 (left) shows the turn taking protocol in Discourje. It consists of a parametrised loop. The parameters of the loop, `p` and `q`, store thread names and are initialised to `"White"` and `"Black"`: at the start of an iteration, `p` and `q` indicate the active and passive player; at the end, a next iteration is initiated with the names swapped. Furthermore, the body of the loop consists of a single communication, namely the send *and* subsequent receive of a message of type `Move` through the channel/queue from `p` to `q` (= passing the turn).

```
1 (loop [p "White"
2        q "Black"]
3   (-->> Move p q)
4   (recur q p))
```



Note: Transition label $[p,q]\dagger m$ indicates a send (if $\dagger$=!) or receive (if $\dagger$=?) of a message of type m through the channel/queue from thread p to thread q.

Fig. 2. Turn taking protocol in Discourje (left) and state machine (right)

```
1 public interface Pr {
2   Optional exch(String threadName, Optional box);
3   default void send(String threadName, Object message) {
4     exch(threadName, Optional.of(m)); }
5   default Object recv(String threadName) {
6     return exch(threadName, Optional.empty()).get(); } }
```

Note: `send` and `recv` are just wrappers for special usages of `exch`.

Fig. 3. API

The main Discourje component of Prut4j is a *compiler*: it consumes as input a protocol in Discourje; it produces as output a Java class that implements interface `Pr` in Fig. 3.

The idea is that a `Pr` class encapsulates a protocol. That is: when a thread is ready to send/receive, instead of calling `put`/`take` of `BlockingQueue`, it calls `send`/`recv` of `Pr`. The crucial point is that the caller indicates only that it intends to communicate, but not when, how, or with whom; these decisions are made inside the `Pr` class (senders/receivers do not know whereto/wherefrom messages go/come; `send`/`recv` have no such parameters). The caller remains blocked until its "request" to send/receive has eventually been "granted". In this way, no protocol logic resides with the caller any more.

Fig. 4 shows the Java class that Prut4j's compiler generates for Fig. 2. First, we note that the queues are now encapsulated as private attributes of class `TurnTakingPr`; this means that White and Black can no longer call `put`/`take` *directly*, but only *indirectly* via a `TurnTakingPr` object. Second, we note that method `exch` (on which `send` and `recv` rely; Fig. 3) effectively runs the *state machine* in Fig. 2 (right): when `exch` is called, depending on the state, the request is either immediately granted (and a state change is effectuated), or not.

For instance (lines 9–12), if the state machine is in state 0, and White calls `send`, then this request is immediately granted (cf. Fig. 2): the message-to-send is put into a queue, the state is changed, Black is notified (if Black is waiting), and an empty container is returned (because `send` was called; if `recv` were called, the container would contain a message-to-receive). In contrast (line 17), if the state machine is *not* in state 0 or 3, then no request of White can be granted in the current state (cf. Fig. 2). Instead, White is suspended (`wait`) and will resume only when Black effectuates a state change (`notifyAll`).

To summarise: the developer writes Fig. 2 (left), and then, Prut4j's compiler internally constructs the corresponding state machine in Fig. 2 (right) and generates Fig. 4.

Unit tests in Owl: Many protocol requirements can be naturally expressed as rules of the form: if *these* sends/receives have happened (or not), then *those* sends/receives are enabled (or not). The antecedent and consequent of such rules can be interpreted as the input and expected output of unit tests. For

```
1  public class TurnTakingPr implements Pr {
2    private BlockingQueue bw = ..., wb = ...;
3    private volatile int state = 0;

5    public exch(String threadName, Optional box) {
6      switch (threadName) {
7        case "White": synchronized (this) { while (true) {
8          switch (state) {
9            case 0: if (box.isPresent()) { // send
10                     wb.put(box.get());
11                     state = 1; notifyAll();
12                     return new Optional.empty(); }
13           case 3: if (box.isEmpty()) { // receive
14                     Object m = bw.take();
15                     state = 0; notifyAll();
16                     return new Optional.of(m); } }
17         wait(); } }
18       case "Black": ... } }
```

Fig. 4. Turn taking protocol in Java ≥8, generated for Fig. 2

```
1 !"Black SEND Move"                                  // test1.owl
2 !"Black SEND Move" U "Black RECV Move"              // test2.owl
3 G("Black SEND Move" =>                              // test3.owl
4   X(!"Black SEND Move" U "Black RECV Move"))        // test3.owl
```

Fig. 5. Selection of unit tests in Owl

instance: "if Black has not received from White, then Black cannot send to White" (otherwise, Black passes the turn back to White, but it is not Black's turn). Owl is a DSL to concisely write such requirements, based on *temporal logic*.

Fig. 5 shows a selection of unit tests in Owl, stored in files `test[1,2,3].owl`. The unit test on line 1 states that initially, Black can*not* (`!`) send a message of type `Move`. The unit test on line 2 states that Black cannot send *until* (`U`) Black receives; this is equivalent to the if–then example, above. The unit test on lines 3–4 states that *always* (`G`), *if* Black sends (`=>`), *then next* (`X`), it cannot send again until it has received.

The main Owl component of Prut4j is an *execution engine*: it consumes as input a unit test in Owl, a `Pr` object, and "dummies"; it produces as output a `true`/`false` verdict.

The idea is that the engine inspects *every* possible run of the `Pr` object, to see if the unit test in Owl succeeds or fails; to obtain these runs, the engine basically makes sequences of `exch` calls on the `Pr` object, using the dummies as messages-to-send. However, the engine does not naively execute the `Pr` object every time from scratch. Instead, first, the engine *re-constructs* the state machine of the "initial" `Pr` object as a set of "successor" `Pr` objects, organised in a graph. Then, it inspects the whole re-constructed state machine, using a *model checking* algorithm. The advantage is that the `Pr` object is executed every time only from the point where different runs start to diverge; their shared "history" is executed only once.

Fig. 6 shows a JUnit class that the developer can write for Fig. 5. First, we note that we use only one dummy in this example: every time a message is sent, it will just be a default `Move` object. In general, the array of dummies can consist of any number of objects, of any type; the engine will randomly pick a dummy of the right type, by need. Second, we note that the engine can be accessed through class `Engine`.

To summarise: the developer writes Figs. 5 and 6, and then, Prut4j's execution engine re-constructs the state machine of the `Pr` object and inspects it.

```
1  public class TurnTakingPrTest {
2    private Object[] dummies = new Object[] { new Move() };
3    private Pr p;

5    @BeforeEach public void init() {
6      p = new TurnTakingPr(); }
7    @Test public void testBlackCannotSend() {
8      assertTrue(Engine.exec("test1.owl", p, dummies)); }
9    @Test public void testBlackCannotSendUntil() {
10     assertTrue(Engine.exec("test2.owl", p, dummies)); }
11   @Test public void testBlackCannotSendAgainUntil() {
12     assertTrue(Engine.exec("test3.owl", p, dummies)); } }
```

Fig. 6. Selection of unit tests in JUnit ≥5, using Fig. 5

```
1   @Test public void testBlackCannotSendUntil_1() {
2     p.send("White", dummies[0]);
3     synchronized (p) {
4       new Thread(() -> {
5         synchronized (p) { p.interrupt(); } }
6       ).start();
7       assertThrows(InterruptedException.class,
8         () -> p.send("Black", dummies[0])); } }
9   @Test public void testBlackCannotSendUntil_2() {
10    p.send("White", dummies[0]);
11    p.recv("Black");
12    p.send("Black", dummies[0]); }
13  ...
```

Fig. 7. Selection of unit tests in JUnit ≥5, *not* using Fig. 5 (cf. Fig. 6)

Integration: Integrating the `TurnTakingPr` class (Fig. 4 into the program requires little effort. We can simply refactor Fig. 1 by replacing the queues with a `TurnTakingPr` object, and the `put`/`take` calls with `send`/`recv` calls. Of course, *new* programs can be written against the `Pr` interface from the start; in that case, no additional refactoring or integration effort is needed. The same holds for Fig. 6: it already uses JUnit, so it can be directly incorporated in an existing JUnit suite.

Finally, we note that it is surely possible to *manually* apply our approach to write protocol modules and accompanying unit tests (e.g., using object-oriented refactoring techniques). For instance, Fig. 4 contains standard Java code, so it can be written by hand as well. Moreover, Fig. 7 shows an *incomplete set* of hand-written unit tests for the same property as `test2.owl` in Figs. 5 and 6. However, it requires considerable effort and discipline to write all this by hand: state machines (e.g., Fig. 4) tend to be verbose and laborious to program as they get larger, while many test methods tend to be needed (e.g., `testBlackCannotSendUntil_[1,2,...]` in Fig. 7) to attain the same "maximal" coverage as Prut4j's execution engine. Prut4j takes away much of the effort.

III. Design and Implementation Details

A. Core Components

Fig. 8 shows the core components of Prut4j and the data flow among them. The two main components are the compiler for Discourje and the execution engine for Owl.

Briefly, the compiler for Discourje works as follows:

- The **input** is a protocol in Discourje. Specifically, the supported syntax is a subset of *Clojure* (= Lisp on the JVM), plus macros for protocols (e.g., `-->>` in Fig. 2).
- The **output** is a Java class that implements interface `Pr`.

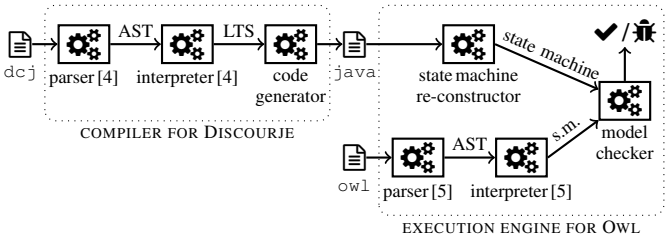


Fig. 8. Architecture

- The **parser**, written in Clojure, converts Discourje code into an internal *abstract syntax tree* (AST), by taking advantage of Clojure's own macro expansion mechanism.
- The **interpreter**, written in Clojure, converts an AST into a *labelled transition system* (LTS), which is a state machine-like data structure. The conversion is based on Discourje's formal semantics, which mathematically defines possible transitions for every operator of Discourje.
- The **code generator**, written in Java, converts an LTS into Java code (e.g., Fig. 4).

The parser and interpreter for Discourje were originally developed as part of a library for *runtime verification* in Clojure [4]; we refer to that paper for details (including formal semantics). In contrast, the code generator is implemented from scratch.

Briefly, the execution engine for Owl works as follows:

- The **input** is a `Pr` class and a unit test in Owl (plus configuration data, such as dummies; omitted in Fig. 8).
- The **output** is a verdict: the unit test succeeds for every run of the `Pr` class (✔), or a bug is reported (🐞) and the engine gives diagnostics information about a failing run.
- The **state machine re-constructor**, written in Java, creates a graph data structure that represents *the same* state machine as the LTS in the compiler. Specifically, every vertex v is a `Pr` object, while every edge (v, v') represents a `send`/`recv` call on v, resulting in v'. Thus, every path $v_1 v_2 \cdots v_n$ through the graph represents a run of v_1.
  To build the graph, first, the re-constructor creates a fresh object of the input `Pr` class; this is the "initial" vertex v_{init}. Then, every possible `send`/`recv` call is made on (deep-)clones of v_{init}; these calls result in the "successor" vertices of v_{init}. The graph is iteratively expanded by computing successors of successors, of successors, etc.
- The **parser** and **interpreter**, written in Java, convert a piece of Owl code into an internal AST, and an AST into a state machine that represents *every* protocol run that is allowed according to the Owl code. The supported syntax is based on *linear temporal logic* (LTL) [6]. This is a form of logic to specify dynamic behaviour over time.
- The **model checker**, written in Java, compares the two state machines: it checks if every run of the re-constructed state machine is also a run of the state machine for the Owl code; if so, the run satisfies the LTL property and passes the unit test. To do this efficiently, we implemented an algorithm based on *nested depth-first search* [7], using *on-the-fly* techniques [8].

The parser and interpreter for Owl come from an advanced library for temporal logic [5]. In contrast, the state machine re-constructor and model checker are implemented from scratch.

We note that to execute unit tests in Owl, alternatively, it seems possible to inspect the LTS instead of the re-constructed state machine; this would save re-construction work. However, there is a decisive reason why *not* to do this: the LTS is not the code that is executed, so we would be testing the wrong artefact. Specifically, if the code generator produces a `Pr` class that differs from the LTS—intentionally (optimisations) or otherwise (bugs)—then it may fail *even* when used in "successfully" unit tested conditions. To avoid this, when unit testing, it is crucial to re-construct the state machine with *exactly the same code* as the code that is normally executed.

B. Key Challenges

We discuss two key implementation challenges of Prut4j: scalability of generated code and computation of successors.

Scalability of generated code: The naive approach to generate a `Pr` class is to let method `exch` simulate *a single* state machine, as in Fig. 4. However, in this approach, threads that call `send`/`recv` require tight synchronisation to ensure that only one thread can effectuate a state change at a time. This is what **`synchronized`** `(`**`this`**`) { ... }` in Java achieves, as on line 7. However, for larger numbers of threads, the need to synchronise has an increasingly negative impact on scalability (waiting times increase super-linearly), so this is inefficient.

To address this issue, we implemented an optimisation that uses *multiple* state machines, instead of a single one. The idea is that we can *decompose* the original "large" state machine into thread-specific "small' ones, such that every small state machine has transitions only for one thread (e.g., [9]). In this approach, method `exch` simulates all small state machines simultaneously: whenever a thread calls `exch`, it accesses and mutates only its own small state machine without affecting the other ones. Thus, no synchronisation among threads is needed.

However, not every state machine can be decomposed while fully preserving its semantics (e.g., [9]), which *could* be a problem. This is why state machine re-construction is so important in Prut4j: even when the semantics of the large state machine is not fully preserved, because of re-construction, the engine always executes unit tests on the actual code. Thus, if the unit tests succeed, then the decomposition optimisation *is* semantics-preserving at least for the scenarios of interest.[1]

Computation of successors: As explained, to compute successors of v (a `Pr` object), every possible `send`/`recv` call is made on (deep-)clones of v. To enumerate all possible `send`/`recv` calls, the state machine re-constructor uses additional configuration data, namely thread names and dummies; it simply calls `send`/`recv` with every combination of them.

The key challenge is *detecting* whether such a call succeeds. Specifically, a problem occurs when the requested send/receive

[1] If a unit test fails, then the decomposition optimisation either is semantics-preserving (and the original state machine does not satisfy the tested property), or it is not semantics-preserving. In the latter case, the original state machine might actually satisfy the tested property.

is *not* enabled. In that case, the underlying `exch` method blocks the caller. However, if there are no other threads to effectuate a state change, the caller will never unblock, so the re-constructor indefinitely gets stuck. A naive way to solve this, is to let a second thread concurrently perform another `exch` call, in an attempt to effectuate a state change and unblock the first thread. However, non-deterministic scheduling make this approach problematic as well.

Our solution is more advanced: it uses a *main thread*, an *auxiliary thread*, and an *intrinsic lock* (via Java's `synchronized` keyword). The idea is that the main thread acquires the lock, spawns the auxiliary thread, and calls `send`/`recv`. The auxiliary thread only tries to acquire the lock; when successful, it *interrupts* the main thread. Now, there are two scenarios:

- If the requested send/receive is enabled, the request is immediately granted. When the call subsequently returns (without blocking), the main thread releases the lock and waits until the auxiliary thread terminates. The auxiliary thread then acquires the lock and interrupts the main thread; this interrupt can be safely ignored.
- If the requested send/receive is *not* enabled, the caller becomes blocked: it calls `wait` and *temporarily releases the lock* (Java's intrinsic locks work this way). The auxiliary thread then acquires the lock and interrupts the main thread. Subsequently, the main thread's `wait` call abnormally terminates with an `InterruptedException`; this exception can be caught and indicates that the requested send/receive is not enabled in the current state (and does not produce a successor of v).

Fig. 7, lines 1–8, shows a simplified version of this solution.

We note that the alternative of adding a non-blocking version of `exch` to the generated code for the *sole* purpose of state machine re-construction might work and be simpler, but is unsatisfactory: it violates the important principle to unit test exactly the same code as the code that is normally executed.

IV. Evaluation

A. Expressiveness (Case Studies)

As noted in Sect. II-A, our current efforts are geared towards studying the feasibility of our approach along two dimensions: expressiveness and efficiency. Regarding expressiveness, we have conducted a first set of case studies to see if a broad range of different programs and protocols are supported by Prut4j. Specifically, we wrote 12 protocol modules and 125 accompanying unit tests in Discourje and Owl to investigate different aspects in three categories: network topologies, games, and scientific kernels.

Network topologies: The aim of this category is to investigate the *distinguishing power* of unit tests in Owl. To this end, we wrote protocol modules in Discourje for a distributed *token algorithm*, for six common network topologies, parametrised in the number of threads n: directed ring, undirected ring, star, binary tree, full mesh, and 2D mesh. For $n{=}4$, every network is unique, so the generated `Pr` classes are different.

We succeeded to write a suite of distinguishing unit tests in Owl: every generated `Pr` class passes a unique combination.

Games: The aim of this category is to investigate the ability to write unit test in Owl for protocols with complex control flow. To this end, we wrote protocol modules in Discourje for two more games, parametrised in the number of threads n ("players"): rock–paper–scissors and go fish. The protocols for these games are substantially more complicated than chess (Sect. II-B). Notably, with rock–paper–scissors, the number of threads that participate in the protocol *decreases over time*; with go fish, the order in which threads take turns is *not statically fixed*, but dynamically determined while playing.

We successfully wrote unit tests in Owl for *generic* properties (e.g., boundedness of channels/queues; every sent message is eventually received; liveness) and *specific* ones (e.g., for rock–paper-scissors, "loser threads" drop out; for go fish, if a thread does not have an asked card, it takes the next turn).

Scientific kernels: The aim of this category is to investigate the ability to use Prut4j in real, existing programs. To this end, we adapted four scientific kernels (in computational fluid dynamics) that are part of *NPB* [10], a third-party benchmark suite designed to evaluate the efficiency of parallel computing architectures. Specifically, we took NPB's reference programs, replaced all existing non-modular protocol code with `Pr` interface calls, wrote equivalent protocols in Discourje, and simply integrated the generated `Pr` classes. Also, we wrote unit tests in Owl and successfully executed them.

To our knowledge, this is the first time that channel/queue-based communication protocols in scientific kernels have been unit tested. We also note that computational science is an area where unit testing of protocols can be particularly important, because: programs in computational science are often concurrent, while publications in prestigious journals (including Science and PNAS) have been retracted because of bugs [11].

B. Efficiency (Benchmarks)

We recognise two possible efficiency concerns with Prut4j.

The first one is about *executing unit tests*: some state machines grow exponentially large in the number of threads (especially in larger programs with higher complexities), so executing unit tests based on state machines could turn out to be prohibitively slow. However, it seems reasonably realistic to assume that protocols are unit tested with 2, 3, or 4 threads, but not hundreds ("if it works for $2{\leq}n{\leq}4$, it should work for $n{>}4$"); with such low numbers, execution times seem fine.[2]

The second possible efficiency concern is about *running generated code*: even if it is sufficient to unit test with low numbers of threads, programs are likely to be run with high(er) numbers. As our approach adds an extra layer of abstraction to the program, we may face considerable overhead.

To investigate such possible overhead, we used the four scientific kernels of NPB (Sect. IV-A) in a quantitative experiment. Specifically, using standardised input (part of the NPB

[2]Anecdotally, the mean execution time of the 125 unit tests of Sect. IV-A is roughly 61 seconds per test, on an ordinary dev machine. We believe this is acceptable, given that: **(1)** Prut4j achieves "maximal" coverage (every possible run of the protocol is inspected); **(2)** protocols are complex functionality; **(3)** there is still substantial room for optimisations (e.g., caching techniques); **(4)** protocols cannot conveniently be unit tested at all using mainstream tools.

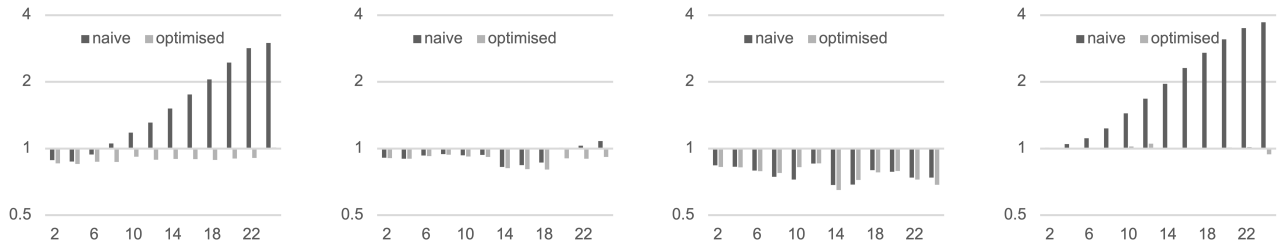


Fig. 9. Slowdown (y) of Prut4j-based versions of kernels CG, FT, IS, MG (from left to right) relative to reference programs, against numbers of threads (x)

distribution), we compared NPB's reference programs with the naive and decomposition-optimised Prut4j-based versions, by running each of these programs thirty times with various numbers of threads on a high-end multi-core machine (24 cores, ample memory). Fig. 9 shows slowdowns (if $y>1$) and speedups (if $y<1$) of the Prut4j-based versions relative to the reference programs (computed using mean run-times); standard deviations were mostly below 5% of the means.

Here, we highlight three findings. First, we observe that the concern is real for the naive Prut4j-based versions: the run-times become progressively worse for kernels CG (leftmost chart) and MG (rightmost chart) as the number of threads increases, up to almost $4\times$ slowdown in the worst case. The reason is that CG and MG, in contrast to kernels FT and IS (middle charts), are *communication-heavy*; thus, protocol overhead has a substantial impact. Second, however, we observe that the decomposition optimisation is really effective: (1) it eliminates the rising curves for CG and MG completely, (2) the worst slowdown is only $1.05\times$, and (3) we actually observe a speedup in many cases. This brings us to the third observation: especially for IS, the generated code is consistently faster than the hand-written code. This is a potential advantage of our work that we did not anticipate and warrants further research.

V. Conclusion

Lessons Learned: Regarding expressiveness, the unit tests that we wrote in Owl would have been prohibitively tedious to write by hand in Java (cf. Fig. 7). This seems to hold especially for unit tests that exercise complicated temporal dynamics. Interestingly, writing the unit tests also forced us to carefully think about the protocols; as a result, we found mistakes in our Discourje code, *even before executing the unit tests*.

Regarding efficiency, we conclude that especially with the decomposition optimisation enabled, the Prut4j-based versions really can compete with third-party reference programs.

Related work: Substantial research has been conducted on unit testing of concurrent programs (e.g., [12]–[16]). The central issue under study has been *flakiness* (e.g., [17]–[19]): unit tests for concurrent programs succeed or fail non-deterministically, due to the many interleavings in which threads can be scheduled. To address this, "existing automated testing tools for multi-threaded code mainly focus on re-executing existing test cases with different schedules" [20]. In contrast, Prut4j targets a different problem: unit testing of functionality that is *crucial* in concurrent programs but *difficult to modularise*.

Future work: The first evidence in support of the feasibility of our approach (in terms of expressiveness and efficiency; Sect. IV) encourages us to continue the development of Prut4j. Specifically, we aim for improved usability/ergonomics (user-friendlier/simpler syntax; less JUnit boilerplate), improved DSL features (advanced parametrisation; data-dependent protocols), optimisations, and real(istic) case studies.

References

[1] T. Tu, X. Liu, L. Song, and Y. Zhang, "Understanding real-world concurrency bugs in go," in *ASPLOS*. ACM, 2019, pp. 865–878.
[2] S. Jongmans, "Toward new unit-testing techniques for shared-memory concurrent programs," in *ICECCS*. IEEE, 2019, pp. 164–169.
[3] B. Goetz, *Java Concurrency in Practice*. Addison-Wesley, 2006.
[4] R. Hamers and S. Jongmans, "Discourje: Runtime verification of communication protocols in clojure," in *TACAS (1)*, ser. Lecture Notes in Computer Science, vol. 12078. Springer, 2020, pp. 266–284.
[5] J. Kretínský, T. Meggendorfer, and S. Sickert, "Owl: A library for ω-words, automata, and LTL," in *ATVA*, ser. Lecture Notes in Computer Science, vol. 11138. Springer, 2018, pp. 543–550.
[6] A. Pnueli, "The temporal logic of programs," in *FOCS*. IEEE Computer Society, 1977, pp. 46–57.
[7] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis, "Memory efficient algorithms for the verification of temporal properties," in *CAV*, ser. Lecture Notes in Computer Science, vol. 531, 1990, pp. 233–242.
[8] 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.
[9] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," *J. ACM*, vol. 63, no. 1, pp. 9:1–9:67, 2016.
[10] M. A. Frumkin, M. G. Schultz, H. Jin, and J. C. Yan, "Performance and scalability of the NAS parallel benchmarks in java," in *IPDPS*. IEEE Computer Society, 2003, p. 139.
[11] Z. Merali, "Computational science: ...Error," *Nature*, no. 467, pp. 775–777, 2010.
[12] 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.
[13] W. Pugh and N. Ayewah, "Unit testing concurrent software," in *ASE*. ACM, 2007, pp. 513–516.
[14] K. Coons, S. Burckhardt, and M. Musuvathi, "GAMBIT: effective unit testing for concurrency libraries," in *PPOPP*. ACM, 2010, pp. 15–24.
[15] 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.
[16] 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.
[17] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *SIGSOFT FSE*. ACM, 2014, pp. 643–653.
[18] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *ISSTA*. ACM, 2019, pp. 101–111.
[19] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! detecting flaky tests caused by concurrency with shaker," in *ICSME*, 2020, pp. 301–311.
[20] S. Steenbuck and G. Fraser, "Generating unit tests for concurrent classes," in *ICST*. IEEE Computer Society, 2013, pp. 144–153.