RFC-006: Unit testing and property-based testing of TLA+ specifications

authors	revision
Igor Konnov, Vitor Enes, Shon Feder, ...	1

Abstract. This document discusses a framework for testing TLA+ specifications. Our first goal is to give the writers of TLA+ specifications an interactive approach to quickly test their specifications in the design phase, similar to unit-testing in programming languages. Our second goal is to give the readers of TLA+ specifications a clear framework for dissecting TLA+ specifications, in order to understand them in smaller pieces. These ideas have not been implemented yet. We believe that the testing framework will enable the users of Apalache and TLC to write and read TLA+ specifications in a much more efficient manner than they do it today.

1. Long rationale

TLA+ is a specification language that was designed to be executable inside a human brain. Moreover, it was intended to run in the brains that underwent a specific software upgrade, called mathematical training. Many years have passed since then. We now have automatic tools that can run TLA+ in a computer (to some extent). Even more, these tools can prove or disprove certain properties of TLA+ specs.

Nowadays, we have two tools that aid us in writing a TLA+ spec: our brain and a model checker. Both these tools have the same problem. They are slow. Software engineers are facing a similar problem when they are trying to test their system against different inputs. Interestingly, software engineers have found a way around this problem. They first test the individual parts of the system and then they test the system as a whole. The former is done with unit tests, whereas the latter is done with integration tests. (Software engineers probably borrowed this approach from industrial engineers.) Unit tests are used almost interactively, to debug a small part of the system, while integration tests are run in a continuous integration environment, which is not interactive at all.

Actually, our brains also have a built-in ability of abstracting away from one part of a problem while thinking about the other part. That is why some of us can still win against automatic tools. Model checkers do not have this built-in ability. So it looks like when we are using TLC or Apalache, we are doing integration testing all the time. Unfortunately, when we are checking a specification as a whole, we rarely get a quick response, except for very small specs. This is hardly surprising, as we are interested in specifying complex systems, not the trivial ones.

Surprisingly, when we are writing large TLA+ specs, our interaction with the model checker looks more like an interaction with a Mainframe computer from the early days of computing than a modern interactive development cycle. We feed the model checker our specification and wait for hours in the hope that it gives us a useful response. If it does not, we have to make the specification parameters small enough for the model checker to do anything useful. If our parameters are already ridiculously small, we have to throw more computing power at the problem and wait for days. In contrast, verification tools for programs are meant to be much more interactive, e.g., see Dafny and Ivy.

Why cannot we do something like Unit testing in Apalache? We believe that we actually can do that. We can probably do it even better by implementing Property-based testing, that is, test parts of our specifications against a large set of inputs instead of testing it against a few carefully crafted inputs.

2. A motivating example

Let's consider a relatively simple distributed algorithm as an example. The repository of TLA+ examples contains the well-known leader election algorithm called LCR (specified in TLA+ by Stephan Merz). The algorithm is over 40 years old, but it is tricky enough to be still interesting. To understand the algorithm, check Distributed Algorithms by Nancy Lynch.

As the description suggests, when we fix `N` to `6` and `Id` to `<<27, 4, 42, 15, 63, 9>>`, TLC checks that the spec satisfies the invariant `Correctness` in just 11 seconds, after having explored 40K states. Of course, had we wanted to check the property for all possible combinations of six unique identifiers in the range of `1..6`, we would had to run TLC `6! = 720` times, which would take over 2 hours.

In Apalache, we can setup a TLA+ module instance, to check all instances of the algorithm that have from 2 to 6 processes:

```
-------------------- MODULE ChangRobertsTyped_Test -----------------------
(*
 * A test setup for ChangRobertsTyped.
 *)
EXTENDS Integers, Apalache

\* a copy of constants from ChangRobertsTyped
CONSTANTS
    \* @type: Int;
    N,
    \* @type: Int -> Int;
    Id

\* a copy of state variables from ChangRobertsTyped
VARIABLES
    \* @type: Int -> Set(Int);
    msgs,
    \* @type: Int -> Str;
    pc,
    \* @type: Int -> Bool;
    initiator,
    \* @type: Int -> Str;
    state

INSTANCE ChangRobertsTyped

\* We bound N in the test
MAX_N == 6

\* we override Node, as N is not known in advance
OVERRIDE_Node == { i \in 1..MAX_N: i <= N }

\* initialize constants
ConstInit ==
    /\ N \in 2..MAX_N
    /\ Id \in [ 1..MAX_N -> Int ]

\* The below constraints are copied from ASSUME.
\* They are not enforced automatically, see issue #69.
Assumptions ==
    /\ Node = DOMAIN Id
    /\ \A n \in Node: Id[n] >= 0
    /\ \A m,n \in Node : m # n => Id[m] # Id[n]  \* IDs are unique

InitAndAssumptions ==
    Init /\ Assumptions
```

By running Apalache as follows, we can check `Correctness` for all configurations of 2 to 6 processes and all combinations of `Id`:

```
apalache check --cinit=ConstInit \
  --init=InitAndAssumptions --inv=Correctness ChangRobertsTyped_Test.tla
```

Actually, we do not restrict `Id` to be a function from `1..N` to `1..N`, but rather allow `Id` to be a function from `1..N` to `Int`. So Apalache should be able to check an infinite number of configurations!

Unfortunately, Apalache starts to dramatically slow down after having explored 6 steps of the algorithm. Indeed, it does symbolic execution for a non-deterministic algorithm and infinitely many inputs. We could try to improve the SMT encoding, but that would only win us several steps more. A more realistic approach would be to find an inductive invariant and let Apalache check it.

It looks like we are trapped: Either we have to invest some time in verification, or we can check the algorithm for a few data points. In case of LCR, the choice of process identifiers is important, so it is not clear at all, whether a few data points are giving us a good confidence.

This situation can be frustrating, especially when you are designing a large protocol. For instance, both Apalache and TLC can run for hours on Raft without finishing. We should be able to quickly debug our specs like software engineers do!

3. An approach to writing tests

What we describe below has not been implemented yet. Apalache has all the necessary ingredients for implementing this approach. We are asking for your input to find an ergonomic approach to testing TLA+ specifications. Many of the following ideas apply to TLC as well. We are gradually introducing Apalache-specific features.

A complete specification can be found in ChangRobertsTyped_Test.tla.

Our idea is to quickly check operators in isolation, without analyzing the whole specification and without analyzing temporal behavior of the specification. There are three principally different kinds of operators in TLA+:

- Stateless operators that take input parameters and return the result. These operators are similar to functions in functional languages.
- Action operators that act on a specification state. These operators are similar to procedures in imperative languages.
- Temporal operators that act on executions, which are called behaviors in TLA+. These operators are somewhat similar to regular expressions, but they are more powerful, as they reason about infinite executions.

3.1. Testing stateless operators

Consider the following auxiliary operator in the specification:

```
succ(n) == IF n=N THEN 1 ELSE n+1  \* successor along the ring
```

While this operator is defined in the specification, it is clear that it is well isolated from the rest of the specification: We only have to know the value of the constant `N` and the value of the operator parameter `n`.

```
\* Note that succ(n) is not referring to state variables,
\* so we can test it in isolation.
\*
\* @require(ConstInit)
\* @testStateless
Test_succ ==
    \* This is like a property-based test.
    \* Note that it is exhaustive (for the range of N).
    \A n \in Node:
        succ(n) \in Node
```

This test is very simple. It requires `succ(n)` to be in the set `Node`, for all values `n \in Node`. The body of the operator `Test_succ` is pure TLA+. We annotate the operator with `@testStateless`, to indicate that it should be checked in a stateless context.

We should be able to run this test via:

```
apalache test ChangRobertsTyped_Test.tla Test_succ
```

We pass the test name `Test_succ`, as we expect the `test` command to run all tests by default, if no test name is specified. Also, we have to initialize the constants with `ConstInit`, which we specify with the annotation `@require(ConstInit)`.

3.2. Testing actions

Testing stateless operators is nice. However, TLA+ is built around the concept of a state machine. Hence, we believe that most of the testing activity will be centered around TLA+ actions. For instance, the LCR specification has two actions: `n0` and `n1`. Let's have a look at `n0`:

```
n0(self) == /\ pc[self] = "n0"
            /\ IF initiator[self]
                  THEN /\ msgs' = [msgs EXCEPT ![succ(self)] = @ \cup
{Id[self]}]
                  ELSE /\ TRUE
                       /\ msgs' = msgs
            /\ pc' = [pc EXCEPT ![self] = "n1"]
            /\ UNCHANGED << initiator, state >>
```

Assume we like to test it without looking at the rest of the system, namely, the predicates `Init` and `n1`. First of all, we have to describe the states that could be passed to the action `n0`. In this section, we will just use TypeOK (see Section 5 for a more fine-grained control over the inputs):

```
TypeOK ==
  /\ pc \in [Node -> {"n0", "n1", "n2", "Done"}]
  /\ msgs \in [Node -> SUBSET {Id[n] : n \in Node}]
  /\ initiator \in [Node -> BOOLEAN]
  /\ state \in [Node -> {"cand", "lost", "won"}]
```

Further, we specify what kind of outcome we expect:

```
\* Assertion that we expect to hold true after firing Action_n0.
Assert_n0 ==
    \E n, m \in Node:
        msgs'[n] = msgs[n] \union {m}
```

(Do you think this condition actually holds true after firing `n0`?)

Finally, we have to specify, how to run the action `n0`. In fact, if you look at `Next`, this requires us to write a bit of code, instead of just calling `n0`:

```
\* Execute the action under test.
\* Note that we decouple Assert_n0 from TestAction_n0.
\* The reason is that we always assume that TestAction_n0 always holds,
\* whereas we may want to see Assert_n0 violated.
\*
\* @require(ConstInit)
\* @require(TypeOK)
\* @ensure(Assert_n0)
\* @testAction
TestAction_n0 ==
    \E self \in Node:
        n0(self)
```

The operator `TestAction_n0` carries several annotations:

- The annotation `@require(TypeOK)` tells the framework that `TypeOK` should act as an initialization predicate for testing `TestAction_n0`.
- The annotation `@testAction` indicates that `TestAction_n0` should be tested as an action that is an operator over unprimed and primed variable.
- The annotation `@ensure(Assert_n0)` tells the framework that `Assert_n0` should hold after `TestAction_n0` has been fired.

We should be able to run this test via:

```
apalache test ChangRobertsTyped_Test.tla TestAction_n0
```

Importantly, we decompose the test in three parts:

- preparing the states by evaluating predicates `ConstInit` and `TypeOK` (similar to `Init`),
- executing the action by evaluating the action predicate `TestAction_n0` (like a single instance of `Next`),
- testing the next states against the previous states by evaluating the predicate `Assert_n0` (like an action invariant).

3.3. Testing executions

Engineers often like to test a particular set of executions to support their intuition, or to communicate an example to their peers. Sometimes, it is useful to isolate a set of executions to make continuous integration break, until the protocol is fixed. Needless to say, TLA+ tools have no support for this standard technique, though they have all capabilities to produce such tests.

Similar to testing an action in isolation, we propose an interface for testing a restricted set of executions as follows:

```
\* Execute a sequence of 5 actions, similar to TestAction_n0.
\* We test a final state with Assert_n0.
\*
\* @require(ConstInit)
\* @require(TypeOK)
\* @ensure(Assert_noWinner)
\* @testExecution(5)
TestExec_n0_n1 ==
    \* in this test, we only execute actions by processes 1 and 2
    \E self \in { 1, 2 }:
        n0(self) \/ n1(self)
```

In this case, we are using a different assertion in the `@ensure` annotation:

```
Assert_noWinner ==
    \A n \in Node:
        state'[n] /= "won"
```

The test `TestExec_n0_n1` is similar to `TestAction_n0` in many aspects. It starts by initializing the state with the predicate `Prepare_n0` and it expects a final state to satisfy the predicate `Assert_noWinner`. There is an important difference between the variables in `Assert_n0` and `Assert_noWinner`:

- Unprimed variables in `Assert_n0` refer to a state before firing an action, whereas primed variables in `Assert_n0` refer to a state after firing the action.

- Unprimed variables in `Assert_noWinner` refer to a state before firing an *execution*, whereas primed variables in `Assert_noWinner` refer to a final state of the execution.

(If you find the above behavior of `Assert_noWinner` confusing, please let us know.)

We should be able to run this test via:

```
apalache test ChangRobertsTyped_Test.tla TestExec_n0_n1
```

If the test is violated, a counterexample should be produced in the file `counterexample_TestExec_n0_n1.tla`.

3.4. Test executions with temporal properties

When we wrote the test `TestExec_n0_n1`, we did not think about the intermediate states of an execution. This test was a functional test: It is matching the output against the input. When reasoning about state machines, we often like to restrict the executions and check the properties of those executions.

Fortunately, we have all necessary ingredients in TLA+ to do exactly this. Test `TestExec_correctness_under_liveness`.

```
\* Execute a sequence of 5 actions, while using temporal properties.
\*
\* @require(ConstInit)
\* @require(TypeOK)
\* @require(Liveness)
\* @ensure(GlobalCorrectness)
\* @testExecution(5)
TestExec_correctness_under_liveness ==
    \E self \in Node:
        n0(self) \/ n1(self)
```

Predicates `Correctness` and `Liveness` are defined in the spec as follows:

```
(***************************************************************************)
(* Safety property: when node n wins the election, it is the initiator     *)
(* with the smallest ID, and all other nodes know that they lost.          *)
(***************************************************************************)
Correctness ==
  \A n \in Node : state[n] = "won" =>
     /\ initiator[n]
     /\ \A m \in Node \ {n} :
           /\ state[m] = "lost"
           /\ initiator[m] => Id[m] > Id[n]

Liveness == (\E n \in Node : state[n] = "cand") => <>(\E n \in Node : state[n] =
"won")
```

Since `Correctness` is a state predicate, we wrap it with a temporal operator to check it against all states of an execution:

```
GlobalCorrectness == []Correctness
```

3.5. Discussion

As you can see, we clearly decompose a test in three parts:

- preparing the states (like a small version of `Init`),
- executing the action (like a small version of `Next`),
- testing the next states against the previous states (like an action invariant).

In the rest of this section, we comment on the alternative approaches.

3.5.1. But I can do all of that in TLA+

True. TLA+ is an extremely expressive language.

Let's go back to the test `TestAction_n0` that was explained in Section 3.2:

```
\* Execute the action under test.
\* Note that we decouple Assert_n0 from TestAction_n0.
\* The reason is that we always assume that TestAction_n0 always holds,
\* whereas we may want to see Assert_n0 violated.
\*
\* @require(ConstInit)
\* @require(TypeOK)
\* @ensure(Assert_n0)
\* @testAction
TestAction_n0 ==
    \E self \in Node:
        n0(self)
```

Can we rewrite this test in pure TLA+? Yes, but it is an error-prone approach. Let's do it step-by-step.

First of all, there is no simple way to initialize constants in TLA+, as we did with `ConstInit` (this is an Apalache-specific feature). Of course, one can restrict constants with `ASSUME(...)`. However, assumptions about constants are global, so we cannot easily isolate constant initialization in one test. The canonical way of initializing constants is to define them in a TLC configuration file. If we forget about all these idiosyncrasies of TLC, we could just use implication (`=>`), as we normally do in logic. So our test `TestAction_n0_TLA` in pure TLA+ would look like follows:

```
TestAction_n0_TLA ==
  ConstInit => (* ... *)
```

Second, we want to restrict the states with `TypeOK`. That should be easy:

```
TestAction_n0_TLA ==
  ConstInit =>
    TypeOK (* ... *)
```

Third, we want to execute the action `n0`, as we did in `TestAction_n0`. The intuitive way is to write it like follows:

```
TestAction_n0_TLA ==
  ConstInit =>
    /\ TypeOK
    /\ \E self \in Node:
         n0(self)
    (* ... *)
```

Although the above code looks reasonable, we cheated. It combines two steps in one: It initializes states with `TypeOK` and it simultaneously executes the action `n0`. If we tried that in TLC (forgetting about `ConstInit`), that would not work. Though there is nothing wrong about this constraint from the perspective of logic, it just restricts the unprimed variables and primed variables. There is probably a way to split this code in two steps by applying the operator `\cdot`, which is implemented neither in TLC, nor in Apalache:

```
TestAction_n0_TLA ==
  ConstInit =>
    TypeOK
      \cdot
    (
      \E self \in Node:
         n0(self)
      (* ... *)
    )
```

In these circumstances, a more reasonable way would be to introduce a new file like `MCTestAction_n0.tla` and clearly specify `TypeOK` as the initial predicate and the action as the next predicate. But we do not want state-of-the-art dictate us our behavior.

Finally, we have to place the assertion `Assert_n0`. Let's try it this way:

```
TestAction_n0_TLA ==
  ConstInit =>
    TypeOK
      \cdot
    (
      /\ \E self \in Node:
         n0(self)
      /\ Assert_n0
    )
```

Unfortunately, this is not the right solution. Instead of executing `n0` and checking that the result satisfies `Assert_n0`, we have restricted the next states to always satisfy `Assert_n0`!

Again, we would like to write something like the implication `Action => Assertion`, but we are not allowed do that with the model checkers for TLA+. We can use the operator `Assert` that is supported by TLC:

```
TestAction_n0_TLA ==
  ConstInit =>
    TypeOK
      \cdot
    (
      /\ \E self \in Node:
         n0(self)
      /\ Assert(Assert_n0, "assertion violation")
    )
```

This time it should conceptually work. Once `n0` has been executed, TLC could start evaluating `Assert(...)` and find a violation of `Assert_n0`. There is another problem. The operator `Assert` is a purely imperative operator, which relies on the order in which the formula is evaluated. Hence, Apalache does not support this operator and, most likely, it never will. The imperative semantics of the operator `Assert` is simply incompatible with logical constraints. Period.

Phew. It was not easy to write `TestAction_n0_TLA`. In principle, we could fix this pattern and extract the test in a dedicated file `MC.tla` to run it with TLC or Apalache.

Let's compare it with `TestAction_n0`. Which one would you choose?

```
\* Execute the action under test.
\* Note that we decouple Assert_n0 from TestAction_n0.
\* The reason is that we always assume that TestAction_n0 always holds,
\* whereas we may want to see Assert_n0 violated.
\*
\* @require(ConstInit)
\* @require(TypeOK)
\* @ensure(Assert_n0)
\* @testAction
TestAction_n0 ==
    \E self \in Node:
        n0(self)
```

Another problem of `TestAction_n0_TLA` is that it has a very brittle structure. What happens if one writes `~ConstInit \/ TypeOK ...` instead of `ConstInit => TypeOK ...`? In our experience, when one sees a logical formula, they expect that an equivalent logical formula should be also allowed.

In the defense of TLA+, the issues that we have seen above are not the issues of TLA+ as a language, but these are the problems of the TLA+ tooling. There is a very simple and aesthetically pleasing way of writing `TestAction_n0` in the logic of TLA+:

```
TestAction_n0_pure_TLA ==
  (ConstInit /\ TypeOK) =>
    (\E self \in Node: n0(self)) => Assert_n0
```

The operator `TestAction_n0_pure_TLA` could be probably reasoned about in TLA+ Proof System. From the automation perspective, it would require a completely automatic constraint-based solver for TLA+, which we do not have. In practice, this would mean either rewriting TLC and Apalache from scratch, or hacking them to enforce the right semantics of the above formula.

We sum up this discussion by quoting Dr. Malcolm from Jurassic Park:

> Yeah, but your scientists were so preoccupied with whether or not they could, that they didn't stop to think if they should.

Thanks to Jure Kukovec for pointing to this quote!

3.5.2. Why annotations instead of special operators

The annotations `@require` and `@ensure` are not our invention. You can find them in Design-by-contract languages. In particular, they are used as pre- and post-conditions in code verification tools, e.g., JML, Dafny, QUIC testing with Ivy.

You could ask a reasonable question: Why cannot we introduce operators such as `Require` and `Ensure` instead of writing annotations? For instance, we could rewrite `TestAction_n0` as follows:

```
TestAction_n0_no_annotations ==
  /\ Require(ConstInit)
  /\ Require(TypeOK)
  /\ \E self \in Node:
        n0(self)
  /\ Ensure(Assert_n0)
```

The above test looks self-contained, no annotations. Moreover, we have probably given more power to the users: They could pass expressions to `Require` and `Ensure`, or they could combine `Require` and `Ensure` in other ways and do something great... Well, we have actually introduced more problems to the users than solutions. Since logical formulas can be composed in a lot of ways, we could start writing interesting things:

```
Can_I_do_that ==
  /\ ~Require(ConstInit)
  /\ Require(TypeOK) => Ensure(ConstInit)
  /\ \E self \in Node:
        n0(self) /\ Require(self \in { 1, 2 })
  /\ Ensure(Assert_n0) \/ Ensure(Assert_noWinner)
```

It is not clear to us how the test `Can_I_do_that` should be understood. But what is written is kind of legal, so it should work, right?

The annotations gives us a clear structure instead of obfuscating the requirements in logical formulas.

For the moment, we are using Apalache annotations in code comments. However, TLA+ could be extended with ensure/require one day, if they prove to be useful.

4. Using tests for producing quick examples

It is often nice to see examples of test inputs that pass the test. Apalache has all the ingredients to do that that. We should be able to run a command like that:

```
apalache example ChangRobertsTyped_Test.tla TestAction_n0
```

The above call would produce `example_TestAction_n0.tla`, a TLA+ description of two states that satisfy the test. This is similar to `counterexample.tla`, which is produced when an error is found.

In a similar way we should be able to produce an example of an execution:

```
apalache example ChangRobertsTyped_Test.tla TestExec_n0_n1
```

5. Bounding the inputs

The following ideas clearly stem from Property-based testing, e.g., we use generators similar to Scalacheck. In contrast to property-based testing, we want to run the test not only on some random inputs, but to run it exhaustively on all inputs within a predefined bounded scope.

5.1. Using Apalache generators

Let's go back to the example in Section 3.2.

In `TestAction_n0` we used `TypeOK` to describe the states that can be used as the input to the test. While this conceptually works, it often happens that `TypeOK` describes a large set of states. Sometimes, this set is even infinite, e.g., when `TypeOK` refers to the infinite set of sequences `Seq(S)`. In Apalache, we can use the operator `Gen` that produces bounded data structures, similar to Property-based testing. Here is how we could describe the set of input states, by bounding the size of the data structures:

```
\* Preparing the inputs for the second test. Note that this is a step of its
own.
\* This is similar to an initialization predicate.
Prepare_n0 ==
    \* the following constraint should be added automatically in the future
    /\ Assumptions
    \* let the solver pick some data structures within the bounds
    \* up to 15 messages
    /\ msgs = Gen(3 * MAX_N)
    /\ pc = Gen(MAX_N)
    /\ initiator = Gen(MAX_N)
    /\ state = Gen(MAX_N)
    \* restrict the contents with TypeOK,
    \* so we don't generate useless data structures
    /\ TypeOK
```

In `Prepare_n0`, we let the solver to produce bounded data structures with `Gen`, by providing bounds on the size of every set, function, sequence, etc. Since we don't want to have completely arbitrary values for the data structures, we further restrict them with `TypeOK`, which we conveniently have in the specification.

The more scoped version of `TestAction_n0` looks like following:

```
\* Another version of the test where we further restrict the inputs.
\*
\* @require(ConstInit)
\* @require(Prepare_n0)
\* @ensure(Assert_n0)
\* @testAction
TestAction2_n0 ==
    \E self \in Node:
        n0(self)
```

5.2. Using TLC Random

Leslie Lamport has recently introduced a solution that allows one to run TLC in the spirit of Property-based testing. This is done by initializing states with the operators that are defined in the module `Randomization`. For details, see Leslie's paper on Inductive invariants with TLC.

6. Test options

To integrate unit tests in the standard TLA+ development cycle, the tools should remember how every individual test was run. To avoid additional scripting on top of the command-line interface, we can simply pass the tool options with the annotation `@testOption`. The following example demonstrates how it could be done:

```
\* A copy of TestExec_n0_n1 that passes additional flags to the model checker.
\*
\* @require(ConstInit)
\* @require(TypeOK)
\* @ensure(Assert_noWinner)
\* @testExecution(5)
\* @testOption("tool", "apalache")
\* @testOption("search.smt.timeout", 10)
\* @testOption("checker.algo", "offline")
\* @testOption("checker.nworkers", 2)
TestExec_n0_n1 ==
    TestExec_n0_n1
```

The test options in the above example have the following meaning:

- The annotation `testOption("tool", "apalache")` runs the test only if it is executed in Apalache. For example, if we run this test in TLC, it should be ignored.

- The annotation `testOption("search.smt.timeout", 10)` sets the tool-specific option `search.smt.timeout` to 10, meaning that the SMT solver should time out if it cannot solve a problem in 10 seconds.

- The annotation `testOption("checker.algo", "offline")` sets the tool-specific option `checker.algo` to `offline`, meaning that the model checker should use the offline solver instead of the incremental one.

- The annotation `testOption("checker.nworkers", 2)` sets the tool-specific option `checker.nworkers` to `2`, meaning that the model checker should use two cores.

By having all test options specified directly in tests, we reach two goals:

- We let the users to save their experimental setup, to enable reproducibility of the experiments and later re-design of specifications.
- We let the engineers integrate TLA+ tests in continuous integration, to make sure that updates in a specification do not break the tests. This would allow us to integrate TLA+ model checkers in a CI/CD loop, e.g., at GitHub.