The TLA$^+$ Trifecta walks into a bar, the bartender asks Why3?

Ion Chirica[1], Mário Pereira[1], and Jorge Sousa Pinto[2]

[1] NOVA LINCS & Universidade Nova de Lisboa, Portugal
[2] HASLab/INESC TEC & Universidade do Minho, Portugal

Abstract. We present a methodology for verifying distributed algorithms that retains the TLA$^+$ style of state-machine specification while conducting the proof effort in Why3. Our methodology is based on a library for encoding state machine specifications and their inductive invariants, Why3-do. To showcase our work we manually translate, into our framework, an already specified and verified example for an Asynchronous Termination Detection algorithm from the literature. This work sets forth a frontend that maps TLA$^+$ specifications into Why3-do models, offering practitioners already accustomed to TLA$^+$ with a deductive-based, mostly-automated verification workflow.

1 Introduction

Among the available languages to model and reason about distributed and concurrent systems, TLA$^+$ [15] stands out for its widespread adoption by the distributed systems community. Based on Temporal Logic of Actions, it allows high-level algorithm descriptions through Zermelo-Fraenkel set theory and implicitly models concurrency through interleaving semantics. A recent trifecta study [14] demonstrates how the ecosystem of TLA$^+$ tools can be combined to cover different parts of the verification spectrum: TLC [20], a model checker, APALACHE [12], a symbolic SMT-based bounded model checker, and TLAPS [7], an interactive proof system for deductive verification.

Within this ecosystem, TLAPS provides the highest correctness guarantees with proofs of properties of arbitrary instances of a specification. Proofs are conducted by the user and checked by the Proof System. These proofs require significant manual guidance, and the integration between TLA$^+$'s mathematical language and backend solvers can be brittle. This motivates exploring alternative platforms that offer stronger automation while preserving the state-machine style of specification, something that TLA$^+$ practitioners are already familiar with.

In this paper, we present a methodology for verifying distributed algorithms that retains the TLA$^+$ style of state-machine specification while conducting the proof effort in Why3 [9], a mature deductive verification platform with strong automation and multi-prover support. Additionally, our methodology is based on a WhyML library for encoding state machine specifications and their inductive invariants, Why3-do [3]. This work sets forth a frontend that maps

TLA$^+$ specifications into Why3-do models. While this translation is, for now, done manually, we expect to be able to automatize this effort as future work.

We adapt the example presented in prior work [14] of an Asynchronous Termination Detection high-level specification and subsequent refinement with a concrete algorithm due to Safra from Dijkstra's notes. All the examples are available in an online artifact [5]. This document adopts a literate programming style of listing presentation.

Outline of the paper: In Section 2 we present Why3 and the Why3-do library, namely, the modules on inductiveness and refinement of state machines. In Section 3 we present the manual translation of an Asynchronous Termination Detection TLA$^+$ specification to WhyML using the Why3-do library primitives. In Section 4 we present a refinement module for the previous specification, an algorithm due to Safra from Dijkstra's notes. In Section 5 we discuss the contrast of manual intervention between Why3 and TLAPS in the context of the refined module. We conclude in Section 6 with a discussion for approaches to automate the translation mechanism and extensions to the Why3-do library.

2 A primer on Why3-do

Why3 is a deductive verification platform that relies on multiple external provers, including SMT solvers such as Alt-Ergo [6] and CVC5 [2], and interactive proof assistants such as Rocq [18], to discharge verification conditions. Programs and specifications are written in WhyML, a functional language with mutable state and a rich contract language for pre- and postconditions, assertions, and invariants. Why3 automatically generates verification conditions from annotated code and dispatches them to the available provers.

Why3-do is a WhyML library used to describe state machine specifications and distributed systems [3]. A state machine specification defines the system's behaviour by mapping a transition of states through actions, or events. A specification of the system behaviour can be described by two types of properties: *safety* and *liveness*. In this context, Why3-do specifies the system's behaviour using inductive invariants, capturing *safety* properties. That means that the behaviour is an invariant that must hold at the beginning and preserved by all state transitions. As of the moment, Why3-do is unable to specify liveness properties. We present the formal model of Why3-do from a previous work [3], but wish to focus mostly on its WhyML encoding.

2.1 Inductiveness in Why3-do

Formally, a state machine specification, S, is defined by the triple $\langle \Sigma, \Sigma_0, \rightsquigarrow \rangle$, where Σ represents a set of states, $\rightsquigarrow$ is the transition relation from one state to another, and Σ_0 a set of initial states, contained within the set of all possible states. Moreover, given a specification S and a predicate Φ on states, we say that Φ is an inductive invariant if ① $\Phi(w_0)$ holds for every $w_0 \in \Sigma_0$, and ② for all

states $w, w' \in \Sigma$, if $\Phi(w)$ holds and $w \rightsquigarrow w'$, then $\Phi(w')$ too holds. This can be encoded in WhyML in a module, called `Inductiveness`, as such:

```
module Inductiveness                                                    WhyML
  type world
  predicate inv (w: world)

  val ghost predicate initWorld (w0: world)
    ① ensures { result → inv w0 }

  val ghost predicate step (w1 w2: world)
    ② ensures { result → inv w1 → inv w2 }
```

Here, the abstract type `world` represents the state space, Σ. In WhyML postconditions are `ensures` clauses and the keyword `result` denotes the return value of the function being specified. The system's behaviour is captured by `initWorld`, with its argument contained in Σ_0, and `step`, $\rightsquigarrow$ by including a postcondition enforcing the inductive step: if the invariant holds in the pre-state (`w1`), a valid transition guarantees it holds in the post-state (`w2`).

Keen-eyed readers might have noticed the use of *ghost* code in this module. In Why3, declarations marked with the `ghost` keyword exist purely for specification and verification purpose, being erased during code extraction and have no computational value. Both `initWorld` and `step` are declared as `val ghost`, meaning they are abstract and uninterpreted functions whose only purpose is to carry proof obligations through their postconditions. The `val` keyword signals that these predicates have no provided definition at this level of abstraction and are left to be supplied by the user when cloning the module with concrete instantiations.

The module also defines state reachability through the inductive predicate `step_TR`, the transitive and reflexive closure of `step`:

```
  inductive step_TR world world =                                       WhyML
  | base : ∀ w: world. step_TR w w
  | step : ∀ w w' w'': world. step_TR w w' → step w' w'' → step_TR w w''
```

A state is considered reachable if there exists a valid initial state from which it can be reached via `step_TR`, written in WhyML as:

```
  predicate reachable (w: world) =                                      WhyML
    ∃ w0. initWorld w0 ∧ step_TR w0 w
```

This module resides in a file named `inductiveness.mlw`. Meaning that an external module can refer to it through `inductiveness.Inductiveness`. This wraps up the module, and consequently, the section.

```
end                                                                     WhyML
```

2.2 Refinement in Why3-do

Proofs by refinement rely on establishing a forward simulation between the two state machines. This simulation is captured with a refinement mapping, say *refn*.

This is a mapping from a concrete to an abstract state, or model. To prove that Safra's algorithm correctly implements the abstract Asynchronous Termination Detection, we must discharge two primary verification conditions. First, initial states in the concrete model must map to valid initial states in the abstract model. Second, every valid transition in the concrete model must either refine to the same state $\mathit{refn}(w) = \mathit{refn}(w')$, or their refinement mapping is captured in the abstract model $\mathit{step}\ (\mathit{refn}\ w)\ (\mathit{refn}\ w')$. Proofs by refinement in Why3-do are best described in [3]. Refinement mappings are generally described in [1].

As to illustrate how this is represented in WhyML, let us briefly showcase parts of the `Refinement` module, from the Why3-do library. We first begin by defining two types for the abstract and concrete states, respectively:

```
module Refinement                                                    WhyML
  type worldA
  type worldC
```

We follow by defining an inductive relation between the two, built on the existence of a refinement mapping between the models. This is a function that takes as argument a concrete and returns an abstract model, as such:

```
  val ghost function refn (worldC) : worldA                          WhyML
```

If we ignore the ideas of the concrete world, we are left with the `Inductiveness` module. The first condition regards the initial states and is captured with the postcondition ③ that the refinement mapping must reflect valid abstract states:

```
  val ghost predicate initWorldA (w: worldA)                         WhyML
    ensures { result → invA w }

  val ghost predicate initWorldC (w: worldC)
    ensures { result → invC w }
③   ensures { result → initWorldA (refn w) }
```

On the other hand, the second condition regards the transition function `stepC` which must, as before, show both states respect the invariant. However, as an addition in ④, we also say that this transition must either correspond to the same refined abstract state or be captured in the abstract model by some action:

```
  val ghost predicate stepA (w1: worldA) (w2: worldA)                WhyML
    ensures { result → invA w1 → invA w2 }

  val ghost predicate stepC (w1: worldC) (w2: worldC)
    ensures { result → invC w1 → invC w2 }
④   ensures { result → invC w1 → refn w1 = refn w2  ∨
                            stepA (refn w1) (refn w2) }
```

When cloning the `Refinement` module, we are required to instantiate both the abstract and concrete models, *i.e.*, the abstract and concrete state type definitions, invariant predicates, and initial and transition steps. This generates four verification conditions showing that the instantiations for the predicates respect the above contracts. This wraps up the module and this section:

```
end                                                                  WhyML
```

2.3 Functional specification of actions

In TLA+, system transitions are described by *actions*, predicates over unprimed, say x, and primed variables, x', capture the conditions under which a transition may fire and the resulting state after it does, respectively. Why3-do encodes this pattern functionally, splitting each action into two components: a so called *enabled* predicate and a *transition function*. These components are user defined, per action, and later used to clone the `Inductiveness` or `Refinement` modules. The enabled predicate captures the precondition of the action, *i.e.*, the conditions under which the transition may fire:

```
predicate action_enabled (w: world) =                    WhyML
  (* some enabling condition *)
```

On the other hand, the transition function takes the current state and returns the successor state. It is decorated with a `requires` clause that ties it to the enabled predicate, and an `ensures` clause that encodes invariant preservation. If we are dealing with an action from a model that is due to refine another we have to also add the second postcondition. This is expressed in WhyML with the following:

```
let ghost function action (w: world) : world            WhyML
  requires { action_enabled w }
  ensures  { inv w → inv result }          (* Inductiveness *)
  ensures  { inv w → refn w = refn result ∨
               step (refn w) (refn result) } (* Refinement *)
=
  { w with ... } (* state update *)
```

Unchanged variables, identified in TLA+ with the keyword UNCHANGED, are easily captured in WhyML with the record update syntax. In WhyML, one can be explicit about record fields to update using the `with` keyword. In most cases, the transition function makes use of this construction to entirely capture unchanged variables without being explicit about them.

The first postcondition directly captures the inductive step for this specific transition, asserting that if the invariant holds in the pre-state, it is preserved in the resulting post-state. A major advantage of this approach is proof modularity: Why3 generates independent verification conditions for each function, allowing automated solvers to discharge the obligations for each action in isolation. Therefore, as long as each transition is verified independently, global safety via reachability is guaranteed to hold.

We note that this stands in contrast to proof methodologies in TLA+, where proving inductive invariance directly against the next-state relation (*Next*) can quickly become too complex for back-end solvers to handle in a single step [7]. In fact, when using TLAPS, invariance proofs are mostly conducted by case analysis on each action. In Why3-do, this case split becomes purely structural and is enforced by construction by verifying each action's invariant preservation. Finally, the complete transition relation of the state machine is assembled via the `stepind` inductive type, which enumerates the valid transitions as constructors:

```
inductive stepind world world =                                    WhyML
| action : ∀ w: world. action_enabled w → stepind w (action w)

let ghost predicate step (w1 w2: world) = stepind w1 w2
```

Each constructor witnesses one possible transition, taking its enabling condition as a hypothesis to relate the current and successor states. The `step` predicate is then simply defined using `stepind`. Alternatively, this relation could be defined as a chain of disjunctions instead of an inductive predicate.

The cloning mechanism in Why3 [10] works by instantiating an abstract module with concrete type, predicate and function definitions. When a module is cloned into a new scope with `clone`, using the `export` keyword inlines the instantiated declarations directly into the current namespace. If a symbol is omitted from the list of cloning arguments, it retains its original definition from the cloned module. Furthermore, Why3 provides a syntactic shortcut: if a concrete symbol shares the same name as the abstract symbol it replaces, its explicit binding can be omitted. In this case, if in the concrete module we had defined a `safe` predicate as our invariant, we could clone the module as:

```
clone export inductiveness.Inductiveness with                     WhyML
  type world, predicate inv = safe, val initWorld = init, val step
```

Given these concrete substitutions, Why3 would generate two verification conditions corresponding to the postconditions of `initWorld` and `step`. These obligations ensure that the concrete functions satisfy inductiveness.

3 Asynchronous Termination Detection

In the same spirit as the trifecta study, we present a Why3-do specification of Asynchronous Termination Detection (ATD). The TLA$^+$ specification is available here [13]. In a later section, we refine this abstract specification.

This algorithm follows a ring topology, where each node has two and only two neighbors, one to the right and another to the left. Nodes can be in an *active* or *inactive* state. Active nodes can send messages, and inactive nodes can receive messages turning them active. By means of this notion of message passing to the neighbors, our goal is to detect termination, that is, when all nodes are inactive and there are no pending messages in the network, which by consequence would make a node active.

To do this, we first begin a WhyML module, and seeing as we will refer to it multiple times later on, we prefer to keep it short. Let us call it:

```
module ATD                                                         WhyML
```

The specification assumes that the number of nodes in the system is a positive non-zero integer. In TLA$^+$ this is described with the following:

$$\text{CONSTANT} \quad N \qquad \text{TLA}^+$$
$$\text{ASSUME} \; NAssumption \triangleq N \in Nat \setminus \{0\}$$

Conversely, in WhyML, we state that `n` is a integer constant and then axiomatize that it is a non-zero integer, as such:

WhyML
```
val constant n : int
axiom n_assumption : n > 0
```

Moreover, let us say that each system node is identified by a positive integer up to the number of nodes. In TLA$^+$ this is expressed as:

TLA$^+$
$$Node \triangleq 0 \,..\, N-1$$

In WhyML we write the predicate `node` indicating where an identifier is a node:

WhyML
```
predicate node (i: int) = 0 ⩽ i < n
```

We could have opted to represent nodes by a record that would have a type invariant restricting its domain with the same predicate, akin to refinement types. This would ease specification by reducing the book-keeping of referring nodes, and shorten the mapping gap between the TLA$^+$ and Why3-do specification, at the cost of being a nightmare to conduct proofs in the refinement module.

When defining the state of the system, we are faced with a design choice on how to represent each node's internal state. One option is a *global view*, where the system state is a single record whose fields are maps indexed by node identifiers. Another option is we assume a *local view* where we could have a global map that maps to states. To be faithful to the TLA$^+$ specification we choose the former.

In TLA$^+$, the specification begins by defining the state variables, namely *active*, *pending* and *terminationDetected*:

TLA$^+$

VARIABLES

active,	activation status of nodes
pending,	number of messages pending at a node
terminationDetected	has termination been detected?

In WhyML, the system state is represented as a record with three fields: `active` that maps integers to Booleans that represent their activation status, `pending` that maps integers to the number of pending messages, and `terminationDetected`, a Boolean that indicates whether the system has detected termination or not.

WhyML
```
type world = {
   active : map int bool;      (* activation status of nodes     *)
  pending : map int int;       (* number of pending messages     *)
  terminationDetected : bool; (* has termination been detected? *)
}
```

We define termination locally on states. This predicate is defined by all nodes being inactive with no pending messages. In TLA$^+$ this is encoded as:

TLA$^+$
$$terminated \triangleq \forall\, n \in Node : \neg active[n] \wedge pending[n] = 0$$

In WhyML, instead of universally quantifying over all nodes, we quantify over all integers and add the premise of being nodes. What TLA$^+$ regards as variables of the specification, we regard as variables of a state, thus this predicate relates to a state that comes as argument, `w`, as follows:

```
predicate terminated w =                                        WhyML
  ∀ i. node i → ¬ w.active[i] ∧ w.pending[i] = 0
```

Because TLA$^+$ is built on untyped set theory, there is no static type system to constrain the domain of variables. Instead, a common specification pattern is to define a *TypeOK* predicate that explicitly restricts each variable to its intended domain. In this specification, *TypeOK* takes the following shape:

$$\begin{array}{l} \mathit{TypeOK} \triangleq \\ \quad \land\ \mathit{active} \in [\mathit{Node} \Rightarrow \text{BOOLEAN}\ ] \\ \quad \land\ \mathit{pending} \in [\mathit{Node} \Rightarrow \mathit{Nat}] \\ \quad \land\ \mathit{terminationDetected} \in \text{BOOLEAN} \end{array} \qquad \textit{TLA}^+$$

In Why3, a strongly typed language, the type system handles for free much of what *TypeOK* expresses, provided the mapping between TLA$^+$ and WhyML types is done correctly. However, some information still needs to be stated explicitly. In particular, the TLA$^+$ constraint $\mathit{pending} \in [\mathit{Node} \Rightarrow \mathit{Nat}]$ tells us that pending messages are always non-negative. In WhyML we must capture this constraint in our own `type_ok` predicate:

```
predicate type_ok w = ∀ i. node i → w.pending[i] ⩾ 0            WhyML
```

This wraps out the state representation. Though we still need to represent the system properties, the initial state and all the possible actions that evolve the system. As a safety property, and subsequently a system invariant, we state that if the system detects termination, by setting the `terminationDetected` flag, then the system must have really terminated, by the definition of the `terminated` predicate. This can be expressed in WhyML with the following:

$$\mathit{Safe} \triangleq \mathit{terminationDetected} \implies \mathit{terminated} \qquad \textit{TLA}^+$$

The translation to WhyML is straightforward, as follows:

```
predicate safe w = w.terminationDetected → terminated w         WhyML
```

Finally, the conjunction of both predicates make up our invariant:

```
predicate inv w = type_ok w ∧ safe w                            WhyML
```

The initial states, in TLA$^+$ identified by *Init*, describe the initial conditions of state variables. In this context, a node is either active or inactive, with no pending messages. Moreover, the flag that indicates whether termination has been detected, `terminationDetected`, can already be set if all nodes are inactive at the beginning. This is expressed as its value being in the set of either FALSE or *terminated*, indicating a non deterministic choice.

$$\begin{array}{l} \mathit{Init} \triangleq \\ \quad \land\ \mathit{active} \in [\mathit{Node} \to \text{BOOLEAN}\ ] \\ \quad \land\ \mathit{pending} = [n \in \mathit{Node} \mapsto 0] \\ \quad \land\ \mathit{terminationDetected} \in \{\text{FALSE}, \mathit{terminated}\} \end{array} \qquad \textit{TLA}^+$$

In WhyML, we can encode non-determinism with an inductive predicate. Let us call it `initind`. This predicate captures both possible initial conditions and necessarily grows exponentially, in the number of cases, if more variables were to be defined non deterministically. This predicate is defined as follows:

WhyML
```
inductive initind world =
 | init  : ∀ w. (∀ i. node i → w.pending[i] = 0) ∧
              w.terminationDetected = false → initind w
 | init' : ∀ w. (∀ i. node i → w.pending[i] = 0) ∧
              w.terminationDetected = terminated w → initind w
```

While the `initind` predicate captures the initial conditions of the system, it does not have any regard for the system invariant. For that, we use a ghost function, `initWorld`, equipped with a postcondition asserting that any state satisfying the initial conditions already respects the invariant.

WhyML
```
let ghost predicate initWorld w
  ensures { result → inv w }
=
  initind w
```

The TLA+ specification has 4 non stuttering actions, defined by the next-state action, *Next*. These actions capture a message reception and node termination, a message exchange between two nodes and the system detecting termination.

TLA+
$$
\begin{aligned}
&\mathit{Next} \triangleq \\
&\quad \lor \exists\, i \in \mathit{Node} : \mathit{RcvMsg}(i) \lor \mathit{Terminate}(i) \\
&\quad \lor \exists\, i, j \in \mathit{Node} : \mathit{SendMsg}(i, j) \\
&\quad \lor \mathit{DetectTermination}
\end{aligned}
$$

In TLA+, the action that receives a message, *RcvMsg*, is parameterized by a node i where, in the pre-state, it must have some pending message to signal it received a message. In the post-state, the node is now active, highlighted by the $[\mathit{active}$ EXCEPT $![i] = \text{TRUE}]$ and the number of pending messages is decremented by one, as such:

TLA+
$$
\begin{aligned}
&\mathit{RcvMsg}(i) \triangleq \quad \text{node } i \text{ receives a pending message} \\
&\quad \land \mathit{pending}[i] > 0 \\
&\quad \land \mathit{active}' = [\mathit{active} \text{ EXCEPT } ![i] = \text{TRUE}] \\
&\quad \land \mathit{pending}' = [\mathit{pending} \text{ EXCEPT } ![i] = @ - 1] \\
&\quad \land \text{UNCHANGED } \mathit{terminationDetected}
\end{aligned}
$$

The $![i]$ captures the value of the map at the i-th position. The @ symbol that appears in the companion conjunct denotes the current value of the function at the specified index. Seeing as the node is now active, termination cannot be concluded, so *terminationDetected* remains unchanged.

Conversely, from a Why3-do functional specification perspective, this equates into gathering unprimed variables into the enabled predicate, and making sure that `i` is a node. This is achieved with the following:

WhyML
```
predicate receive_message_enabled (w: world) (i: int) =
  node i ∧ w.pending[i] > 0
```

The transition function, `receive_message`, takes the enabling predicate as a precondition and updates all the primed variables with their new value. The `terminationDetected` field remains unchanged by the record update:

WhyML
```
let ghost function receive_message (w: world) (i: int) : world
  requires { receive_message_enabled w i }
  ensures  { inv w → inv result }
=
  { w with active = w.active[i ← true];
           pending = w.pending[i ← w.pending[i] - 1] }
```

In TLA$^+$, the *SendMsg* action is parameterized by two nodes, i and j. The enabling condition simply requires the sender i to be active. As a result of the transition, the network registers a new message in transit towards j by incrementing its pending count, while leaving the nodes' activation status and the termination flag unchanged:

TLA$^+$
$$
\begin{aligned}
&\mathit{SendMsg}(i, j) \triangleq \\
&\quad \land\ \mathit{active}[i] \\
&\quad \land\ \mathit{pending}' = [\mathit{pending}\ \text{EXCEPT}\ ![j] = @ + 1] \\
&\quad \land\ \text{UNCHANGED}\ \langle \mathit{active}, \mathit{terminationDetected} \rangle
\end{aligned}
$$

Translated to our functional Why3-do methodology, we extract the precondition into `send_message_enabled`, ensuring both parameters are valid nodes. The post-state increments the number `j`'s pending messages, as such:

WhyML
```
predicate send_message_enabled (w: world) (i j: int) =
  node i ∧ node j ∧ w.active[i]

let ghost function send_message (w: world) (i j: int) : world
  requires { send_message_enabled w i j }
  ensures  { inv w → inv result }
=
  { w with pending = w.pending[j ← w.pending[j] + 1] }
```

The *Terminate* action is parameterized by some node i. In this case, the pre-state requires the node to be active and in the post-state it becomes inactive. The *pending* variable remains unchanged. Now that we have de-activated a node, it may happen that the system is able to detect termination, the *terminationDetected* variable is updated non-deterministically: it can either retain its pre-state value or take on the Boolean value of the post-state predicate *terminated*′.

TLA$^+$
$$
\begin{aligned}
&\mathit{Terminate}(i) \triangleq \\
&\quad \land\ \mathit{active}[i] \\
&\quad \land\ \mathit{active}' = [\mathit{active}\ \text{EXCEPT}\ ![i] = \text{FALSE}] \\
&\quad \land\ \mathit{pending}' = \mathit{pending} \\
&\quad \land\ \mathit{terminationDetected}' \in \{\mathit{terminationDetected}, \mathit{terminated}'\}
\end{aligned}
$$

In WhyML, we encode the enabling predicate for `terminate` with the following:

WhyML
```
predicate terminate_enabled (w: world) (i: int) =
  node i ∧ w.active[i]
```

As was the case with the initial state description, `initind`, we can now also represent non-determinism with two different actions, `terminate` and `terminate'`. The first action portrays the case where termination is not detectable, with the post-state only deactivating node i, as such:

```
let ghost function terminate (w: world) (i: int) : world
  requires { terminate_enabled w i }
  ensures  { inv w → inv result }
=
  { w with active = w.active[i ← false] }
```
WhyML

In the second action, we still need to deactivate i, but it now can happen that the system is able to detect termination. What is interesting to note when translating this to a sequential language like WhyML is the dependency between the state variables. The $terminated'$ predicate is simply a macro evaluated over the newly defined $active'$ state. However, in WhyML, we must explicitly order these updates, updating the active state first before checking for termination:

```
let ghost function terminate' (w: world) (i: int) : world
  requires { terminate_enabled w i }
  ensures  { inv w → inv result }
=
  let w' = { w with active = w.active[i ← false] } in
  { w' with terminationDetected = terminated w'; }
```
WhyML

Finally, the *DetectTermination* action is responsible for setting the variable *terminationDetected* flag to true, provided that the system has entirely ceased activity. While the previous action actively detected termination, this one does it passively. In a sequential and synchronous environment, this action would not be necessary, but as we are dealing asynchronous concurrency, a node may terminate and fail to flip the `terminationDetected` flag. This action plays that part independently. Its TLA$^+$ definition is straightforward and requires the *terminated* predicate to hold in the pre-state:

$$
\begin{aligned}
&\mathit{DetectTermination} \triangleq \\
&\quad \land\ \mathit{terminated} \\
&\quad \land\ \mathit{terminationDetected}' = \text{TRUE} \\
&\quad \land\ \text{UNCHANGED } \langle \mathit{active}, \mathit{pending} \rangle
\end{aligned}
$$
TLA$^+$

At this point it should not come as a surprise to get the following for the enabling predicate:

```
predicate detect_termination_enabled (w: world) =
  terminated w
```
WhyML

The action definition itself is too straightforward, only setting the termination flag to true, as such:

```
let ghost function detect_termination (w: world) : world
  requires { detect_termination_enabled w }
  ensures  { inv w → inv result }
=
  { w with terminationDetected = true }
```
WhyML

With all individual actions defined as contract-equipped functions, we bundle them into the single transition relation using the `stepind` inductive type, as we have seen before. Each constructor maps to the five possible actions:

```
inductive stepind world world =                                    WhyML
| receive_message : ∀ w: world, i: int.
    receive_message_enabled w i → stepind w (receive_message w i)
| send_message : ∀ w: world, i j: int.
    send_message_enabled w i j → stepind w (send_message w i j)
| terminate : ∀ w: world, i: int.
    terminate_enabled w i → stepind w (terminate w i)
| terminate' : ∀ w: world, i: int.
    terminate_enabled w i → stepind w (terminate' w i)
| detect_termination : ∀ w: world.
    detect_termination_enabled w → stepind w (detect_termination w)
```

```
let ghost predicate step (w1 w2: world) = stepind w1 w2            WhyML
```

To conclude the formalization of this abstract module and generate the corresponding proof obligations, we clone the abstract `Inductiveness` module, substituting its abstract parameters with the newly defined concrete types and functions.

```
clone export inductiveness.Inductiveness with                     WhyML
  type world, predicate inv, val initWorld, val step
```

In total, Why3 generates 8 verification conditions, one for each action, one for the initial state, and two that come from the `Inductiveness` module, namely the contract for `initWorld` and `step`. The SMT solvers dispatch all of them in an instant. We can now close the module and this section.

```
end                                                                WhyML
```

4 Safra's Algorithm for Termination Detection

In this section, we present the concrete algorithm proposed by Shmuel Safra (as part of Dijkstra's notes EWD 998) to solve the distributed termination detection problem over a ring topology [8,14]. The algorithm overlays a token-passing mechanism on top of the asynchronous network. The token circulates around the ring, aggregating the number of messages sent and received by each node, while also tracking whether any node might have been reactivated.

In order to achieve this, each node maintains a *color* (either white or black) and a local *counter* tracking the difference between the number of messages it has sent and received. A node turns black if it sends a message, indicating that it could have reactivated another node. The circulating token also carries a color and a counter value, q. When an inactive node passes the token, it adds its local counter to the token's counter q. If a black node passes the token, the token itself becomes black, effectively invalidating the current detection probe. The token becomes white again only when a new probe is initiated by node 0.

Termination is detected when the token returns to node 0 and satisfies a specific set of conditions: the token is white, node 0 is white and inactive, and the sum of the token's counter q and node 0's local counter is zero.

In TLA$^+$, the state of the algorithm introduces new variables for the circulating token, the colors of the nodes, and their local counters.

TLA$^+$

$$Color \triangleq \{\texttt{"white"}, \texttt{"black"}\}$$
$$Token \triangleq [pos : Node,\ q : Int,\ color : Color]$$
VARIABLES *active*, *color*, *counter*, *pending*, *token*

Following the methodology introduced in previous sections, we map this state representation into WhyML, say in a `EWD998` module. We start by defining the type for the colors and the record type for the token, as such:

WhyML

```
module EWD998

  type color = White | Black
  type token = { pos: int; q: int; t_color: color }
```

Next, the global state of the system is captured by extending the abstract state of the asynchronous network with the newly introduced variables. We maintain the *global view* approach where maps are indexed by node identifiers:

WhyML

```
  type world = {
    active : map int bool;  (* activation status of nodes *)
    color  : map int color; (* color of nodes *)
    counter: map int int;   (* sent minus received messages *)
    pending: map int int;   (* messages in transit *)
    token  : token          (* token structure *)
  }
```

Just like in the abstract specification, we formalize what it means for the algorithm to detect termination. In Safra's algorithm, termination is signaled at node 0 when a successful probe concludes. We encode this condition in WhyML with the `terminationDetected` predicate:

WhyML

```
  predicate terminationDetected (w: world) =
    w.token.pos = 0 ∧ w.token.t_color = White ∧ w.color[0] = White ∧
    ¬ w.active[0] ∧ w.token.q + w.counter[0] = 0
```

This formulation effectively replaces the abstract `terminationDetected` Boolean flag from the previous module, computing it instead from the concrete variables. In this case, we bridge the gap between the concrete algorithm and the simple abstract specification using a *refinement mapping*. In WhyML, this is described by a ghost function, `refn`, which takes as argument a state from the `EWD998` module and returns a state from `ATD`, as follows:

WhyML

```
  let ghost function refn (w: world) : ATD.world =
    { ATD.active = w.active; ATD.pending = w.pending;
      ATD.terminationDetected = terminationDetected w }
```

As before, the TLA$^+$ specification includes a *TypeOK* predicate that constrains each variable to its intended domain:

TLA+
$$
\begin{aligned}
&\mathit{TypeOK} \triangleq \\
&\quad \land\ \mathit{active} \in [\mathit{Node} \Rightarrow \text{BOOLEAN}\ ] \\
&\quad \land\ \mathit{color} \in [\mathit{Node} \Rightarrow \mathit{Color}] \\
&\quad \land\ \mathit{counter} \in [\mathit{Node} \Rightarrow \mathit{Int}] \\
&\quad \land\ \mathit{pending} \in [\mathit{Node} \Rightarrow \mathit{Nat}] \\
&\quad \land\ \mathit{token} \in \mathit{Token}
\end{aligned}
$$

Again, Why3's type system handles most of this for free. Though as in the abstract module, $\mathit{pending} \in [\mathit{Node} \Rightarrow \mathit{Nat}]$ requires non-negativity. In addition, the $\mathit{token} \in \mathit{Token}$ conjunct requires the token's position to be a valid node index, $\mathit{pos} : \mathit{Node}$, a constraint not enforced by the record type alone. These two residual obligations make up our `type_ok` predicate:

WhyML
```
predicate type_ok (w: world) =
  (∀ i. node i → w.pending[i] ≥ 0) ∧ node w.token.pos
```

The safety invariant for Safra's algorithm, from Dijkstra's notes and the TLA+ specification, is encoded as the `safe` predicate and constitutes the inductive invariant that must be established and preserved by every action in the concrete module. The comments relay the original description in the notes.

WhyML
```
predicate safe (w: world) =
```

The invariant is structured as a conjunction of two parts. The first, *P0*, captures the global consistency condition: the sum[3] of all node counters must be equal to the number of messages in transit. This ensures no message is lost or fabricated.

WhyML
```
  (* P0: The number of counted messages at each node and
         the number of messages in transit is consistent. *)
∑_i w.counter[i] = ∑_i w.pending[i] ∧
```

The second part, requires at least one of four conditions to hold. The first, *P1*, captures the well-formedness of the token: all nodes beyond the token's position are passive, with each counter being accounted for in the token's accumulator `q`. It is this condition that also allows us to deduce termination.

WhyML
```
  (* At least one of P1, P2, P3, P4 must hold *)
  ((* P1: (Ai: t < i < N: machine i is passive) ∧
           (Si: t < i < N: ci.i) = q *)
    ((∀ i. w.token.pos < i < n → ¬ w.active[i]) ∧
     (if w.token.pos = n - 1
      then w.token.q = 0
      else w.token.q = ∑_{i = w.token.pos + 1}^{n - 1} w.counter[i]))
```

The other conditions, *P2*, *P3*, and *P4* allows us to explain why termination may not yet be safely concluded even when the token has completed a full circuit. *P2* captures invalidations where the net message balance is positive.

WhyML
```
    ∨  (* P2: (Si: 0 ≤ i ≤ t: c.i) + q > 0. *)
    (∑_{i = 0}^{w.token.pos} w.counter[i] + w.token.q > 0)
```

[3] Hereinafter the symbol $\sum$ is used logically; in Why3 it is defined in the Standard Library: `https://why3.org/stdlib/map.html`

P3 captures visited nodes that may have been reactivated upon receiving some message, turning their status to black and thus invalidating the probe:

```
    ∨  (* P3: Ei: 0 ⩽ i ⩽ t : machine nr.i is black. *)          WhyML
   (∃ i. node i ∧ 0 ⩽ i ⩽ w.token.pos ∧ w.color[i] = Black)
```

Finally, *P4* covers the case where the token is itself black:

```
    ∨  (* P4: The token is black. *)                            WhyML
   (w.token.t_color = Black)
 )
```

As before, the conjunction of `safe` and `type_ok` make up our invariant, `inv`:

```
predicate inv w = type_ok w ∧ safe w                            WhyML
```

For brevity, we omit the action translation from the TLA+ specification to WhyML. Though, for future purposes, let us present the `receive_message` action function. When a node receives a message, it decrements the number of outstanding messages and the local counter. It sets its status to active and it also turns its color to `Black` to let the token know that its probe run is invalid. The action function is the following:

```
let ghost function receive_message (w:world) (i: int) : world   WhyML
  requires { receive_message_enabled w i }
  ensures  { inv w → inv result }
  ensures  { inv w → refn w = refn result ∨
             ATD.step (refn w) (refn result) }
=
  let w' = { w with pending = w.pending[i ← w.pending[i] - 1];
                    counter = w.counter[i ← w.counter[i] - 1];
                    active  = w.active[i ← true];
                    color   = w.color[i ← Black] } in
  (* Intermediate assertion discussed in Section 5.2 *)
  w'
```

Although the refinement postcondition is quickly dispatched by the SMT-solvers (less than half a second with 1100 mechanized proof steps), the invariance postcondition is not straightforward. This is due to the nature of infinite domain maps we chose to represent the state. In the TLA+ specification these are finite domain sets. We found infinite maps to be more ergonomic in this example.

```
clone refinement.Refinement with                                WhyML
  type worldC=world, type worldA=ATD.world, val refn,
  predicate invC=inv, predicate invA=ATD.inv,
  val initWorldC=initWorld, val initWorldA=ATD.initWorld,
  val stepC=step, val stepA=ATD.step
```

In the end, Why3 generates 12 VCs, where the SMT-solvers take around 22 seconds to fully verify the module. Most actions and their postconditions are quickly dispatched and though two require an additional auxiliary assertion. These are better described in Section 5.2, meaning we can close this one for now:

```
end                                                             WhyML
```

5 Proving Correctness and Refinement

While the abstract specification of Asynchronous Termination Detection, presented previously, was fully automatic, the verification of Safra's algorithm and the accompanying refinement proof required some manual guidance. Most of the manual guidance was done by following the TLAPS proof from [14].

In Why3, there are essentially two ways to conduct manual proof sessions. The first is through Why3's IDE [4], which allows the user to interactively apply tactics and inspect the proof context. While useful for exploration, proof sessions are stored in a `.why3session` file that is sensitive to source changes and installed SMT-solvers, hindering reproducibility. The second is to embed proof guidance directly in the source as intermediate `assert` statements, breaking complex goals into smaller steps that the SMT solvers can handle individually. Since the guidance lives entirely in the source file, the proof is replayable by simply re-running Why3. We adopt the second approach.

5.1 Verbosity and Organization

The two proof artifacts, in TLAPS and Why3-do, differ substantially in size and organization. The TLAPS proof is spread across two files: `EWD998.tla`, the specification at 217 lines, and `EWD998_proof.tla`, the proof module at 567 lines, totalling 784 lines. The proof module alone declares 30 lemmas and theorems, and, contains 572 lines carrying direct proof obligations via `BY`, `CASE`, `ASSUME`, or `PROVE` clauses, and 185 explicit definition unfoldings via `BY DEF`. The WhyML module combines specification and proof guidance in a single file of 200 lines, with only 4 `assert` statements and 2 lemmas. Table 1 summarises these figures.

Module	Metric	TLA^+	Why3
ATD	Total lines (spec + proof)	123 + 123 = 246	97
	Theorems / Lemmas	5	0
	Numbered proof steps	35	
	Proof-guidance lines	73	0
	Explicit `DEF` unfoldings	22	0
EWD998	Total lines (spec + proof)	217 + 567 = 784	200
	Theorems / Lemmas	30	2
	Numbered proof steps	336	
	Assertions		4
	Explicit `DEF` unfoldings	185	0
	Sum/fold helper lemmas	14	1

Table 1: Statistics across both modules of the EWD998 verification.

The number of proof guidance lines in TLAPS is substantially greater than in WhyML. TLAPS structures proofs as a named, numbered hierarchy of steps (`<1>`, `<2>`, `<3>`, ...), each step being a first-class proof object that can be refered to by

later steps. In TLAPS, we count the proof guidance lines of the type correctness, invariance and refinement theorems, and auxiliary lemmas that are not required in Why3. The resulting structure is a proof tree: wide in the sense that many sibling cases must be handled, but with each leaf discharged by a call to an SMT backend or a first-order prover [16]. In Why3, proof guidance appears, if even needed, as `assert` statements embedded directly into the body of each ghost action function.

The two WhyML lemmas are in respect to map properties, namely, how to carry the sum of a map when an update took place, and a lemma that exploits the non-negativity of the pending map, if the sum of that map is zero then all of its elements are zero. The last one showed itself to be very important to affirm termination. The same lemmas are found in the TLAPS proof.

5.2 The ASSUME/PROVE duality in TLAPS

A recurring pattern in the TLAPS proof is the `ASSUME` ϕ `PROVE` ψ construct, which introduces a local hypothesis ϕ within a named sub-proof step and asks the backend to derive ψ under it. In WhyML, it does not make much sense to use the same `assume`/`assert` duality, as an assumption is introduced in the proof context of the whole action, possibly polluting the assertions that succeed it. We can instead collapse into a single implication assertion:

```
assert { φ → ψ };                                                    WhyML
```

Or as an alternative we can prove ψ by explicitly using ϕ, in which case, we have to also prove ϕ, as expressed in:

```
assert { ψ by φ };                                                   WhyML
```

This is specially useful to dispatch the clause *P0* in the `send_message` and `receive_message` actions. To showcase how this is done, we present the assertion, left hanging in a previous section, of the `EWD998` module that the first clause of the first postcondition holds. The action function is defined as follows:

```
(* Intermediate assertion *)                                         WhyML
assert { inv w → ∑_i w'.counter[i] = ∑_i w'.pending[i] by
                 ∑_i w'.pending[i] = ∑_i w.pending[i] - 1 ∧
                 ∑_i w'.counter[i] = ∑_i w.counter[i] - 1 };
```

This assertion is trivially true from `inv`. The clause *P0* from the invariant captures that the sum of maps `counter` and `pending` are equal. The action performs a decrement in both maps, so to prove that in the post state the maps retain the same sum it suffices to assert that the two sums in `w'` are those from `w` minus one. Here we have to add the invariant implication as we have not yet obtained the postcondition of the function. Adding the invariant as a precondition of the function would not suffice as it would invalidate the functional specification of actions where a function could never be executed in a state where it is not met. The previous assertion, the only one for the `receive_message` function, can be easily contrasted with the TLAPS snippet where to prove the same clause is carried from the trigger of the action, it takes 20 lines of proof statements.

This is in part due to the mature standard library of Why3 and its auto-active nature, where an assertion is a self-contained VC handed to the SMT solvers, which can decide the implication in one shot without needing the hypothesis to be staged separately.

6 Conclusion and Future work

Conclusion. This work presents an alternative methodology for verifying TLA$^+$ specification based on translating actions into functions decorated with contracts and resorting to a Why3 library to describe inductiveness and the notion of refinement, Why3-do. The main motivation for this work is to set forth an automatic translation frontend for TLA$^+$ to WhyML using the Why3-do library, seeing as all of the work presented was done manually. As a by-product, we also aim for the following avenues for future work:

Liveness. In common folklore, safety properties are used to express "nothing bad happens" properties, for instance, the algorithm has detected termination when all nodes are inactive with no pending messages. While liveness properties, which have not yet tackled in this work, express "something good eventually happens", for instance, if all nodes are inactive with no pending messages then the system will at some point detect termination.

From the perspective of Why3-do, action functions can never invalidate safety properties seeing as these inductive invariants reason about traces of unbounded lengths at an arbitrary step. The same is not true for liveness properties. Proving that a system eventually reaches a desired state requires reasoning beyond single-step state transitions, requiring observing infinite execution traces and accounting for fairness conditions, both weak and strong, that dictate action scheduling.

In a deductive verification framework like Why3, tackling liveness traditionally requires the introduction of well-founded relations or variant functions to demonstrate progress. This has been done previously for Ironfleet [11]. To support liveness in our methodology, the Why3-do library would need to be extended to encode TLA$^+$ fairness assumptions into corresponding proof obligations. We would need to formally demonstrate that the system makes strict progress toward the "good" condition, bounding the execution against indefinite stuttering steps. One alternative approach could be to reduce liveness into safety properties using first order logic, something SMT-solvers are really good at. This has been explored in recent works, namely, [17] and [19].

Standard Library. While Why3 and TLA$^+$ have their own standard libraries, they do not natively align in their underlying foundational logic. TLA$^+$ is built upon an untyped Zermelo-Fraenkel set theory with choice, whereas Why3 employs a strongly typed, polymorphic first-order logic. To bridge this semantic gap and enable automated reasoning of TLA$^+$ specifications within Why3, we aim to develop a dedicated standard library in WhyML. This library will capture a significant subset of TLA$^+$ primitives, data structures, and lemmas, in Why3's logic and will be distributed as a self-contained component of Why3-do.

Network models and topologies. Why3-do already provides a mechanism to represent a network using message passing, where each action produces a new state and a list of messages to be processed. The library currently offers a model where messages are allowed to be duplicated. We aim to capture other network failure modes, such as *message loss*, *out-of-order delivery*, or *node crashes*. Furthermore, the specifications presented in this work assume a fully connected network, where any node may communicate with any other. Nonetheless, we intend to relax this assumption by equipping the state machine with a *neighbourhood relation* $\mathcal{N} \subseteq \mathit{Node} \times \mathit{Node}$, constraining message-passing actions so that node i may only send to nodes j with $(i, j) \in \mathcal{N}$, with concrete topologies such as rings or trees provided as separate modules cloned into the algorithm's specification. We have already observed this to be useful in practice: a flood algorithm for computing a minimum spanning tree on a directed acyclic graph can be captured simply by defining the neighbourhood relationship.

Extraction of executable code. A primary advantage of modeling TLA$^+$ specifications within WhyML is the ability to leverage Why3's extraction mechanism to generate executable code. While TLA$^+$ is inherently abstract and designed for specification rather than execution, our translation to WhyML opens the door to leveraging Why3's extraction mechanism, which can, for instance, produce OCaml functions from verified WhyML code. One can imagine that at each refinement step the abstraction targets increasingly fine-grained aspects of the system. Nevertheless, refinement alone can only take us so far. Bridging the gap between an abstract state machine and a concrete and interacting system requires handling system-level I/O, side effects, and network communication, all of which are deliberately abstracted away in the formal model. One approach to this problem is to introduce a final refinement layer that axiomatizes system level operations as drivers. The correctness of the extracted code then rests on two explicit assumptions: the correctness of Why3's extraction mechanism, and the correctness of the drivers themselves.

Acknowledgements. Support for this work was provided by the Fundação para a Ciências e Tecnologias (Portuguese Foundation for Sciences and Technology) under grant 2025.00543.BD, by NOVA LINCS under grant UID/04516/2025 with the financial support of FCT, I.P., by Agence Nationale de la Recherche (ANR) under grant ANR-22-CE48-0013-01 (GOSPEL), and by the DReason project under grant 2024.15569.PEX.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991). https://doi.org/10.1016/0304-3975(91)90224-P
2. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M.,

Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Belo Lourenço, C., Sousa Pinto, J.: Auto-active verification of distributed systems and specification refinements with Why3-do. Science of Computer Programming **247**, 103352 (2026), https://doi.org/10.1016/j.scico.2025.103352
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd Your Herd of Provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wroclaw, Poland (2011), https://inria.hal.science/hal-00790310
5. Chirica, I., Pereira, M., Pinto, J.S.: The TLA+ Trifecta walks into a bar, the bartender ask Why3? https://ionchirica.github.io/isola2026 (2026), Companion artifact
6. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories (2018)
7. Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., Vanzetto, H.: TLA+ proofs. CoRR (2012), http://arxiv.org/abs/1208.5933
8. Dijkstra, E.W.: Shmuel Safra's version of termination detection (1987), https://www.cs.utexas.edu/~EWD/ewd09xx/EWD998.PDF
9. Filliâtre, J.C., Paskevich, A.: Why3 — Where Programs Meet Provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. pp. 125–128. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
10. Filliâtre, J.C., Paskevich, A.: Abstraction and Genericity in Why3. In: ISoLA 2021 - 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. vol. 12476. Rhodes, Greece (2021). https://doi.org/10.1007/978-3-030-61362-4_7
11. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: Ironfleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles. p. 1–17. SOSP '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2815400.2815428, https://doi.org/10.1145/2815400.2815428
12. Konnov, I., Kukovec, J., Tran, T.H.: TLA+ model checking made symbolic. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360549
13. Konnov, I., Kuppe, M., Merz, S.: TLA+ specifications of EWD998 (2021), https://github.com/tlaplus/Examples/tree/ISoLA2022/specifications/ewd998
14. Konnov, I., Kuppe, M., Merz, S.: Specification and verification with the TLA$^+$ Trifecta: TLC, Apalache, and TLAPS. In: Margaria, T., Steffen, B. (eds.) 11th Intl. Symp. Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2022). Lecture Notes in Computer Science, vol. 13701, pp. 88–105. Springer, Rhodes, Greece (2022). https://doi.org/10.1007/978-3-031-19849-6_6
15. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2003), https://books.google.pt/books?id=SeRQAAAAMAAJ
16. Merz, S., Vanzetto, H.: Automatic Verification of TLA + Proof Obligations with SMT Solvers. In: Bjørner, N., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 289–303. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
17. Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., Shoham, S.: Reducing liveness to safety in first-order logic. Proc. ACM Program. Lang. **2**(POPL) (2017). https://doi.org/10.1145/3158114, https://doi.org/10.1145/3158114

18. The Rocq Development Team: The Rocq Prover (2025). https://doi.org/10.5281/zenodo.15149629
19. Yao, J., Tao, R., Gu, R., Nieh, J.: Mostly automated verification of liveness properties for distributed protocols with ranking functions. Proc. ACM Program. Lang. **8**(POPL) (2024). https://doi.org/10.1145/3632877
20. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: In Correct Hardware Design and Verification Methods (CHARME '99), Laurence Pierre and Thomas Kropf editors. Lecture Notes in Computer Science, Springer-Verlag. vol. 1703, pp. 54–66 (1999), https://www.microsoft.com/en-us/research/publication/model-checking-tla-specifications/