Why3-do: The Way of Harmonious Distributed System Proofs

Cláudio Belo Lourenço[1] and Jorge Sousa Pinto[2]

[1] Huawei Research Centre, United Kingdom, claudio.lourenco@huawei.com
[2] HASLab/INESC TEC & Universidade do Minho, Portugal, jsp@di.uminho.pt

Abstract. We study principles and models for reasoning inductively about properties of distributed systems, based on programmed atomic handlers equipped with contracts. We present the Why3-do library, leveraging a state of the art software verifier for reasoning about distributed systems based on our models. A number of examples involving invariants containing existential and nested quantifiers (including Dijsktra's self-stabilizing systems) illustrate how the library promotes contract-based modular development, abstraction barriers, and automated proofs.

1 Introduction

The formal verification of properties of distributed algorithms and protocols is an important and notoriously difficult activity. The dominant approaches are: (i) Automatic exploration of the state space, known as *model checking* [10,4], a technique that can be used for both safety and liveness properties, expressed using variants of temporal logic. Its application to distributed systems is a consolidated area that has held many significant results. However, the *state explosion* phenomenon means that in practice only systems of modest size can be verified. (ii) Deductive reasoning based on the use of *inductive invariants*. A number of tools [26,18,13] now exist for the verification of single-threaded systems based on first-order logic (FOL), loop invariants, and contracts, with solid theoretical foundations [21,16]. Reasoning about distributed systems using inductive invariants was, until recently, mostly a pen-and-paper activity, but tools like Verdi [42], IronFleet [20], and Ivy [34] have made significant advances to this state of things (see Section 7 for details). Relying on external provers (and in the case of IronFleet, on the Dafny verifier to check the sequential code), these tools support verification of asynchronous message-passing systems based on atomic handlers, reusable network/fault models, and different abstract specification mechanisms.

Based on the same principles, we propose in this paper a conceptual contract-based framework for reasoning about distributed systems, as well as the Why3-do library for the Why3 verifier [18]. Distinctive aspects of our approach include the following:

– It allows for reasoning about distributed systems using a standard program verification tool (rather than a dedicated tool or a proof assistant), and methods and techniques that are standard for sequential software.

I. Sergey (Ed.): ESOP 2022, LNCS 13240, pp. 114–142, 2022.
https://doi.org/10.1007/978-3-030-99336-8_5

- Systems and protocols are described algorithmically by means of programmed handlers equipped with *contracts* that guarantee the inductiveness of invariants. Thus Why3-do brings modular development using the popular programming by contract methodology to the scope of distributed systems.
- Why3-do offers other system models in addition to message-passing. We illustrate this in this paper by describing a locally shared memory model.
- It takes advantage of Why3's state of the art proof management (including replayability, bisection of hypotheses, and inconsistency detection); ability to interact with all major proof tools (automated and interactive); and internal transformations that allow for a combination of interactive and automated development, avoiding the use of proof assistants for inductive proofs.

Contributions of the Paper. We contribute to the state of the art of distributed system verification, and in general to software verification with Why3:
(i) We introduce (Section 3) principles for modular verification of distributed systems based on clonable models, capturing in a uniform way different system semantics. Each model declares a set of handlers equipped with contracts.
(ii) We present (sections 4, 5, 6) a Why3 library with different system models and fault semantics. A concrete system is defined by cloning a model and defining its handlers and invariants. Handler implementations are required to respect the contracts declared in the model, which in particular ensures inductiveness of the invariants. Although Dafny contracts can also be used in IronFleet, the novelty in Why3-do is the presence of dedicated contracts in the library models, that are used to automatically generate verification conditions when cloning.
(iii) We introduce (Section 5) a model-independent specification mechanism based on system traces, to act as abstraction barrier between specification (observable properties) and implementation. Traces are a common specification mechanism; the novelty here is the support for modular development through the use of model-independent *clonable specification modules*; different implementations can be given for a specification, using different system models.
(iv) We present (Section 6) a locally-shared memory model illustrating how our approach is applied uniformly beyond message-passing models. As far as we are aware Verdi, IronFleet and Ivy work with message-passing systems only.
(v) We formalize and verify one of Dijsktra's self-stabilizing systems [15] and verify its closure (safety) and convergence (liveness) properties using Why3-do. This verification is of independent interest: our proof of convergence, using a measure function, takes advantage of SMT solvers and significantly improves on previous, much more laborious efforts using proof assistants (Section 6).
(vi) We propose two techniques for reasoning with inductive invariants containing existential and nested quantifiers: *stepwise bounded validation* (Section 6), and the use of *dual definitions* containing both code and logic (sections 4 and 6). Together with Why3's ability to interact with multiple solvers with different strengths, dual definitions allow for more robust and natural specifications, as well as for easier automated proofs, without the need for tricks like quantifier hiding [20]. Both techniques are explained by means of examples.

```
module MapList
use int.Int, list.List, list.Mem, list.Length, list.NthNoOpt

val function f (x:int) : int    requires {x >= 0}    ensures {result >= 0}

predicate nonNeg (l:list int) = forall x :int. mem x l -> x >= 0

let rec map_list (l:list int) : list int
  requires { nonNeg l }
  ensures { nonNeg result /\ forall j. 0<=j<length l -> nth j result  = f(nth j l) }
  variant { l }
= match l with
  | Nil -> Nil
  | Cons h t -> Cons (f h) (map_list t)
  end
end (* module MapList *)

module MapFib
use int.Int, list.List, list.Mem, list.Length, list.NthNoOpt, ref.Ref

inductive fibpred int int =
| zero : fibpred 0 0
| one : fibpred 1 1
| oth : forall n r1 r2 :int. n>=2 -> fibpred (n-1) r1 /\ fibpred (n-2) r2 -> fibpred n (r1+r2)

let function calcfib (m:int) : int
  requires { m >= 0 }
  ensures { result >= 0 /\ forall r. fibpred m r <-> r=result }
= let n = ref 0 in let x = ref 0 in let y = ref 1 in
  while !n < m do
    invariant { 0 <= !n <= m /\ !x >= 0 /\ !y >= 0 }
    invariant { forall r. (fibpred !n r <-> r = !x) /\ (fibpred (!n+1) r <-> r = !y) }
    variant { m - !n }
    let tmp = !x in x := !y; y := !y+tmp; n := !n+1;
  done;
  !x

clone MapList  with val f = calcfib

lemma mapFib_lm: forall l:list int.nonNeg l-> let fibl = map_list l in
                    nonNeg fibl /\ forall j.0<=j<length l-> nth j fibl = calcfib (nth j l)
end (* module MapFib *)
```

Listing 2.1. Why3 example

All the models and example modules mentioned in the paper are available for experimentation in the Why3-do artifact [28].

2 The Why3 Languages in a Nutshell

The example in Listing 2.1 illustrates the use of Why3's logic and programming languages, as well as the module cloning mechanism. The `MapList` module first imports a number of theories for mathematical integers and lists from the standard library. Why3 includes a wide range of theories, usable across provers. A program function `f` is then declared with the `val` keyword, including a simple contract: a precondition requiring its argument to be nonnegative, and a postcondition stating that the result is also nonnegative. In the rest of the module this contract will be assumed to hold for `f`. Next, a logic predicate `nonNeg`

is defined. It uses a universal quantifier to state that every element of its argument list is nonnegative. Finally, the `map_list` program function is defined. The definition includes both the function's recursive definition and a contract, in particular a postcondition that uses a universal quantifier to state the mapping property (`result` refers to the return value). From this module, Why3 will generate verification conditions (VCs) ensuring that the definition is consistent with its contract, *assuming the definition of `f` keeps to its own contract.* This interplay between contracts plays a fundamental role in deductive verification.

This little example allows us to elaborate on another aspect of Why3. `nonNeg` is also a function (returning a truth value), but it lives in a different namespace from `map_list`, which is a WhyML *program function*. `nonNeg` belongs to Why3's *logic language* [17], and its definition contains a quantifier, which cannot be used in programs. However, *pure* program functions, which do not modify the global state, may also be used in the logic, if their declaration includes the `function` keyword. This is the case of `f`, used in both the code and the contract of `map_list`. We will refer to program functions that can be used in the logic as "let functions". `map_list` is also pure, but is not declared as a let function.

Why3 encodes both the code and contracts of let functions, so one may choose to write certain logic functions algorithmically or logically, or both. For instance `nonNeg` could be defined alternatively as follows (the postcondition is optional):

```
let rec predicate nonNeg (l:list int)
  ensures { result <-> forall x :int. mem x l -> x >= 0 }
= match l with
  | Nil -> true  | Cons h t -> h>=0 && nonNeg t  end
```

If the postcondition is present, the logic encoding of the predicate will contain redundancy (no inconsistency can be created since the definition must respect the contract). Writing such "dual definitions" of logic functions may be a good idea for a number of reasons, namely the possibility of including preconditions, and termination checks based on user-provided variants. Moreover, dual definitions increase the robustness of specifications and may facilitate automated proofs of results involving quantifiers. Not every logic function can be defined as a let function: since the latter must remain executable, they may not contain for instance occurrences of logic equality or quantifiers. In these cases *let ghost functions* can be used. These are pure logic definitions that are not meant to be executed, but are still written as programs.

A second module, `MapFib`, defines a program function `calcfib` that computes Fibonacci numbers using a loop. The recursive definition of the Fibonacci sequence (used in the function and loop invariant of `calcfib`) cannot be written as a logic function, since it is not total. It could be defined as a let function with a precondition restricting its domain, but we use instead an *inductive predicate* `fibpred`: the formula `fibpred n f` means that `f` is the `n`th. Fibonacci number. Inductive predicates, familiar to readers acquainted with proof assistants, are defined by means of a set of inference rules. They are used in our models to define non-deterministic transition relations on distributed system configurations.

Why3 will generate and successfully discharge VCs ensuring the correctness of `calcfib` with respect to its contract. Now, since `calcfib` is in accordance with

the contract of `f` in `MapList`, this module can be *cloned* instantiating the latter function with the former. This imports into the current module a copy of every element of `MapList`, with `calcfib` substituted for `f`, and generates *refinement VCs*, to ensure that `calcfib`'s contract is stronger than `f`'s. Finally, the lemma `mapFib_lm` states that indeed `map_list` maps the function `calcfib` as expected.

3 Distributed Systems and Models

A distributed system consists of a set $\mathbf{N}$ of *nodes*, each of which can at any moment be in a state taken from a set Σ, together with additional elements, such as a communication network or a shared memory. We will call the global state of such a system a *world* and denote by $\mathbf{W}$ the set of all worlds. In general, worlds will include the local state of every node in the system, captured as a mapping $\mathsf{IS} : \mathbf{N} \to \Sigma$. Different models will specialize this basic setting to define different notions of distributed system (and consequently also of world), including for instance different communication and fault models (we will always write $\mathbf{N}$, Σ, or $\mathbf{W}$ in the context of a specific system model, left implicit).

Models are *handler-based*: systems are described by writing code executed by nodes in response to certain events, such as receiving a message from the network or an input from the local environment, or simply being enabled by a guard predicate that becomes true. Handlers are assumed to execute atomically. Each model defines a transition semantics describing how worlds evolve step by step, allowing for all possible schedules (both locally and globally). Each model contains a set of rules inferring judgments of the form $w \rightsquigarrow w'$, meaning that the system's global state w evolves to w'. The general form of the rules states the following: *if the world w' results from w when a handler is executed by one of the system's nodes, then $w \rightsquigarrow w'$.*

Let w_0 correspond to the initial state of the system, and $\rightsquigarrow^*$ denote the reflexive-transitive closure of $\rightsquigarrow$. A world w is said to be *reachable* if $w_0 \rightsquigarrow^* w$. Let Φ be some property of worlds; we will write $w \models \Phi$ to signify that Φ is satisfied by the world w . A system is said to be *correct with respect to Φ* if $w \models \Phi$ holds *for every reachable world w*. A typical correctness proof involves finding an *inductive invariant*: a property I such that (i) $w_0 \models I$, and (ii) for every pair w, w' of worlds, if $w \models I$ and $w \rightsquigarrow w'$, then $w' \models I$. If $w \models I$ implies $w \models \Phi$, this is sufficient to guarantee correctness.

Contract-based Models. We introduce the use of handler contracts for designing and verifying distributed systems. Let us consider a model with worlds of the form $\langle \mathsf{IS}, \ldots \rangle$, with $\ldots$ standing for other components of worlds in addition to the state function. The signature and contract of a handling function will be of the following general form, where I is a candidate invariant predicate, and other arguments and return values $(\ldots)$ may be present:

$$\mathsf{handle}(n : \mathbf{N}, \mathsf{IS} : \mathbf{N} \to \Sigma, \ldots) : (\sigma : \Sigma, \ldots)$$
$$\textbf{requires } I\langle \mathsf{IS}, \ldots \rangle$$
$$\textbf{ensures } I\langle \mathsf{IS}[n \mapsto \sigma], \ldots \rangle$$

The function returns the new state σ of the node n that executes the handler in a world with state function IS. This general form will be adapted with modifications in different models. For instance, handling functions may have access only to the local state and not to the entire state function IS, or they may return, in addition to a new state, a list of messages to be sent by n. Transition rules have the following general form, updating the state of the node that executes the handler, and reflecting in the world other effects of the execution.

$$\frac{\mathsf{handle}(n, \mathsf{IS}, \ldots) = (\sigma, \ldots)}{\langle \mathsf{IS}, \ldots \rangle \rightsquigarrow \langle \mathsf{IS}[n \mapsto \sigma], \ldots \rangle}$$

The handler's contract, consisting of precondition $I\langle \mathsf{IS}, \ldots \rangle$ and postcondition $I\langle \mathsf{IS}[n \mapsto \sigma], \ldots \rangle$, ensures that if the handler is executed in a world satisfying the invariant I, then the world resulting from this transition still satisfies I.

It is common for handlers to have access only to the state σ of the node n where they are being executed. In this case it is not possible to include $I\langle \mathsf{IS}, \ldots \rangle$ as a precondition in the contract, since IS is not passed as a parameter. Preservation of the invariant can be written instead as a conditional postcondition, stating that *for every world satisfying I in which σ is the state of node n and this node executes the handler, then the resulting world still satisfies I*:

$$\begin{array}{l} \mathsf{handle}(n : \mathbf{N}, \sigma : \Sigma, \ldots) : (\sigma' : \Sigma, \ldots) \\ \textbf{ensures } \forall_{\mathsf{IS}:\mathbf{N}\to\Sigma}.\ \sigma = \mathsf{IS}\, n \ \to\ I\langle \mathsf{IS}, \ldots \rangle \ \to\ I\langle \mathsf{IS}[n \mapsto \sigma'], \ldots \rangle \end{array}$$

The Why3-do Library. Listing 3.1 illustrates how contract-based models are written as Why3 modules. The `World` module declares basic types and functions, and defines the `world` structured type. The `Steps` module includes `val` declarations for (i) the *initial world*, (ii) an *inductive invariant predicate*, and (iii) a set of *handling functions* (illustrated here by `handle_1`). Contracts enforce that the inductive invariant is satisfied by the initial world, and preserved by handlers. Each handler's contract makes use of a `step_1` auxiliary function, that is also used in the definition of the transition semantics through the `step` inductive predicate. The module ends with the definition of reachable world, and a lemma stating that the invariant holds in all reachable worlds (this is proved inductively for each model, using proof transformations and SMT solvers).

That is all that is required to define a system model, which may now be cloned to produce concrete distributed systems. Listing 3.2 illustrates how simple this is. We write a `System` module that defines, first of all, *types* for nodes, states, messages, and other relevant elements, and if appropriate, *well-formedness predicates* for different entities. The `World` module from the desired Why3-do library model can then be *cloned*, after which the following are defined: (i) the *initial world*, (ii) a candidate *inductive invariant* predicate, and (iii) *handler functions* specifying the behavior of the system's nodes/processes. The `Steps` module from the same model is now cloned, instantiating these elements. Why3 will produce a set of VCs, generated from the contracts contained in the cloned module, ensuring that the invariant is inductive. Properties of interest can at last be stated and proved (which may involve writing additional definitions and lemmas).

```
module World     (* file model.mlw *)
type node
type state
type world =  (map node state, ...)
function localState (w:world) : map node state =      (* projection functions for worlds *)
  let (lS, ...) = w in lS
end (* module World *)

module Steps     (* file model.mlw *)
...
val function initState (node) : state             (* init functions for world components *)
constant initWorld : world = (initState, ...)

val ghost predicate indpred (w:world)
  ensures { w=initWorld -> result }              (* initial world must satisfy invariant *)

(* specifying the new world that results from w when n executes a handler yielding results r *)
function step_1 (w:world) (n:node) (r:(state, ...)) : world =
  let (st, ...) = r in
    let newLocalState = set (localState w) n st in
      (newLocalState, ...)

(* handlers' arguments include a node h and its state; results include a new state for h *)
val function handle_1 (h:node) (sig:state) ... : (state, ...)
  ensures  { forall w :world. indpred w -> sig = localState w h -> ... ->
               indpred (step_1 w h result) }

inductive step world world =
| step_1 : forall w :world, n :node.
           step w (step_1 w n (handle_1 n (localState w n) ...))
| ...

inductive step_TR world world =
| base : forall w :world. step_TR w w
| step : forall w w' w'' :world. step_TR w w' -> step w' w'' -> step_TR w w''

predicate reachable (w:world) = step_TR initWorld w

(* inductive invariant holds in all reachable worlds *)
lemma indpred_reachable : forall w :world. reachable w -> indpred w
end (* module Steps *)
```

Listing 3.1. Basic structure of a Why3-do model

4 The Basic Message-Passing Model

In this model nodes communicate by exchanging *packets*: triples of the form (d, s, m), carrying a message $m \in \mathbf{Msg}$ from node $s \in \mathbf{N}$ to node $d \in \mathbf{N}$, with $\mathbf{Msg}$ a given set of *messages*. Worlds are pairs $\langle \mathsf{lS}, \mathsf{nt} \rangle$ where $\mathsf{lS} : \mathbf{N} \to \Sigma$ is a function assigning a state to each node and $\mathsf{nt} : \mathbf{Msg}^*$ is a network, abstracted as a list of packets. In a system based on this asynchronous model, nodes execute a *message handler* whenever they receive a message, and may in turn send messages to other nodes. The $\mathsf{handleM}$ function implements this local message-handling behavior. Its parameters include the node h handling the message, the node that sent the message, the state of the handling node, and the message itself. It returns a new state for h and a list of packets to be sent to the network.

```
module System     (* file system.mlw *)
type node = int
type state = int
clone model.World with type node, type state

let function initState (n:node) : state = ...

let ghost predicate indpred (w:world) = ...

let function handle_1 (h:node) (lS:map node state) : state = ...

clone model.Steps with type node, type state, val initState, val indpred, val handle_1

goal systemProperty : forall w :world. reachable w -> ...

end (* module System *)
```

Listing 3.2. Basic structure of a Why3-do system module

Its signature and contract are (with I a candidate invariant):

$$\begin{aligned}&\mathsf{handleM}(h : \mathbf{N}, s : \mathbf{N}, m : \mathbf{Msg}, \sigma : \Sigma) : (\sigma' : \Sigma, \mathsf{nt}' : \mathbf{Msg}^*)\\ &\textbf{ensures } \forall_{\mathsf{lS}:\mathbf{N}\to\Sigma,\mathsf{nt}:\mathbf{Msg}^*}.\ \sigma = \mathsf{lS}\,h \to (h,s,m) \in \mathsf{nt}\\ &\quad \to\ I\langle \mathsf{lS},\ \mathsf{nt}\rangle \to I\langle \mathsf{lS}[h \mapsto \sigma'],\ \mathsf{nt}' + \mathsf{nt} - \{(h,s,m)\}\rangle)\end{aligned}$$

The semantics of the model are given by the following transition rule:

$$\frac{\mathsf{handleM}(h,s,m,\mathsf{lS}(h)) = (\sigma,\mathsf{nt}') \qquad (h,s,m) \in \mathsf{nt}}{\langle \mathsf{lS}\ ,\ \mathsf{nt}\rangle \rightsquigarrow \langle \mathsf{lS}[h \mapsto \sigma]\ ,\ \mathsf{nt}' + \mathsf{nt} - \{(h,s,m)\}\rangle}\ (\mathit{message})$$

We use notation $+$, $-$, and $\in$ for list concatenation, difference, and membership. Any packet that is in transit in the network may be selected by the rule to be delivered and handled by the receiving node. The rule removes the packet from the network, updates the state of the handling node, and sends new packets as prescribed by the handler. The semantics takes into account all possible orders of message delivery, since any message may be extracted from the packet pool. The semantics is otherwise idealized, but the library contains additional models in which messages may be dropped or duplicated by the network (an example verification of a system assuming message duplication is given in Section 5).

The contract of handleM ensures that executions of *(message)* preserve the invariant I. Let $ok^I(\mathsf{handleM})$ signify that the implementation of the handler adheres to its contract, instantiated with the candidate invariant I. If I holds in the initial world then it is indeed inductive and holds in all reachable worlds:

Lemma 1. *Let $w_0, w \in \mathbf{W}$ and I be a predicate such that $ok^I(\mathsf{handleM})$. If $w_0 \models I$ and $w_0 \rightsquigarrow^* w$ then $w \models I$.*

A simplified version of the corresponding Why3-do model is shown in Listing 4.1. The `World` module defines the tuple types `packet` and `world` and auxiliary functions. `Steps` declares the following elements to be instantiated when cloning: the `ok_Msg` well-formedness predicate; `initState` and `initMsgs`,

```
module World
type node     type state     type msg
type packet = (node, node, msg)
function dest (p:packet) : node = let (d,_,_)=p in d
function src (p:packet) : node = let (_,s,_)=p in s
function payload (p:packet) : msg = let (_,_,m)=p in m
type world = (map node state, list packet)
function localState (w:world) : map node state = let (lS,_)=w in lS
function inFlightMsgs (w:world) : list packet = let (_,ifM)=w in ifM
end (* module World *)

module Steps
...
predicate ok_Msg (node) (node) (msg)

val function initState (node) : state
val constant initMsgs : list packet
constant initWorld : world = (initState, initMsgs)

val ghost predicate indpred (w:world)
  ensures { w=initWorld -> result }
  ensures { result -> forall p: packet. mem p (inFlightMsgs w) ->
              ok_Msg (dest p) (src p) (payload p) }

function step_message (w:world) (p:packet) (r:(state, list packet)) : world
= let (st, ms) = r in let localState = set (localState w) (dest p) st in
  let inFlightMsgs = ms ++ (remove p (inFlightMsgs w)) in (localState, inFlightMsgs)

val function handleMsg (h:node) (s:node) (m:msg) (sig:state) : (state, list packet)
  requires { ok_Msg h s m }
  ensures { forall w :world. indpred w -> mem (h, s, m) (inFlightMsgs w) ->
              sig = localState w h -> indpred (step_message w (h, s, m) result) }

inductive step world world =
| step_msg : forall w :world, p :packet. mem p (inFlightMsgs w) ->
  step w (step_message w p
    (handleMsg (dest p) (src p) (payload p) (localState w (dest p))))

inductive step_TR world world = ...
predicate reachable (w:world) = step_TR initWorld w

lemma indpred_reachable : forall w :world. reachable w -> indpred w
end (* module Steps *)
```

Listing 4.1. Message-passing model: `modelMP`

used to construct `initWorld`; the inductive invariant `indpred`; and finally the `handleMsg` handler. The contract of `indpred` ensures that it is satisfied by the initial world, and that all messages in the network are well-formed. Well-formedness conditions are singled out from the invariant because the handler function may need to assume basic facts about messages. The module ends with lemma `indpred_reachable`, corresponding to Lemma 1 (the $ok^I(\mathsf{handleM})$ and $w_0 \models I$ premises are enforced by the contracts of `indpred` and `handleMsg`). It is proved using a Why3 transformation for predicate induction, and SMT solvers.

Example: Leader Election on a Ring. *Leader Election* is a *coordination* problem, where a set of processes or nodes collectively designate one of them to act as leader. One of the simplest solutions to this problem on a unidirectional ring network is the maximum-finding distributed algorithm devised by Chang and

Roberts [7]. Let each node have a distinct identifier of some type equipped with a total order relation. Informally the algorithm can be described as follows: (i) messages are node identifiers; each node starts by sending its id to the next node in the ring. (ii) Each node then enters a message-handling loop. If a received message has a higher value than the receiver's id, the message is forwarded to the next node. Otherwise, it is discarded. (iii) If a node receives back a message with its own id, it claims to be the leader. The fundamental property to be proved of this system is that *at most one node claims to be leader*. The system has been used as example in [34] and later in [29]. The Ivy description of the system is based on the decidable EPR fragment of FOL (See Section 7), whereas our formalization below uses unrestricted quantification.

The Why3-do encoding of this algorithm is given in Listing 4.2, based on the `modelMP` library model. The first step is to define types for nodes, identifiers, states, and messages. Identifiers are uniquely associated to nodes by means of the `id` function and the `uniqueIds` axiom. The constant `n_nodes` is the number of nodes in the ring. A minimum of 3 nodes is assumed, with no upper bound. The constant `maxId_global` corresponds to the (unique) node having the *highest-value id* in the ring. Node states are records having a single field `leader` of Boolean type, which indicates when a node claims to be leader. The `ok_Msg` predicate describes the notion of *well-formed message* in the ring topology.

The types for nodes and identifiers could be left undefined, with a set of axioms for the `next` function and the `maxId_global` constant. But in our experience, using library types, as well as defined constants, predicates, and functions when adequate, is advantageous from the point of view of provability, and also reduces the danger of introducing inconsistencies. For instance the `maxId_global` constant is defined algorithmically using a recursive let function `maxId_fn` with a "dual definition" (it is equipped with a contract describing precisely what it does). We could instead simply write an axiom concerning `maxId_global`, but using the dual definition let function, containing code, not only increases the degree of assurance in what is being specified, but also makes it easier to reason about, since Why3 will generate a more easily provable set of VCs.

Cloning the module `modelMP.World` introduces new composed types and auxiliary definitions. The system description then proceeds to give the initial conditions of the system, by means of a state function `initState`, and a constant `initMsgs` for the list of messages that are sent upon booting, also defined by means of a recursive let function. The handler definition then follows. The next element in the module is the invariant `indpred`, defined as a let predicate (since logic elements like quantifiers and equality are required, it is defined as a `let ghost predicate` using an auxiliary predicate `inv`, see Section 2). It states that every inflight message is well-formed; it contains the id of some node in the ring, with value not less than the sender's id, and it is not the id of any node `i` such that `maxId_global` is located between `i` and the message's destination node (an auxiliary predicate `between` is used to express this). Moreover if the message contains its destination's id then that id is the highest in the network. Finally, any node that is claiming to be the leader has the highest id in the ring.

```
type node = int
val constant n_nodes : int
axiom n_nodes_ax : 3 <= n_nodes
let function next (x:node) : node = mod (x+1) n_nodes

type id = int
val function id (node) : id
axiom uniqueIds : forall i j :node. id i = id j <-> i=j

let rec function maxId_fn (n:int) : node
  requires { 1 <= n <= n_nodes }
  ensures { 0 <= result < n}
  ensures { forall k :node. 0<=k<n -> k<>result -> id k < id result}
  variant { n }
= if n=1 then 0
  else let m = maxId_fn (n-1) in if id (n-1) > id m then n-1 else m

constant maxId_global : id = maxId_fn n_nodes

type state = { leader : bool }

type msg = id
predicate ok_Msg (dest:node) (src:node) (m:msg) =
  0 <= dest < n_nodes /\ 0 <= src < n_nodes /\ dest = next src

clone modelMP.World with type node = node, type state = state, type msg = msg

let function initState (i:node) : state = { leader = false }

let rec function initMsgs_fn (n:node) : list packet
  requires { 0<=n<=n_nodes }
  ensures  { forall s d :node, m :msg. mem (d, s, m) result ->
               m = id s /\ d = next s /\ n<=s<n_nodes /\
             (forall i :node. between i maxId_global d -> m <> id i) /\
             (m = id d -> d = maxId_global) }
  variant { n_nodes-n }
= if (0<=n<n_nodes) then Cons (next n, n, id n) (initMsgs_fn (n+1))
  else Nil

let constant initMsgs : list packet = initMsgs_fn 0

let function handleMsg (h:node) (src:node) (m:msg) (s:state) : (state, list packet)
= if m = (id h) then ({ leader = true }, Nil)
  else if m > id h then (s, Cons (next h, h, m) Nil)
               else (s, Nil)

predicate between (lo:node) (i:node) (hi:node) =
  (lo < i < hi) \/ (hi < lo < i) \/ (i < hi < lo)

lemma btw_next_lm : forall i j k :node.
  0 <= i < n_nodes -> 0 <= j < n_nodes -> 0 <= k < n_nodes -> i <> k ->
    between (next i) j k -> between i j k

predicate inv (lS:map node state) (iFM:list packet) =
  (forall s d :node, m :msg. mem (d, s, m) iFM ->
    (ok_Msg d s m /\ m >= id s /\
    (exists i :node. 0 <= i < n_nodes /\ m = id i) /\
    (forall i :node. between i maxId_global d -> m <> id i) /\
    (m = id d -> d = maxId_global) )) /\
  (forall i:node. 0<=i<n_nodes -> (lS i).leader = true -> i = maxId_global)

let ghost predicate indpred (w:world) = inv (localState w) (inFlightMsgs w)

clone modelMP.Steps with type node, type state, type msg, predicate ok_Msg,
  val initState, val initMsgs, val indpred, val handleMsg

goal uniqueLeader :
  forall w :world, i j:node.
    reachable w -> 0<=i<n_nodes -> 0<=j<n_nodes ->
      (localState w i).leader = true -> (localState w j).leader = true -> i = j
```

Listing 4.2. Leader election on a ring (Chang-Roberts)

The module then clones the `Steps` module from `modelMP` instantiating the necessary elements, and formulates the `uniqueLeader` proof goal. The verification results depend on the provers that are available. In our setup we were able to prove automatically all VCs using the Alt-Ergo [11], CVC4 [5], and Vampire [36] SMT solvers after (i) providing lemma `btw_next_lm`, proved automatically by Alt-Ergo; and (ii) including in the postcondition of function `initMsgs_fn` the relevant facts relating in-transit messages and `maxId_global`, as required by the invariant. Observe that this postcondition is proved automatically by the program verification engine following the recursive definition of the function.

5 Trace Specifications

In the previous section we have considered a specification property expressed at the implementation level, with access to internal node states. Other internal elements of worlds, including messages, could be mentioned in such implementation-level properties. It is however very useful to introduce an *abstraction barrier* between specifications and implementation details. This can be achieved by logging certain *observable events* onto a *trace* of the system, and then writing specifications as properties of the trace. Models in our setting can be equipped with traces, allowing for protocols and systems to be specified in this way.

We will illustrate this by equipping the message-passing model of Section 4 with traces. Each system using this model defines an $\mathbf{Out}$ type of *outputs*, and the model defines *external events* as $\mathbf{Evt} = \mathbf{N} \times \mathbf{Out}$, outputs paired with the node that originated them (other models may use additional notions of external event, such as inputs received by nodes from their local environments). A trace is a sequence of external events; the function $\mathrm{rec} : \mathbf{N} \to \mathbf{Out}^* \to \mathbf{Evt}^*$ produces a trace from a sequence of outputs, pairing them with the source node. Given a predicate ν on traces and $\tau \in \mathbf{Evt}^*$, we will write $\tau \models \nu$ when τ satisfies ν.

A *commit specification* (μ_p, μ_f) consists of a predicate $\mu_p(\Sigma, \Sigma)$ and a function $\mu_f(\Sigma, \Sigma) : \mathbf{Out}^*$, expressing respectively when outputs should be produced, and what those outputs should be. The signature of the message handler is similar to that in the model of Section 4, with a trace as additional output. Its contract states that it complies with a given commit specification.

$$
\begin{array}{l}
\mathsf{handleM}(h : \mathbf{N}, s : \mathbf{N}, m : \mathbf{Msg}, \sigma : \Sigma) : (\sigma' : \Sigma, \mathsf{nt}' : \mathbf{Msg}^*, l : \mathbf{Out}^*) \\
\textbf{ensures } \forall_{\mathsf{IS}:\mathbf{N}\to\Sigma, \mathsf{nt}:\mathbf{Msg}^*}.\ \sigma = \mathsf{IS}\, h \to (h, s, m) \in \mathsf{nt} \\
\quad I\langle \mathsf{IS},\ \mathsf{nt}\rangle \to I\langle \mathsf{IS}[h \mapsto \sigma'],\ \mathsf{nt}' + \mathsf{nt} - \{(h, s, m)\}\rangle) \\
\textbf{ensures } (\mu_p(\sigma, \sigma') \to l = \mu_f(\sigma, \sigma')) \wedge (\neg \mu_p(\sigma, \sigma') \to l = \varepsilon)
\end{array}
$$

We will write $ok^{I,\mu_p,\mu_f}(\mathsf{handleM})$ to signify that the implementation of $\mathsf{handleM}$ adheres to its contract, with invariant I and commit specification (μ_p, μ_f).

Worlds are tuples $\langle \mathsf{IS}, \mathsf{nt}, \tau\rangle$ with $\mathsf{IS} : \mathbf{N} \to \Sigma$, $\mathsf{nt} : \mathbf{Msg}^*$, and $\tau : \mathbf{Evt}^*$. The semantics will now be given by the relation $\rightsquigarrow \subseteq \mathbf{W} \times \mathbf{N} \times \mathbf{W}$, with $w \rightsquigarrow_n w'$ meaning that world w transitions to w' with node n executing a handler. The following transition rule commits outputs to the trace:

$$\frac{\mathsf{handleM}(h, s, m, \mathsf{IS}(h)) = (\sigma, \mathsf{nt}', l) \qquad (h, s, m) \in \mathsf{nt}}{\langle \mathsf{IS}\,,\, \mathsf{nt}\,,\, \tau \rangle \rightsquigarrow_h \langle \mathsf{IS}[h \mapsto \sigma]\,,\, \mathsf{nt}' + \mathsf{nt} - \{(h, s, m)\}\,,\, \mathrm{rec}_h(l) + \tau \rangle} \; (message)$$

A *specification* is a triple (μ_p, μ_f, ν) consisting of a commit specification and a predicate $\nu(\mathbf{Evt}^*)$ expressing some notion of trace consistency. Correctness implies that the commit specification is respected and traces are consistent.

Definition 1. *A system with initial world $w_0 \in \mathbf{W}$ is said to be* correct *with respect to a specification (μ_p, μ_f, ν) if*

1. *for all $w = \langle \mathsf{IS}, \mathsf{nt}, \tau \rangle \in \mathbf{W}$, $w' = \langle \mathsf{IS}', \mathsf{nt}', \tau' \rangle \in \mathbf{W}$ and $n \in \mathbf{N}$ such that $w_0 \rightsquigarrow^* w \rightsquigarrow_n w'$, if $\mu_p(\mathsf{IS}(n), \mathsf{IS}'(n))$ then $\tau' = rec_n(\mu_f(\mathsf{IS}(n), \mathsf{IS}'(n))) + \tau$, otherwise $\tau' = \tau$*
2. *$\tau \models \nu$ for every world $w = \langle \mathsf{IS}, \mathsf{nt}, \tau \rangle \in \mathbf{W}$ such that $w_0 \rightsquigarrow^* w$*

Lemma 2. *Let (μ_p, μ_f, ν) be a specification, and I a predicate such that $ok^{I,\mu_p,\mu_f}(\mathsf{handleM})$, $w_0 \models I$, and for every world $w = \langle \mathsf{IS}\,,\, \mathsf{nt}\,,\, \tau \rangle$, $w \models I$ implies $\tau \models \nu$. Then the system is correct with respect to (μ_p, μ_f, ν).*

As usual the lemma is proved mechanically in the Why3-do module for this model. Every Why3-do model extended with traces contains a similar lemma.

A simplified version of the `modelMPTrace` model is shown in Listing 5.1 (`...` indicate elements that are preserved from the `modelMP` module). The `world` type extends the tuple of `modelMP` with a trace of type `list externalEvent`. The functions/predicates `commitp`, `commitf`, and `consistent`, corresponding respectively to μ_p, μ_f, and ν, are to be instantiated when cloning the model. The `indpred` inductive predicate gains a new postcondition ensuring that it enforces consistency of the system's trace (following the conditions of Lemma 2). The `step` inductive predicate is modified to include as an additional parameter the node involved in each transition. The `commit_step` and `consistent_reachable` lemmas (mechanically proved, using the contracts of `indpred` and `handleMsg`) together correspond to Lemma 2 above.

Example: Distributed Lock. This example will show how Why3-do models can be extended in a flexible way. Its verification was first carried out in [20] and later also in [34] and [29]. We adapt it here to make use of trace specifications, which will allow us to demonstrate their effectiveness as an abstraction barrier. In addition to traces, the example also illustrates the use of guarded actions in models (through the use of *enabling predicates*), as well as the use of a non-idealized network model, in which in-transit messages can be duplicated. Two implementations will be given: one that is in accordance with the trace spec if the idealized model is used, and a second implementation that tolerates duplicating messages. The specification of the distributed lock system is the following:

1. the state of each node must include information on whether it is holding a lock (a Boolean), together with the lock's current *epoch* (an integer);
2. whenever a node acquires a lock it outputs its current epoch;

```
module World    ...
type externalEvent    ...
type world = (map node state, list packet, list externalEvent)    ...
function trace (w:world) : list externalEvent = let (_,_,t)=w in t
end (* module World *)

module Steps   ...
type output
type externalEvent
val function record_outputs (n:node) (outs:list output) : list externalEvent
predicate commitp (state) (state)
function commitf (state) (state) : list output
predicate consistent (t:list externalEvent)

val ghost predicate indpred (w:world)
  ensures { ...  /\ result -> consistent (trace w) }

function step_message (w:world) (p:packet) (r:(state, list packet, list output)) : world =
  let (st, ms, outs) = r in let localState = set (localState w) (dest p) st in
      let inFlightMsgs = ms ++ (remove p (inFlightMsgs w)) in
        let trace = (record_outputs (dest p) outs) ++ (trace w) in
          (localState, inFlightMsgs, trace)

val function handleMsg (h:node)(s:node)(m:msg)(sig:state) : (state, list packet, list output)
  requires { ... }
  ensures  { ...  /\ let (s',_,lo) = result in (commitp s s' ->
                       lo = commitf s s') /\ (not (commitp s s') -> lo = Nil) }

inductive step world node world =
| step_msg : forall w :world, p :packet.
    mem p (inFlightMsgs w) -> step w (dest p) (step_message w p
     (handleMsg (dest p) (src p) (payload p) (localState w (dest p))))
...
lemma commit_step :
  forall w w' :world, n :node. reachable w -> step w n w' ->
     (commitp (localState w n) (localState w' n)  ->
       trace w' = (record_outputs n (commitf (localState w n) (localState w' n))) ++ trace w)
  /\ (not (commitp (localState w n) (localState w' n))  ->  trace w' = trace w)

lemma consistent_reachable :
  forall w :world. reachable w -> consistent (trace w)
end (* module Steps *)
```

Listing 5.1. Message-passing model: `modelMPTrace`

3. in every reachable world an output n is stored in position n of the trace.

The system's trace stores the sequence of outputs sent by different nodes. Together, these requirements mean that a node acquiring the lock at epoch n writes to position n of the trace, which implies (since traces are only modified by appending at the head) that no two nodes acquire the lock in the same epoch.

Specifications are written as Why3-do modules defining the `output` and `externalEvent` types, together with projection and the `record_outputs` functions. Most importantly, they define the `commitp` and `consistent` predicates, as well as the `commitf` function. However, the specification is abstract and does not impose the use of any specific system model. It requires the presence of certain types, but does not specify how the types are implemented. The requirement that states should contain specific information is included by declaring functions

```
module Spec
  (* to be instantiated when cloning this module *)
  type node
  type state
  function getEpochS (s:state) : int
  predicate getHeldS (s:state)

  type output = | Locked int
  function getEpoch0 (o:output) : int =
    match o with | Locked e -> e end
  type externalEvent = (node, output)
  function node (e:externalEvent) : node = let (n,_) = e in n
  function outp (e:externalEvent) : output = let (_,o) = e in o
  let rec function record_outputs (n:node) (outs:list output) : list externalEvent
    ensures { forall i :int. 0<=i<length outs -> nth i result = (n, nth i outs) }
  = ...
  predicate commitp (s:state) (s':state) = not (getHeldS s) /\ getHeldS s'
  function commitf (_:state) (s':state) : list output = Cons (Locked (getEpochS s')) Nil
  predicate consistent (t:list externalEvent) =
    match t with
    | Nil -> true
    | Cons (_,o) tt -> getEpoch0 o = length t /\ consistent tt
    end
end (* module Spec *)
```

Listing 5.2. Specification module for distributed lock

and/or predicates on states. Implementation modules will define these types and functions and clone the specification module, instantiating them.

This specification of the distributed lock is written as the Why3-do module of Listing 5.2. It assumes the use of a system model defining types `node`, `state`, `output`, and `externalEvent`. The above requirements are formalized as follows:

1. the functions `getEpochS` and `getHeldS` express required state information;
2. the `output` type has a single constructor carrying an integer; `externalEvent`s are outputs paired with nodes; the `commitp` predicate states that outputs are produced when the state of a node changes from not holding to holding a lock, and the `commitf` function returns a list with the node's current epoch;
3. the `consistent` predicate uses the list `length` function to require that the output stored in each position n of the trace contains epoch n.

We will consider two message-passing implementations for this specification based on a ring topology, shown in listings 5.3 and 5.4. Node states are records with two fields: a Boolean `held` indicating whether the node holds the lock, and its current `epoch`. After the appropriate type definitions, both implementation modules clone the same `Spec` module, and then the `World` module from the appropriate model. The idealized model `modelMPEnabledTrace` is used in the implementation of Listing 5.3, whereas Listing 5.4 uses `modelMPEnabledTraceDupl` in which messages can be duplicated. Both are extensions of `modelMPTrace` (Listing 5.1) with an *enabling predicate*. Enabling predicates allow for nodes to execute guarded actions: when cloning the model, the `enabled` predicate (with a node and its state as parameters) and the `handleEnbld` function are instantiated; the semantics states that the handler may be executed whenever the predicate

```
type node = int
val constant n_nodes : int
axiom n_nodes_ax : 2 <= n_nodes
let function next (x:node) : node = mod (x+1) n_nodes

type state = { held : bool; epoch : int }
function getEpochS (s:state) : int = epoch s
predicate getHeldS (s:state) = held s

type msg = int
predicate ok_Msg (dest:node) (src:node) (_:msg) =
  0<=dest<n_nodes /\ 0<=src<n_nodes /\ dest = next src

clone specLDT.Spec with type node, type state, function getEpochS, predicate getHeldS

clone modelMPEnabledTrace.World with type node, type state,
  type msg, type output, type externalEvent

let function initState (n:node) : state
= let h = if n=0 then true else false in
    let e = if n=0 then 1 else 0 in
      { held = h; epoch = e }
let constant initMsgs : list packet = Nil
let constant initTrace : list externalEvent = Cons (0,Locked(1)) Nil

let function handleMsg (_:node)(_:node) (m:msg) (s:state) :(state, list packet, list output)
= if (not (held s) ) then ({ held = True; epoch = m }, Nil, Cons (Locked m) Nil)
  else (s, Nil, Nil)

let ghost predicate enabled (s:state) (i:node)
= 0<=i<n_nodes && held s

let function handleEnbld (h:node) (s:state) : (state, list packet, list output)
= let e = epoch s in ({ held = False; epoch = e }, Cons (next h, h, e+1) Nil, Nil)

let rec ghost predicate zeroHeld (lS:map node state) (n:int) = ...
let rec ghost predicate oneHeld (lS:map node state) (n:int) = ...
let rec ghost predicate oneMsg (lp:list packet) = length lp = 1
let rec ghost predicate noMsgs (lp:list packet) = length lp = 0

let rec ghost predicate ok_trace (t:list externalEvent)
  ensures { result -> consistent t }
= match t with
  | Nil -> true
  | Cons (_,o) Nil -> getEpoch0 o = 1
  | Cons (_,o1) os ->
    match os with
    | Nil -> true
    | Cons (_,o2) _ -> getEpoch0 o1=(getEpoch0 o2)+1 && ok_trace os
    end
  end

predicate inv (lS:map node state) (iFM:list packet)
  (tr:list externalEvent)
= (forall p: packet. mem p iFM -> ok_Msg(dest p)(src p)(payload p))
  /\ ((oneMsg iFM /\ zeroHeld lS n_nodes)
      \/ (noMsgs iFM /\ oneHeld lS n_nodes))
  /\ (forall n :node. 0<=n<n_nodes -> held (lS n) ->
       n = node (hd tr) /\ epoch (lS n) = getEpoch0(outp (hd tr)))
  /\ (forall p: packet. mem p iFM ->
       src p = node (hd tr) /\ payload p=getEpoch0(outp (hd tr))+1)
  /\ length tr > 0 /\ ok_trace tr

let ghost predicate indpred (w:world)
= inv (localState w) (inFlightMsgs w) (trace w)

clone modelMPEnabledTrace.Steps with ...
```

Listing 5.3. Distributed lock with idealized model

```
...
let function handleMsg (_:node) (_:node) (m:msg) (s:state)
  : (s':state, lp:list packet, lo:list output)
= let nop = (s, Nil, Nil) in
     if (held s) ||  m <= epoch s then nop
    else ({ held = True; epoch = m }, Nil, Cons (Locked m) Nil)
...
(* helper definitions for invariant predicate *)
let rec ghost predicate zeroHeld (lS:map node state)(n:int) ...
let rec ghost predicate atMostOneHeld (lS:map node state)(n:int)...
let rec ghost predicate isFresh (p: packet) (lS:map node state)...
let rec ghost predicate allStale (lS:) (lp:list packet)...
let rec ghost predicate atMostOneFresh (lS:...)(lp:...)...
let rec ghost predicate ok_trace (t:list externalEvent)...

predicate inv (lS:map node state) (iFM:list packet)
  (tr:list externalEvent)
= (forall p: packet. mem p iFM -> ok_Msg (dest p)(src p)(payload p))
  /\ atMostOneFresh lS iFM /\ atMostOneHeld lS n_nodes
  /\ (zeroHeld lS n_nodes \/ allStale lS iFM)
  /\ (forall n :node. 0<=n<n_nodes -> held (lS n) ->
        n = node (hd tr) /\ epoch (lS n) = getEpoch0(outp (hd tr)))
  /\ (forall p: packet. mem p iFM -> isFresh p lS ->
    src p = node (hd tr) /\ payload p = getEpoch0(outp (hd tr))+1)
  /\ length tr > 0 /\ ok_trace tr
...
```

Listing 5.4. Distributed lock with duplicating messages model

is true. In the present example, `enabled` is defined as true when a node holds a lock, in which case it is free to release it. The lock is released when `handleEnbld` executes, sending a message to the next node in the ring. The message includes the value of the sender's current epoch, incremented by one.

The system is initialized with node 0 holding the lock (and this fact is registered in the system trace). The handling functions then follow. The enabling predicate and the corresponding handler are the same in both implementations; it is in the message handlers that they differ. With the idealized model nodes can trust that messages are never stale, so they react by blindly acquiring the lock. With the duplicating model the receiving node first checks whether the epoch in the received message is higher than its present epoch (in which case it cannot be a stale copy of a previous message). The inductive invariants are also different for both implementations, but both include a property expressed with the `ok_trace` predicate, stating that events in the trace contain incremental epochs, starting from 1. This implies consistency of the trace (as defined in the specification), and is easier to check for inductiveness.

Let us consider in detail the system of Listing 5.4. A message is fresh if the current epoch of its destination node is lower than the message. Transfer messages are always sent from the highest epoch node (holding the lock) and thus, at the time of sending, the destination has a lower epoch, which will be updated when the message is received and the lock acquired. Other copies of the message are stale because their destinations' epochs have since increased. The system's invariant is given as the conjunction of the following properties, using the `zeroHeld`, `atMostOneHeld`, `allStale`, and `atMostOneFresh` predicates: (i)

in-transit messages are well-formed; (ii) there is at most one in-transit fresh message, and at most one node holding a lock; if a node holds a lock then all in-transit messages are stale; (iii) If node n holds the lock then the last `Locked x` was written in the trace by n, and x is the current epoch of n; (iv) if there exists a fresh in-transit message, then it was sent by the last node that output `Locked x`, and it carries the value $x + 1$; (v) the trace obeys the `ok_trace` predicate.

The VCs generated for the modules of listings 5.3 and 5.4, proved automatically, establish the correctness of each system with respect to the specification of Listing 5.2: events are being logged in the specified way, and traces are consistent.

6 Locally Shared Memory Model

Dijkstra described certain distributed systems (including the self-stabilizing systems described below) using a guarded processes model, in which nodes/processes do not exchange messages, but instead have direct read access to each other's states. Although particular systems will only require read access to a limited set of states (typically its immediate neighbors'), our model allows read access universally. This is not a shared-memory model in all generality, but it may be implemented over shared memory, with a single-writer multiple-reader data structure for each node's state (and readers–writer locks for atomicity).

We formalize this in our setting as a model where worlds are simply of the form $\langle \mathsf{IS} \rangle$ with $\mathsf{IS} : \mathbf{N} \to \Sigma$ a state-assigning function. A system based on this model is programmed by defining an enabling predicate on nodes and a handling function describing the behavior that can be executed whenever a node is enabled. Formally we will consider that the enabling predicate has signature $\mathsf{ep}(n : \mathbf{N}, \mathsf{IS} : \mathbf{N} \to \Sigma)$, taking as parameters a node and a global state assigning function, and the handling function has the following signature and contract:

$$\begin{array}{l} \mathsf{handleE}(h : \mathbf{N}, \mathsf{IS} : \mathbf{N} \to \Sigma) : (\sigma : \Sigma) \\ \textbf{requires } \mathsf{ep}(h, \mathsf{IS}) \wedge I\langle \mathsf{IS} \rangle \\ \textbf{ensures } I\langle \mathsf{IS}[h \mapsto \sigma] \} \rangle \end{array}$$

The enabling predicate and the handler code have read access to every node's state, but the handler may only modify the state of the node where it is running. This semantics is given by the following rule:

$$\frac{\mathsf{handleE}(h, \mathsf{IS}) = \sigma \qquad \mathsf{ep}(h, \mathsf{IS})}{\langle \mathsf{IS} \rangle \rightsquigarrow_h \langle \mathsf{IS}[h \mapsto \sigma] \rangle} \ (enabled)$$

where $\rightsquigarrow_h$ means that node h runs the handler. The contract of $\mathsf{handleE}$ ensures that executions of the *(enabled)* transition rule preserve the property I (the contract ensures this if the node is enabled, and the semantics only allow for transitions satisfying this requirement). We will write $ok^I(\mathsf{ep}, \mathsf{handleE})$ when the implementation of the handling function $\mathsf{handleE}$ adheres to its contract, with invariant I and enabling predicate ep. Listing 6.1 shows a simplified version of the Why3-do `modelReadallEnabled` module, including the following Lemma, proved using an induction transformation and SMT solvers.

```
module World
  type node, type state, type world =  map node state
end

module Steps
  val predicate validNd (n:node)
  val function initState (node) : state
  constant initWorld : world = initState

  val ghost predicate indpred (w:world)
    ensures { w=initWorld -> result }
  val ghost predicate enabled (map node state) (i:node)
    requires { validNd i }

  function step_enbld (w:world) (n:node) (st:state) : world = set w n st

  val function handleEnbld (h:node) (lS:map node state) : state
    requires { validNd h /\ enabled lS h /\ indpred lS }
    ensures  { indpred (step_enbld lS h result) }

  inductive step world node world =
  | step_enbld : forall w :world, n :node. validNd n -> enabled w n ->
                   step w n (step_enbld w n (handleEnbld n w))

  lemma indpred_step :
    forall w w' :world, n :node. step w n w' -> indpred w -> indpred w'
  lemma step_preserves_states :
    forall w w' :world, n i :node. step w n w' -> i<>n -> w i = w' i

  (* keeps track of number of transition steps *)
  inductive step_TR world world int =
  | base : forall w :world. step_TR w w 0
  | step : forall w w' w'' :world, n :node, steps :int.
            step_TR w w' steps -> step w' n w'' -> step_TR w w'' (steps+1)

  lemma noNeg_step_TR : forall w w' :world, steps :int. step_TR w w' steps -> steps >= 0
  lemma indpred_manySteps :
    forall w w' :world, steps :int . step_TR w w' steps -> indpred w -> indpred w'

  predicate reachable (w:world) = exists steps :int. step_TR initWorld w steps
  lemma indpred_reachable : forall w :world. reachable w -> indpred w
end
```

Listing 6.1. Locally shared memory model: `modelReadallEnabled`

Lemma 3. *Let* $w_0, w \in \mathbf{W}$*, with* ep *and* I *predicates such that* $ok^I(\mathsf{ep}, \mathsf{handleE})$*,* $w_0 \models I$*, and* $w_0 \rightsquigarrow^* w$*. Then* $w \models I$*.*

Example: Stabilizing Mutual Exclusion. Self-stabilizing systems [15,38] are designed to tolerate failures resulting from "horrible errors" (such as data corruption), by including a recovery mechanism. Given some notion of *legal configuration*, a system is said to be *self-stabilizing* if (i) starting from an illegal configuration, all executions eventually *converge* to a legal configuration (a liveness property), and (ii) legal configurations are *closed* under normal execution steps, i.e. no illegal configuration is reachable if no corruption of data occurs (a safety property). One of Dijkstra's examples of such a system in his seminal paper [15] was a directed ring of processes sharing a resource, with mutual exclusion enforced by means of a circulating token. Legal configurations are those in

```
module SelfStab_Ring_Closure
  type node = int
  val constant n_nodes : int
  axiom n_nodes_bounds : 2 < n_nodes
  let predicate validNd (n:node) = 0 <= n < n_nodes
  type state = int
  val constant k_states : int      axiom k_states_lower_bound : n_nodes < k_states
  let function incre (x:state) : state = mod (x+1) k_states

  clone modelReadallEnabled.World with type node, type state

  let function initState (n:node) : state = if n=n_nodes-1 then 1 else 0

  predicate has_token (lS:map node state) (i:node) =
    (i = 0 /\  lS i =  lS (n_nodes-1)) \/ (i > 0 /\ i < n_nodes /\ lS i <> lS (i-1))
  let ghost predicate enabled (lS:map node state) (i:node) = has_token lS i

  let function handleEnbld (h:node) (lS:map node state) : state
  = if h = 0 then incre (lS (n_nodes-1)) else lS (h-1)

  let rec ghost predicate atLeastOneToken (lS:map node state) (n:int)
    requires { validNd n }
    ensures { result <-> exists k :int. 0<=k<n /\ has_token lS k }
    variant { n }
  = n > 0 && (has_token lS (n-1) || atLeastOneToken lS (n-1))

  predicate atMostOneToken (lS:map node state) (n:int) = validNd n ->
    forall i j :int. 0<=i<n -> 0<=j<n -> has_token lS i -> has_token lS j -> i=j

  lemma first_last : forall n: int, lS :map node state.
                       n >= 0 -> (forall j :int. 0<j<=n -> lS j = lS (j-1)) -> lS 0 = lS n
  lemma atLeastOneTokenLm : forall w :world. atLeastOneToken w n_nodes

  predicate inv (lS:map node state) =
    (forall n :int. validNd n -> 0 <= lS n < k_states) /\ atMostOneToken lS n_nodes
  let ghost predicate indpred (w:world) = inv w

  clone modelReadallEnabled.Steps with type node, type state,
    val validNd, val initState, val indpred, val enabled, val handleEnbld

  predicate oneToken (w:world) = atMostOneToken w n_nodes /\ atLeastOneToken w n_nodes
  goal oneToken : forall w :world. reachable w -> oneToken w
end
```

Listing 6.2. Self-stabilizing mutual exclusion on a ring – Closure

which exactly one process carries a token. In case of failure the system converges back into a single-token configuration. Dijkstra's proposal for self-stabilizing mutual exclusion was the following: processes have integer numbers in $\{0, \ldots K-1\}$ as states, with K greater than the size of the ring. Each process observes the state of its predecessor in the ring; the process with index 0 holds a token when its state is *the same* as that of its predecessor (the last process in the ring); other processes hold a token when their state is *different* from their predecessor's. When holding a token, each process may modify its state by copying its predecessor's state; node 0 additionally increments (modulo K) this state.

Listing 6.2 shows the Why3-do formalization of this system, based on the locally shared memory model. Nodes and states are both integers; `n_nodes` and `k_states` are the size of the ring and the number of different states. The en-

abling predicate is defined as true for a node exactly when it is carrying a token, as specified by the `has_token` predicate. The handler defined by `handleEnbld` copies states as previously described. Mutual exclusion is expressed using predicates `atLeastOneToken` and `atMostOneToken` that apply to the first `n` nodes.

The module of Listing 6.2 verifies the closure property. The invariant expresses that node states are within bounds, and there is no more than one token in the ring. One possible (legal) initial configuration of the system is described by the `initState` let function. These definitions are instantiated when cloning `modelReadallEnabled`. The module ends with the `oneToken` goal, stating that there exists exactly one token in all reachable configurations.

Stepwise Bounded Validation. In the verification of closure we use the following technique: we introduce an axiom bounding the size of the system, passed to the solvers to make automated proofs easier (soundness of the verification may be compromised at this point). We then introduce parts of the invariant step by step, and check them in this bounded system in order to gain insight as to their validity. Once we feel confident about the elected invariant, we remove the bounding axiom to achieve soundness of the verification, possibly stating additional lemmas or strengthening the invariant. For the present system:

1. We started with the following invariant. Inductiveness is proved automatically, but the `oneToken` goal cannot be proved from it (as expected):
```
forall i :int. validNd i -> 0 <= lS i < k_states.
```
2. Next, we included `atMostOnetoken lS n_nodes` in the invariant; preservation was proved automatically, but `oneToken` could still not be proved. We then added a bounding axiom `n_nodes <= 10`, which allowed the goal to be proved.
3. We strengthened the invariant with `atLeastOnetoken lS n_nodes` and removed the bounding axiom. The `oneToken` goal was proved trivially; however, the VC pertaining to the preservation of the invariant could not be proved.
4. Preservation could be proved by reintroducing a bound on `n_nodes` (with a bound of 1000, all VCs could be proved within 30 seconds in our setup).

These bounded proof results indicate that, in all likelihood, (i) the property `atLeastOnetoken lS n_nodes` is preserved by system transitions, and thus inductive, but (ii) it is not necessary to include it in the inductive invariant to prove `oneToken`: in our development the `oneToken` goal could be proved for a number of processes up to 10 without including the former property in the invariant. The reason for this is that in fact the `atLeastOnetoken lS n_nodes` property is satisfied by definition in all configurations: in order for a token to be present, either any two adjacent processes have different states, or the first and last processes have the same state. If all processes have the same state, then the second case holds. Including the property in the invariant still requires a bound (to prove preservation), but this can now have a much higher value (1000 rather than 10).

An unbounded proof is obtained by including in the module the `first_last` lemma (proved by induction on n). This allows for the goal to be proved automatically without `atLeastOnetoken lS n_nodes` in the invariant, and with no upper bound on `n_nodes`. We remark that the dual definition (recursive +

	TLAPS	Verdi	IronFleet	Ivy	Why3-do
Contract-based design			✓(partial)		✓
DS models	generic	MP	MP	MP	MP; LSM
Reusable Models		✓	✓		✓
Different fault models		✓			✓
Verified system transforms		✓			
Abstract Specifications	state machines; spec to protocol refinement	observ. traces	state machines; spec to protocol refinement		observ. traces (model-independent)
Liveness properties	✓(TLC)		✓(TLC)		
Logic	TLA+	FOL	FOL	EPR	FOL
Invar. discovery support				✓	
Automated provers	multiple		Z3	Z3	multiple
Proof assistants	multiple	Coq			multiple
Programming language	PlusCal	Gallina (F)	state machines; Dafny (F/I)	RML	WhyML (F/I)
Implementation support			UDP model/ machine types		mutable/machine (WhyML) types
Generation of executables		✓	✓		

Table 7.1. Comparison of DS deductive verification frameworks
MP: message-passing, LSM: locally shared memory, F: functional, I: imperative

contract) of the `atLeastOneToken` let function was crucial for proving the goal automatically (this was not possible with a logic definition).

The convergence property is more challenging; its Why3-do formalization can be found in the artifact [28]. We have also verified Dijkstra's version of this system with a bidirectional array topology. Bounded exploration again allowed us to validate parts of the invariant; attaining an unbounded verification required strengthening the invariant, rather than a lemma.

7 Related Work

Deductive verification methods are typically based on first-order logic reasoning and focus on safety properties, with correctness proofs requiring users to manually provide appropriate *invariants* and to discharge (either automatically or interactively) proof obligations generated in the process. Invariants may apply to loops, recursive functions, or non-deterministic transition relations, and allow for correctness proofs by induction on the length of executions. In the last few years a number of frameworks and tools have been proposed for reasoning about asynchronous message-passing systems using inductive invariants, based on atomic handler models and different specification mechanisms. We will now briefly survey these and compare them with Why3-do in terms of design choices.

Verdi [42] introduced the use of models based on worlds and atomic handlers, with models capturing different fault semantics. Why3-do's semantic framework is inspired by Verdi; we enrich handlers with interface specifications in the form of *contracts*, allowing for the use of methods that are standard in deductive verification of single-thread software. Verdi is a Coq development, and reasoning is carried out within the Coq proof assistant [22]. The implementation of our

framework as a Why3 library allows for the use of automated tools (all the proofs in this paper use SMT solvers and a few Why3 transformations).

Whereas Verdi handlers are defined in a purely functional style, in Why3-do they are written in WhyML, combining functional and imperative features. Verdi supports system transformations that allow for verified systems to be obtained from systems verified with simpler models (additional mechanisms may be automatically introduced to compensate for the presence of faults). Transformations are verified once and for all, so the resulting systems do not need to be verified. An important difference is that Verdi targets exclusively message-passing systems, whereas Why3-do covers different system models. Verdi supports traces, but specifications may not be written in a completely abstract, model-independent way. In Why3-do this is achieved through the use of clonable specification modules defining commit specifications and trace consistency.

The IronFleet [20] platform is built on top of a deductive verification tool, Dafny [26], which uses the Z3 [31] SMT solver for proofs. Like Verdi, it supports only message-passing systems. A major difference with respect to Why3-do and Verdi is that, instead of a specification mechanism based on traces, IronFleet separates development in a specification level (where worlds are viewed abstractly) and a concrete protocol level, both described in FOL as state machines. A refinement function [1] maps protocol worlds to the specification level, and a refinement proof shows that protocol steps are compatible with the abstract behavior (in Why3-do this is achieved by trace consistency proofs). There is a third, implementation level, where event handlers are programmed using mutable data structures and machine types, for performance and realism. IronFleet extends Dafny with a UDP specification to support networking, which allows non-atomic handlers to be developed assuming low-level interleaving. In order to establish refinement proofs between low-level implementations and protocols, reduction-based reasoning is supported. IronFleet also includes an embedding of TLA that makes possible reasoning about liveness properties. It is overall an ambitious tool that has been used by its authors to verify practical systems.

Up to a point Why3-do implementations cover both the protocol and implementation levels, since WhyML accommodates both functional programs and stateful code with mutable structures and machine types. Why3 supports code extraction from verified WhyML programs, and it should not be difficult to obtain a distributed implementation from a verified Why3-do system, using one of the available OCaml libraries. Our framework allows for diverse system models, with different implementation infrastructure requirements. In general each node must run a scheduler that will, for instance, receive incoming local inputs and messages from the network, check enabling predicates, and run the appropriate handlers, reflecting locally and globally the effects prescribed by the semantics.

The Ivy tool [34] differs from Why3-do and the previous frameworks in several important ways. It uses a dedicated modeling/programming language called RML, and a logic language restricted to the effectively propositional (EPR) class of formulas, whose satisfiability is decidable (Ivy also uses Z3). Specifications may refer to any part of the model (no specification/protocol distinct layers or

observation traces are used). The use of EPR imposes severe restrictions: RML does not allow arithmetic operations, so for instance a ring topology cannot be modeled using integer modulo arithmetic. A verification methodology based on the use of EPR, and details on how it has been used to verify variants of the PAXOS protocol, are extensively described in [33] (the method proposed for reducing quantifier alternation is of general interest, even when unrestricted FOL is used). Leveraging the decidability of the logic, Ivy focuses on assisting the user in writing the protocol and its specification, and in discovering adequate inductive invariants. A few initial steps of execution are first considered, which may allow for bugs to be found in the protocol and/or target properties; Ivy then assists the user in finding an inductive invariant by performing interactive strengthening and generalization steps, and representing states visually.

A more general, comprehensive framework for reasoning about distributed systems has been constructed around the TLA+ specification language, based on the Temporal Logic of Actions [25]. TLA+ is without any doubt a widely successful toolset, and its adoption in practice is well documented [32]. The toolset comprises the specification language itself; the PlusCal algorithmic language; the TLC model checker [43]; the TLAPS proof system [8]; and a development environment. Correctness proofs are based on the notion of refinement mapping [1]. If one writes a TLA+ specification and a PlusCal implementation, and then translates the latter to TLA+, its correctness can be stated as a refinement problem, whose VC is itself written as a TLA+ formula. The TLAPS proof system is an ongoing effort but can already be used to prove many such refinements. TLAPS proofs [12] are constructed using both proof assistants and SMT solvers.

Table 7.1 summarizes the distinctive aspects of the discussed tools. Additionally, the I4 technique has been proposed [29] based on the automatic synthesis (by model checking) of inductive invariants for small instances of protocols, followed by their generalization. Invariants are checked with Ivy, and if necessary the process is repeated, considering a bigger instance or a pruned invariant. Kaizen [23] is a verified blockchain system that has been developed using an approach similar to IronFleet. Implementations of distributed systems that have been formally verified using different tools have been empirically scrutinized in [19].

Program logics for distributed systems have also been the subject of recent work, typically based on or inspired by concurrent separation logics [6], and mechanized in the Coq proof assistant. Notable examples include Disel [39], which focuses on modularity and compositionality, and Aneris [24], which includes support for node-level concurrency in addition to inter-node reasoning. ModP [14] is an actor-based compositional programming framework that offers assume-guarantee reasoning principles to support compositional system testing.

The self-stabilizing ring system has been verified interactively using the PVS [35] and Isabelle [30] proof assistants, and also by symbolic model checking [41,9]. A general framework for building certified proofs of self-stabilizing algorithms (using Coq) is described in [3].

8 Conclusion

In this paper we have proposed principles for contract-based verification of distributed systems, based on a library promoting modular development. The approach enables the use of state of the art sequential software verifiers for reasoning about distributed systems, supports model-independent trace specifications, and is uniform across system models, beyond the message-passing setting.

To implement these principles we have chosen the Why3 verification platform. We have shown how specific features of Why3, such as the ability to interface with different solvers and the use of dual definitions, contribute to successful automated proofs. For instance, we were able to prove the inductiveness of an invariant for the leader election protocol containing a quantifier 'alternation' (a sequence of the form $\forall\exists$ [33], outside the decidable EPR logic). In particular, the Alt-Ergo and Vampire solvers were able to prove these VCs, whereas Z3 and CVC4 failed (with a generous timeout value). On the other hand, the dual definition of the `atLeastOneToken` predicate in the self-stabilization systems, when the invariant included this predicate containing an existential quantifier, allowed Z3 or CVC4 (not the other solvers) to prove inductiveness. In neither case was it necessary to employ invariant quantifier hiding, as in [20].

Unbounded domains (nodes, messages, etc.) are typical of distributed systems. Considering bounded systems, in combination with dual definitions, allowed us to explore the inductiveness of invariant properties before tackling the unbounded case (by strengthening invariants or writing lemmas). This should not be mistaken with the use of bounded verification in Ivy, which considers the first few system steps in order to debug models, or in I4, which produces finite quantifier-free instances of problems, amenable to model checking.

The limitations of the framework are that, in the spirit of verification of sequential programs with Why3, Why3-do targets the verification of distributed systems at the *algorithmic level*, and is not intended for reasoning about executable implementations (but see the discussion on implementation extraction in Section 7). Also, no support for reasoning with non-atomic handlers is included.

Why3 is a stable tool, actively developed by a solid team, with a growing user community and very low risk of obsolescence. It is being successfully used for formal verification in contexts as diverse as safety-critical programming [2], multicore schedulers [27], or blockchain smart contracts [37,40]. Why3-do brings Why3's strengths in terms of usability and proof engineering to the mechanical verification of distributed systems, making it available to a wider community.

Acknowledgments. The development of Why3-do was initiated during a visit of the second author to the Toccata team at Inria Saclay-Île-de-France/LRI Univ Paris-Saclay/CNRS and greatly benefited from the team's hospitality and Why3 expertise. This work is financed by the ERDF – European Regional Development Fund through the North Portugal Regional Operational Programme - NORTE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project NORTE-01-0145-FEDER-028550 - PTDC/EEI-COM/28550/2017.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science **82**(2), 253–284 (1991). https://doi.org/10.1016/0304-3975(91)90224-P
2. AdaCore and Altran UK Ltd: SPARK 2014 Reference Manual – Release 2020 (2020)
3. Altisen, K., Corbineau, P., Devismes, S.: A framework for certified self-stabilization. In: Albert, E., Lanese, I. (eds.) Formal Techniques for Distributed Objects, Components, and Systems. pp. 36–51. Springer International Publishing, Cham (2016)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
5. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 171–177. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
6. Brookes, S., O'Hearn, P.W.: Concurrent separation logic. ACM SIGLOG News **3**(3), 47–65 (Aug 2016). https://doi.org/10.1145/2984450.2984457
7. Chang, E., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Commun. ACM **22**(5), 281–283 (May 1979). https://doi.org/10.1145/359104.359108
8. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the TLA + proof system. In: Giesl, J., Hähnle, R. (eds.) Automated Reasoning. pp. 142–148. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
9. Chen, J., Abujarad, F., Kulkarni, S.: Towards scalable model checking of self-stabilizing programs. Journal of Parallel and Distributed Computing **73**(4), 400–410 (2013). https://doi.org/10.1016/j.jpdc.2012.12.009
10. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
11. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories. Oxford, United Kingdom (Jul 2018)
12. Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., Vanzetto, H.: TLA + proofs. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods. pp. 147–154. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
13. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - A software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
14. Desai, A., Phanishayee, A., Qadeer, S., Seshia, S.A.: Compositional programming and testing of dynamic distributed systems. Proc. ACM Program. Lang. **2**(OOPSLA) (oct 2018). https://doi.org/10.1145/3276529
15. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (Nov 1974). https://doi.org/10.1145/361179.361202
16. Dijkstra, E.W., Scholten, C.S.: Predicate calculus and program semantics. Springer-Verlag New York, Inc., New York, NY, USA (1990)
17. Filliâtre, J.: One logic to use them all. In: Bonacina, M.P. (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. Lecture Notes in Computer

Science, vol. 7898, pp. 1–20. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_1
18. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)
19. Fonseca, P., Zhang, K., Wang, X., Krishnamurthy, A.: An empirical study on the correctness of formally verified distributed systems. In: Proceedings of the Twelfth European Conference on Computer Systems. p. 328–343. EuroSys'17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3064176.3064183
20. 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
21. Hoare, C.A.R.: An Axiomatic Basis For Computer Programming. Communications of the ACM **12**, 576–580 (1969)
22. Huet, G., Kahn, G., Paulin-Mohring, C.: The Coq proof assistant : A tutorial : Version 6.1. Tech. rep., INRIA (07 1997)
23. Kalim, F., Palmskog, K., Mehar, J., Murali, A., Gupta, I., Madhusudan, P.: Kaizen: Building a performant blockchain system verified for consensus and integrity. In: 2019 Formal Methods in Computer Aided Design (FMCAD). pp. 96–104 (2019). https://doi.org/10.23919/FMCAD.2019.8894248
24. Krogh-Jespersen, M., Timany, A., Ohlenbusch, M.E., Gregersen, S.O., Birkedal, L.: Aneris: A mechanised logic for modular reasoning about distributed systems. In: Müller, P. (ed.) Programming Languages and Systems. pp. 336–365. Springer International Publishing, Cham (2020)
25. Lamport, L.: The temporal logic of actions. Tech. Rep. 79, Digital Equipment Corporation (May 1994), aCM Transactions on Programming Languages and Systems 16
26. Leino, R.: Dafny: An automatic program verifier for functional correctness. In: 16th International Conference, LPAR-16, Dakar, Senegal. pp. 348–370. Springer Berlin Heidelberg (April 2010)
27. Lepers, B., Gouicem, R., Carver, D., Lozi, J.P., Palix, N., Aponte, M.V., Zwaenepoel, W., Sopena, J., Lawall, J., Muller, G.: Provable multicore schedulers with ipanema: Application to work conservation. In: Proceedings of the Fifteenth European Conference on Computer Systems. EuroSys'20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3342195.3387544
28. Lourenço, C.B., Pinto, J.S.: Why3-do: The way of harmonious distributed system proofs. ESOP 2022 Artifact (2022). https://doi.org/10.5281/zenodo.5914171
29. Ma, H., Goel, A., Jeannin, J.B., Kapritsos, M., Kasikci, B., Sakallah, K.A.: I4: Incremental inference of inductive invariants for verification of distributed protocols. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. p. 370–384. SOSP '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3341301.3359651
30. Merz, S.: On the verification of a self-stabilizing algorithm. Tech. rep., University of Munich (1998)
31. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver, Lecture Notes in Computer Science, vol. 4963/2008, pp. 337–340. Springer Berlin (April 2008)

32. Newcombe, C.: Why amazon chose TLA+. In: Ait Ameur, Y., Schewe, K.D. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 25–39. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
33. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: Decidable reasoning about distributed protocols. Proc. ACM Program. Lang. **1**(OOPSLA) (Oct 2017). https://doi.org/10.1145/3140568
34. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: Safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 614–630. PLDI '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908118
35. Qadeer, S., Shankar, N.: Verifying a self-stabilizing mutual exclusion algorithm. In: Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods. pp. 424–443. PROCOMET '98, Chapman & Hall, Ltd. (1998)
36. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Commun. **15**(2-3), 91–110 (2002)
37. Rognier, B.: Verify a smart contract with archetype. `https://medium.com/coinmonks/verify-a-smart-contract-with-archetype-6e0ea548e2da` (2019)
38. Schneider, M.: Self-stabilization. ACM Comput. Surv. **25**(1), 45–67 (Mar 1993). https://doi.org/10.1145/151254.151256
39. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158116
40. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ArXiv **abs/2008.02712** (2020)
41. Tsuchiya, T., ichi Nagano, S., Paidi, R.B., Kikuno, T.: Symbolic model checking for self-stabilizing algorithms. IEEE Trans. Parallel Distrib. Syst. **12**(1), 81–95 (2001)
42. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: A framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 357–368. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737958
43. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Pierre, L., Kropf, T. (eds.) Correct Hardware Design and Verification Methods. pp. 54–66. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)