Universidade do Minho
Escola de Engenharia
Departamento de Informática

Carla Isabel Novais da Cruz

Verification of Distributed Algorithms with the Why3 tool

December 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Carla Isabel Novais da Cruz

Verification of Distributed Algorithms with the Why3 tool

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Professor Jorge Miguel de Matos Sousa Pinto
Professor Georg Weissenbacher

December 2021

ACKNOWLEDGEMENTS

I want to thank Ph.D. Jorge Miguel de Matos Sousa Pinto for accepting to be my supervisor and helping me to perform this scientific research. I want to thank for all the motivation and commitment throughout the whole project.

I want to thank Ph.D. Georg Weissenbacher for supporting me during my Erasmus period.

I want to thank my family, boyfriend and friends for always believing in me and for supporting every choice I made.

This work is financed by the ERDF – European Regional Development Fund through the North Portugal Regional Operational Programme - NORTE2020 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**.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

Nowadays, there currently exist many working program verification tools however, the developed tools are mostly limited to the verification of sequential code, or else of multi-threaded shared-memory programs.

Due to the importance that distributed systems and protocols play in many systems, they have been targeted by the program verification community since the beginning of this area. In this sense, they recently tried to create tools capable of deductive verification in the distributed setting (deductive verification techniques offer the highest degree of assurance) and claim to have achieved impressive results.

Thus, this dissertation will explore the use of the Why3 deductive verification tool for the verification of distributed algorithms. It will comprise the definition of a dedicated Why3library, together with a representative set of case studies. The goal is to provide evidence that Why3 is a privileged tool for such a task, standing at a sweet spot regarding expressive power and practicality.

KEYWORDS Deductive verification, Distributed systems and protocols, Why3, Why3do

RESUMO

Nos dias de hoje, possuímos diversas ferramentas de verificação, ferramentas essas limitadas à verificação de código sequencial, ou então de programas multi-thread de memória partilhada.

Devido à importância que os sistemas e protocolos distribuídos desempenham em muitos sistemas, estes foram alvos por parte da comunidade de verificação de programas desde o início desta área. Neste sentido, recentemente tentaram criar ferramentas capazes de realizar a verificação dedutiva no ambiente distribuído (técnicas de verificação dedutiva que oferecem o mais elevado grau de segurança) e afirmam ter alcançado resultados impressionantes.

Assim, esta dissertação irá explorar o uso da ferramenta de verificação dedutiva Why3 com o propósito de verificar algoritmos distribuídos. Irão ser desenvolvidos modos e modelos da biblioteca Why3do, juntamente com um conjunto representativo de casos de estudos. O objetivo é fornecer evidências de que Why3 é uma ferramenta privilegiada para esta tarefa, estando no ponto ideal na relação poder expressivo e praticabilidade.

PALAVRAS-CHAVE Verificação dedutiva, sistemas e protocolos distribuídos, Why3, Why3do

CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

1

INTRODUCTION

1.1 MOTIVATION AND PROBLEM STATEMENT

Deductive program verification methods and techniques have been the subject of great attention over the last two decades, due in part to great advances in logic solving tools, such as SAT and SMT solvers [1]. As a result, there currently exist a number of working program verification tools that are capable of checking safety and functional properties of challenging programs written in real-world programming languages like C [2]. However, the developed tools are mostly limited to the verification of sequential code, or else of multi-threaded shared memory programs.

Distributed systems and protocols [3] play an important role in many computer systems, often in scenarios where a bug may have tremendous effects but are famously difficult to implement in a correct way. For this reason, they have been targeted by the program verification community from the inception of the area. However, the same reasons that make these systems difficult to implement also make them difficult to check for property violations, and until very recently no capable tools existed for deductive verification in the distributed setting (deductive verification techniques offer the highest degree of assurance). But this state of things is changing rapidly, with a number of recently introduced tools claiming to have achieved impressive results.

This dissertation will explore the use of the Why3 deductive verification tool for the purpose of verification of distributed algorithms [4]. It will comprise the definition of a dedicated Why3 library, together with a representative set of case studies. The goal is to provide evidence that Why3 is a privileged tool for such a task, standing at a sweet spot regarding expressive power and practicality.

1.2 OBJECTIVES

The main goal and desired result of this dissertation is to contribute towards a framework for the verification of protocols and algorithms.

In order to achieve the main objective, this had to be divided into small parts. So, as an initial task, I started by studying the Why3 tool in detail, and as soon as the necessary knowledge is consolidated, I was able to study and experiment with the library I used, the Why3do library. As a starting point, it only existed a part of message-passing and I studied a specific example that used this model. After this initial part, a very important objective

was to analyze and understand different distributed protocols/algorithms to be able to select some simple case studies.

After this in-depth study of both areas in parallel, it was possible to extend the Why3do library and produce new verified examples. The newly selected examples were the self-stabilization systems. It only existed a draft of a model of locally-shared memory. Using this as a basis, it was possible to develop and improve the model modelReadAllEnable and finally do the prove and the verification of the Dijkstra systems using this improved model - it was made the adaptation for the self-stabilization examples.

A final analysis of the results and problems encountered during the development was carried out, being done a comparison between what was developed in this project and the ones that were used as a start.

1.3 DOCUMENT STRUCTURE

This dissertation report will consist of three main chapters.

The chapter 2 reviews background concepts and in particular contains an introduction to the Why3 tool, extensively used in this dissertation.

In chapter 3, the current state of the art of this project will be explained. It will contain information about the previous work and a specific analysis of the library that I will use, as well as a concrete example that will be one of the bases for the solution to this problem.

In chapter 4, the verification of a self stabilizing algorithm will be made where it is verified the properties of closure and convergence.

In the last chapter 5, it is summarised all the work that was developed and it is also made a review and future work.

2

BACKGROUND

2.1 PROGRAM VERIFICATION

The formal verification of the properties of distributed algorithms and protocols is an undeniably important task, and it is notoriously difficult. The two main methods are essentially algorithmic exploration of the state space (also called model checking) and logical reasoning based on inductive invariants.

2.1.1 *Model Checking*

In computer science, model checking is a method for checking whether a finite-state model of a system meets a given specification (also known as correctness). This is typically associated with hardware or software systems, where the specification contains liveness requirements (such as avoidance of livelock) as well as safety requirements (such as avoidance of states representing a system crash). A simple model-checking problem consists of verifying whether a formula in the propositional logic is satisfied by a given structure [5].

Model checking of distributed systems is a well-developed field that has produced numerous practical results. In particular, the TLA+ framework has had a huge impact on the distributed systems community and is widely recognized as a crucial breakthrough in computer science. It cannot be overstated that distributed systems engineers are using it in practice, and they should - and not just by advocates of formal methods. The current verification efforts at Amazon using this tool are an excellent example of what can be achieved with formal approaches. [6]

Model checking has the distinct advantage of being an automated technique that does not necessitate the discovery of difficult invariants. Moreover, it can be used organically for both safety and liveness. However, due to the phenomenon of state-space explosion phenomenon, it is subject to some limitations. The scale of the system (number of nodes or processes involved) and the length of its executions both contribute to the complexity of the state space, and model checking a system often requires considering constraints in one of these two dimensions (e.g., considering a baby version of a client-server system with only one server and two clients, or considering runs limited to K execution steps). [7]

2.1.2 *Deductive Verification*

Deductive software verification [8], also known as program proving, expresses the correctness of a program as a set of mathematical statements, called verification conditions. They are then discharged using either automated or interactive theorem provers.

Deductive software verification aims at formally verifying that all possible behaviors of a given program satisfy formally defined, possibly complex properties, where the verification process is based on logical inference. As an example of this logic, we have the Hoare logic that is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs [9]. In the deductive verification of programs can be used the loop invariants and some proof tools to achieve the intended goals.

There are some tools that help to do this deductive software verification, like Why3, Dafny and Frama-C. These tools are based on first order logic theories and inductive invariants, and they interface with a great number of first-order proof tools, including all the major SMT solvers and interactive proof assistants [10].

2.2 WHY3 TOOL

This dissertation, it will focus on verification of distributed systems and to achieve this goal, it is necessary to solve the hard task of discovering inductive invariants, using deductive verification. The tool that will be used is Why3 with the Why3do, a Why3 library [4] for reasoning about distributed systems based on first order logic theories and inductive invariants, and interfacing with a great number of first-order proof tools, including all the major SMT solvers and interactive proof assistants.

In the next section, it is possible to understand how this tool works and the properties involved.

2.2.1 *What is the Why3 Tool?*

1. A logical proofing tool
    - Uses as logic language an extension of first-order logic with
        - inductive types and definitions, including inductive predicates
        - polymorphism

      it is a “sweet spot” that allows coding many problems of interest in computer science
    - Interacts with a wide variety of "solvers" and theorem provers, being unique in this respect. It is possible to use several proofing tools
    - Offers a set of proof transformations, which allow, in particular, to make inductive proofs, something that is not possible with an SMT solver. The presence of these transformations can avoid the use of an interactive tool like Coq.
2. A program verification tool

- WhyML internal programming language, mixing functional and imperative characteristics
- Based on the principles of Hoare Logic, on the notions of precondition, postcondition, and loop invariant, and also on the development method known as design-by-contract
- Includes a generator of verification conditions that, given a program annotated with a specification, produce a set of logical formulas (verification conditions) whose validity will imply that the program is correct about its specification.
- The verification conditions are proven using the multiple tools with which Why3 interacts
- After being proved, a WhyML program can be extracted to a real programming language: OCaml or C
- It is possible to combine the two aspects of the tool: the logical language can be used in the reasoning about program properties
- Support for ghost code

2.2.2 *Why the Why3 Tool?*

This dissertation will explore the use of the Why3 deductive verification tool for the purpose of verification of distributed algorithms. The goal is to provide evidence that Why3 is a privileged tool for such a task, standing at a sweet spot regarding expressive power and practicality.

- Why3 interacts with all major proof tools used for first-order reasoning, both automated and interactive, with support for counterexample visualization.
- It supports replayability of proofs, as well as other proof engineering features such as bisection of hypotheses (which allows stripping down the context of successfully proved results, resulting in increased efficiency when replaying proofs) and inconsistency (smoke) detection, which allowed us to detect numerous subtle inconsistencies in our developments.
- The tool supports task transformations that support interactive proof steps without resorting to interactive proof assistants. In our distributed system models we conducted some proofs in this way. Automatic strategies are also supported, that iterate simple transformations and proof attempts with different SMT solvers.
- The Why3 IDE supports visual mapping of VCs to programs, with control-flow highlighting.

2.2.3 *Example with Why3 Tool - Euclidean Division Algorithm*

In order to understand how the Why3 tool really works, a simple example will be presented (see Listing 2.1). We have the traditional Euclidean division algorithm[11]:

```
let division (a:int) (b:int) : int
 =
   let q = ref 0 in
   let r = ref a in
   while !r >= b do
     q := !q+1;
     r := !r-b
   done;
   !q
```

Listing 2.1: Euclidean Division Algorithm

To verify this function it is necessary:

1. To respect the languages - logic and semantics - of the Why3 Tool when writing the problem
2. To write a contract: a pre-condition (‘requires‘) and a post-condition (‘ensures‘)
3. To provide an invariant and a variant for the while loop

So, having this is in mind, we have:

```
module Division

  use int.Int
  use ref.Ref

  let division (a:int) (b:int) : int
    requires { 0 <= a && 0 < b }
    ensures  { exists r: int. result * b + r = a && 0 <= r < b }
  =
    let q = ref 0 in
    let r = ref a in
    while !r >= b do
      invariant { !q * b + !r = a && 0 <= !r }
      variant   { !r }
      q := !q+1;
      r := !r-b
    done;
    !q

end
```

Figure 1: Euclidean Division Algorithm with Why3 Tool

It is also possible to see in Why3 Tool the Verification Condition (VC) generated and this will be easily proved by any solver:

```
------------------------- Local Context -------------------------

------------------------------ Goal -----------------------------

goal division'vc :
  forall a:int, b:int.
   0 <= a && 0 < b ->
   (((0 * b) + a) = a && 0 <= a) /\
   (forall r:int, q:int.
     ((q * b) + r) = a && 0 <= r ->
     (if r >= b
      then forall q1:int.
            q1 = (q + 1) ->
            (forall r1:int.
              r1 = (r - b) ->
              (0 <= r /\ r1 < r) /\ ((q1 * b) + r1) = a && 0 <= r1)
      else exists r1:int. ((q * b) + r1) = a && 0 <= r1 /\ r1 < b))
```

Figure 2: Generated Verification Condition

Figure 3: Proof of the Verification Condition

The lines of the solvers are all green checked, what means that the verification is done and the verification condition was proved. To be considered proved, at least one solver must successfully discharge the verification condition. Therefore, if we would have the next output, this will also considered valid and proved (not all green checked, but at least one):

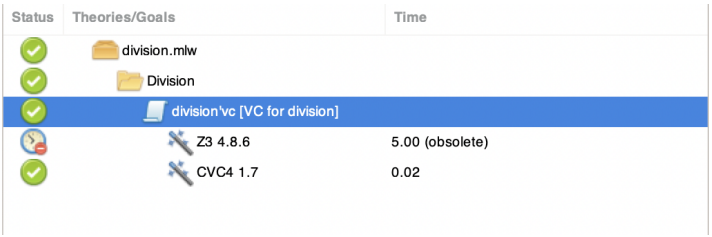

Figure 4: Proof of the Verification Condition with just one solver

In the case, that any of the solvers can not prove, the Verification Condition is probably not valid and means that the conditions that were defined are not right or not enough to verify the program:

Figure 5: Invalid Verification Condition

It will be useful to "split" this VC to observe that 4 separate conditions arise, associated with:

- invariant initialization and preservation
- post-condition (utility of the invariant)
- condition for decreasing the variant

Transformations like this split can be crucial to obtain sucessful proofs. In this case, it did not happen because the "whole" VC was immediately proved by one of the solvers, but in practice, there are proof objectives that are treatable by the solvers after being transformed by Why3.

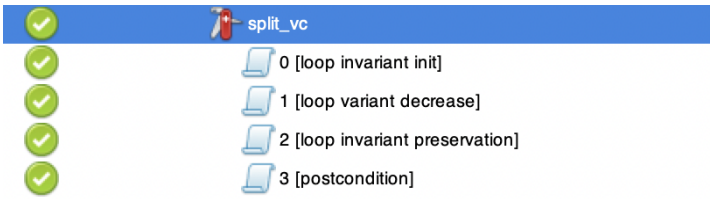

Figure 6: The "Split" of the Verification Condition

A very useful feature of the Why3 IDE is that it allows, by clicking on each VC, to observe in the code the instructions and conditions to which it is associated.

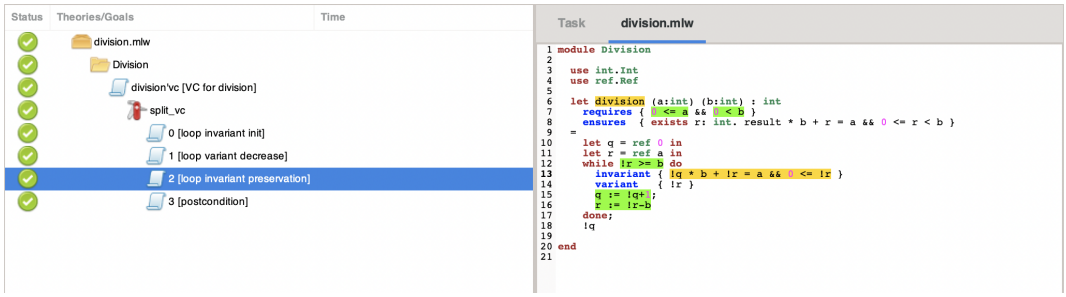

Figure 7: Click on [loop invariant preservation] VC and observe in the code the instructions associated

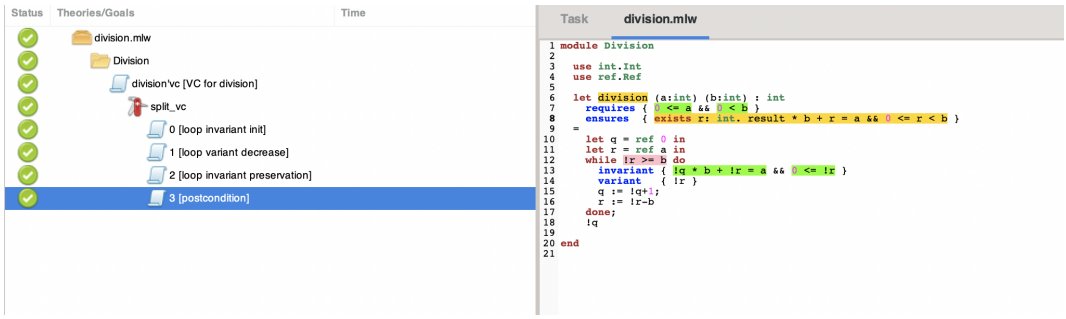

Figure 8: Click on [postcondition] VC and observe in the code the instructions associated

2.3 SIMPLE FORMALIZATION IN WHY3

In this section, it is going to be presented an example of a formalization in Why3. It is not a distributed system but it helps to understand and illustrates how the cloning mechanism works [4]. The example is the Fibonacci algorithm - see Listing 2.3 - which illustrates several things [12].

```
theory 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 }
    ensures { 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

end

theory 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
| others : 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 }
      invariant { !x >= 0 /\ forall r. fibpred !n r <-> r = !x }
      invariant { !y >= 0 /\ forall r. 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
```

Listing 2.2: Why3 example theory - Fibonacci Algorithm

In the first instance, the MapList theory imports theories for mathematical arithmetic and lists from the standard library, such as list membership, length, and indexation. Why3 contains a diverse set of theories that may be used to a variety of provers.

The val keyword is used to declare the function f, which has a basic contract: a precondition that the input must be nonnegative, and a postcondition that the result must also be nonnegative. This contract will be considered to hold for f in the rest of the theory. The nonNeg logic predicate is then defined. It declares that every element of its input list is nonnegative using a universal quantifier.

The map list function is then defined. Both the function's code and a contract are included in the definition. The code returns the list obtained by mapping the previously declared function f over the argument list; the contract prevents negative elements from appearing in the list, and the postcondition states the mapping property (result refers to the function's return value) using a universal quantifier and the nth list operator. As a termination measure, a variation (the argument list itself) is also given. Why3 will generate verification conditions (VCs) based on this theory to ensure that map list's definition is compatible with its contract. It's worth noting that this is achievable despite the lack of a definition for f.

This use of contracts is extremely important in deductive verification, but the example illustrates also a different aspect of Why3. The nonNeg predicate is a logic function that exists in a different namespace than WhyML

program functions like f and map list. Its definition includes a quantifier that cannot be used in programs. Pure program functions (without side effects) may also be utilized in the logic, if they are explicitly declared with the function keyword. This is the situation with f, which appears in both the code and the contract of map list. The expression "let function" will be used to refer to program functions that may be employed at the logic level. map_list is an example of a function that, although it is pure, cannot be used in logic because it is not specified as a let function.

Why3 encodes both the code and the contracts of let functions in logic, which implies that some logic functions may be written algorithmically, logically, or both. The nonNeg predicate, for example, might be defined as follows (the contract/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
```

Listing 2.3: nonNeg predicate

A second theory, MapFib, defines the calcfib program function, which uses a loop to compute Fibonacci numbers. We would want to employ the Fibonacci sequence's recursive definition in its contract, but because it's not total, it can't be stated simply as a logic function on integers. It may be defined as a let function with a domain restriction, but we'll use the inductive predicate fibpred instead: the formula fibpred n f denotes that f is the nth. Fibonacci sequence. Inductive predicate definitions, which will be familiar to proof assistant users, allow for the specification of a predicate using a collection of inference rules. Inductive predicates are helpful for establishing non-functional relations, and they are utilized to construct non-deterministic transition relations in our distributed systems models, which describe how system configurations develop.

Fibpred is utilized in calcfib's contract and loop invariants. With the aid of invariants specifying that variable references x and y contain the values of the two latest Fibonacci numbers computed, Why3 will produce and correctly discharge VCs, assuring the overall correctness of calcfib with regard to its contract. Now, because calcfib is in agreement with the MapList theory's contract of f, this theory may be cloned by replacing the latter function with the former. To guarantee that calcfib's contract is stronger than f's, this generates'refinement' verification conditions. Note that cloning will import a copy of every element of the cloned theory into the current theory, with calcfib replacing f. Finally, the (provable) lemma mapFib lm demonstrates that map list does indeed map the function calcfib.

3

STATE OF THE ART

For this section, I will start to mention previous work in the interest area with other tools. It will also contain information about a concrete example that will be one of the bases to solve this problem.

3.1 DEDUCTIVE VERIFICATION OF DISTRIBUTED ALGORITHMS WITH OTHER TOOLS

In recent years, a variety of tools have been developed expressly for supporting users with inductive invariant reasoning, often utilizing first-order logic or a variant of it. Because first-order logic is undecidable [13], we can classify these approaches into one of two groups:

- *Interactive Tools*, which assist the user in conducting proofs without much automation offered
- *Automated Tools*, which attempt to establish inductivity, and the validity of the desired properties, automatically - the proof tool is automatic but not the verification, we need to define the invariants anyway. These tools use SMT solvers instead of interactive proof assistants.

Following the first approach the Verdi framework, built on top of the Coq proof assistant, is a recent proposal that has had substantial impact. We also have the Ironfleet tool that interfaces with the Z3 SMT solver using first-order logic and the Dafny verifier.

Related to the second point, it is possible to find the Ivy tool. Ivy is a specialized tool that uses a decidable logic fragment to assist the user with the difficult task of discovering inductive invariants.

These three tools - Verdi, Ironfleet, and Ivy - are going to be more explored in the next sections and how they were a turning point in the verification of distributed algorithms.

3.1.1 *Verdi*

Verdi is an interactive proof assistant based on the Coq proof assistant [14]. This tool does not use automated provers or SMT solvers for first-order reasoning, being all this done and handled entirely within Coq. Verdi was specifically designed to work with asynchronous message-passing systems. The system model is made up of a collection of nodes, each of which communicates with its local context and with other nodes through messages. Using this local context it is possible to receive inputs and then send the correspondent outputs.

The semantics that is applied in this example is based on a notion of a world, where we have:

- Each node has a state associated with it
- A group of in-transit messages
- A sequence of interactions between the system and the external environment (i.e. a sequence of inputs/outputs received and written by system nodes) - Execution Trace.

The concept is that the system's specification is defined solely on the trace, with a clear abstraction separating it from unmentioned internal node states or in-transit communications. As a result, the system's specification is observational, as contrast to deductive formalisms like program logics or process algebras.

The system's evolution is described by a transition relation specified on worlds. When nodes receive local inputs or messages addressed to them over the network, transitions occur. Verdi distinguished himself from earlier attempts in this area:

1. Handlers describe how nodes respond when they receive inputs and messages, and are used to design specialized systems. This makes it possible to specify semantics once and for all, regardless of individual systems, as it should be.
2. Different network semantics can be used for reasoning to capture various fault models: for example, an idealized semantics will assume that messages are never lost or duplicated, but a more realistic semantics may include both possibilities in the formulation of the transition relation.
3. Verdi enables system transformations, which allow for the verification of more liberal network models from systems proven with a simpler model. For example, to compensate for the introduction of certain errors in the model, a redundancy mechanism may be automatically implemented in the system. The changed system does not need to be certified again because these system transformers are verified once and for all.
4. By extracting code to OCaml and combining it with a piece of code that implements the semantics, an executable system can be created from handlers written in Coq's Galina language. Verdi's model is asynchronous but based on an atomic even handler: each protocol step receives a packet/reads an input atomically, does local computations, and transmits packets/writes outputs, thus low-level interleavings are ignored.

This tool was used to verify the linearizability of a practical distributed system for the Raft consensus protocol.

3.1.2 *Ironfleet*

A team at Microsoft Research presented IronFleet **??**, a 3-layer framework for the verification of distributed systems, supporting refinement proofs between the layers, being an important progress for the area. Based on

this, then we have 3 layers: the first one is focused on abstract specification; the second layer is the protocol layer - being and abstract implementation and the third concerns the concrete implementation. The specification is written at the abstract layer, but unlike Verdi, where the spec is written based on observed input-output traces without establishing any connection with the protocol implementation. On the other side, the spec layer in IronFleet defines an abstract state and has already introduced an invariant relating the abstract state and the (abstract) implementation state.

For example, the abstract state of a lock service may be defined as a list of nodes ordered by the order in which they hold the lock, and an invariant indicating that if node n has delivered a message notifying that it obtained the lock in epoch n, then n is in position e of the list. The spec in Verdi wouldn't be able to refer to in-transit communications, therefore lock acquisitions would have to be registered by outputting the current epoch and then specifying that the trace is ordered by epochs in ascending order. The protocol layer is built on an asynchronous message-passing mechanism, similar to Verdi. Unlike in Verdi, where nodes are programmed by writing their various event handlers in Coq's programming language, in this tool, the nodes are described as state machines using the 'prime' notation, by writing predicates Init and Next that describe the initial state and state transitions, respectively, in first-order logic. Because the specification and protocol layers are separated, refinement must be asserted using an abstraction function and an inductive invariant defined at the protocol level. For encoding all of the foregoing formalization, as well as demonstrating inductive invariants and refinement findings, the Iroonfleet uses the Dafny first-order logic and program verification tool. In this case, the Dafny tool is utilized as a logical framework that interacts with the Z3 SMT solver to fulfill proof responsibilities.

At last, Dafny code makes up the implementation layer. The Dafny tool has its own programming language for program verification — the logic language, which is used to encode the first and second layers in Ironfleet, is really meant for reasoning about Dafny programs. The third layer provides a low-level trusted UDP specification (with Init, Send, and Receive methods) that use machine-level, bounded types rather than the mathematical types used in the previous two layers. A refinement proof, based on an abstraction function from implementation states to protocol states and a proof of correctness of the UDP primitives with regard to protocol-level network primitives, is required once again. Despite the fact that the protocol layer, like Verdi, is built on an atomic event handler, the system enables reduction-based reasoning and provides proof requirements that ensure the soundness proof assuming atomicity is likewise valid if atomicity is relaxed.

One other feature of Ironfleet, which we will just touch on quickly, is that it also focuses on reasoning about liveness qualities. It does so by including a Dafny embedding of the TLA temporal logic, as well as a library of validated TLA proof rules that are said to be useful for establishing the liveness qualities of distributed systems in general.

The authors were able to verify a Paxos-based replicated state machine library and a lease-based sharded key-value store using the Ironfleet tool.

3.1.3 *Ivy*

The Ivy tool, which was launched in 2016, shares certain similarities with the two prior frameworks, but it also varies in some key aspects [15]. Ivy does not recognize a difference between specification and protocol layers (as with IronFleet) or a notion of trail of observable operations (as does Verdi): the specification can relate to any element of the model. It introduces RML, a dedicated modeling (programming) language, as well as EPR, a logic language limited to the effectively propositional (EPR) class of formulae whose satisfiability can be determined. Note that this puts significant limits on RML modeling; for example, it does not enable arithmetic operations, thus although a ring topology may be expressed using integer modulo arithmetic in Verdi or Ironfleet, it must be encoded axiomatically in Ivy. Ivy checks satisfiability of formulae directly in Z3, rather than using an intermediary tool like Dafny.

Ivy focuses on supporting the user in properly developing the protocol and its specification, as well as finding appropriate inductive invariants, which is perhaps the most difficult component of distributed protocol verification. To this goal, the tool offers a few essential features that take use of the logic's decidability. The first method (influenced by the Alloy tool) is model debugging via constrained verification. This is the first stage in the Ivy workflow since bounded verification considering the first few steps of execution may allow defects in the protocol and/or the intended characteristics to be detected without the requirement for an inductive invariant. The user will next employ aided strengthening and generalization stages to try to obtain an acceptable inductive invariant interactively. Ivy helps the user by visualizing states and validating user-provided generalizations (again, using limited verification), displaying counter-examples, and recommending further generalizations.

The Ivy tool was used to partially verify the Chord peer-to-peer protocol, which implements a distributed hash table.

3.2 LEADER ELECTION ON A RING

We will now have a look at a famous distributed algorithm and explain how it can be formalized and proved with Why3 and SMT solvers, using a model from the Why3do library.

3.2.1 *How does this algorithm work?*

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 [16]. Let each node have a distinct identifier of some type equipped with a total order relation. Informally the algorithm can be described as follows:

- Each node starts by sending its id to the next node in the ring.
- If a received message has a higher value than the node's id, then the message is forwarded to the next node. Otherwise, the message is discarded.

- If a node receives back a message with its own id, it claims to be the leader.

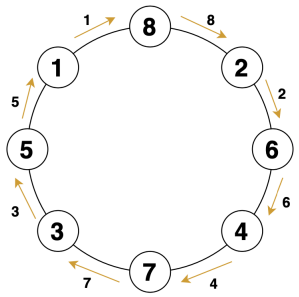

Figure 9: Leader Election on a Ring

3.2.2 *ModelMP and Theories*

It will be presented a model for message-passing systems [17]. It was necessary to develop and improve this model to later be able to prove the systems using it. It was defined as a library modelMP - message-passing model - where the data is shared by sending and receiving messages between cooperating processes. In this library, this was idealized as a fault model, and messages are not lost or duplicated. This library is split into two theories - World shown in Listing 3.1 and Steps, shown in Listings 3.2 and 3.3.

Importance of ModelMP and Theories

The definition of this modelMP library was really important because it is an abstract definition of the components we need to have in the various examples.

The modelMP is divided into two theories - World and Steps - the first one gives us the abstract definition of the world for which example and the second how our worlds evolve. After cloning the theories on a spefic example, it is defined a concrete definition of each field. This is very useful because it avoids writing the definition of their components for every example.

Why are there two Theories?

The World theory defines some composing types and this theory, in each example must be cloned once the basic types are defined to be possible to get additional types/functions.

The Steps theory, like the name suggests, represents the Network theory. Once this theory is cloned in each example will generate the Verification Conditions to ensure the proposed predicate to be the inductive invariant.

World Theory

In the defined model, it is defined the first theory (see Listing 3.1). It is known that the nodes communicate by exchanging packets. These packets are defined in this theory as triples (d, s, m), where we have d and s being Nodes and m a message. Each triple(d, s, m) is going to correspond to a packet and this packet will carry a message m from a node s to a node d. In this model, the Worlds are pairs where the first element assigns a state to each node and the second element is a network, , abstracted as a multiset of packets - it is defined the functions localState and inFlightMsgs to access to this information. The packet and the world, they have selectors that allow them to get their components when necessary.

```
module World
  use int.Int
  use map.Map
  use list.List
  use list.Append
  use list.Mem
  use list.Map as Lmap

  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
```

Listing 3.1: World Theory

Steps Theory

In the second Theory - Steps - it was written very important definitions for the way that the system evolves and the conditions the systems needs to respect to be proven. First of all we have the ok_Msg that defines if a message is well-formed. For example, in the leader election on a ring, the messages are always send to the next node, so a well-formed message needs to ensure this property and not send, for example, to two nodes after. This well-formedness predicate for packets and node states will be instantiated when cloning. Here it is also

defined an auxiliary predicate that compares two packets and it was also declared the components of the initial world, defined for each system - initState, initMsgs and initWorld. These definitions are shown in see Listing 3.2.

```
module Steps
  use int.Int
  use map.Map
  use list.List
  use list.Mem
  use list.Append
  use list.Map as Lmap

 (...)

  predicate eq_pckt (p:packet) (q:packet) =
    let (dp,sp,mp) = p in
    let (dq,sq,mq) = q in
    dp=dq /\ sp=sq /\ mp=mq

  predicate ok_NodeState node state
  predicate ok_Msg node node msg

  val function initState node : state
  val constant initMsgs : list packet

  constant initWorld : world = (initState, initMsgs)
```

Listing 3.2: Steps Theory - Part 1

The next step was to define the indpred predicate. This is a candidate inductive predicate. In this predicate, the contract ensures well-formedness of message and node states, and that the initial world satifies the predicate. In the post condition of the indpred, if the indpred in the world w is true means that w satifies this predicate for all packet that is part of the inFlightMsgs of this world w. So the postcondition says that if the inductive predicate is true, for any packet p that belongs to the in-transit messages, then these in-transit messages are well formed. Furthermore, we have a step_message that is a function that specifies how the outputs of the handlers are used both in the handler's contract and in the transition relation.

The handler function might be one of the most important ones which receives the node where the message is being handled and implements its local message-handling behavior. It also receives the node that sent the message, the message that is being sent and the state of the node handling the message. This will return the new state of this node and additionally the list of packets to be introduced in the network. It is required the well-formedness of packet and the state of the node where it is being executed are required. Case analysis predicates are introduced in the context and preservation of the inductive predicate is enforced in every world where handler is executed.

Then it was defined some more lemmas and definitions important for every system, for example, the step_msg tells us what is the world that results from w when a step is made - this consists of executing the handler in a given node. We can see all these definitions in Listing 3.3.

```
val ghost predicate indpred (w:world)
    ensures { w = initWorld -> result }
    ensures { result -> forall n :node. ok_NodeState n (localState w n) }
    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_one p (inFlightMsgs w)) in
    (localState, inFlightMsgs)

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

  lemma indpred_step :
    forall w w' :world. step w w' -> indpred w -> indpred w'

  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

  lemma indpred_manySteps :
```

```
    forall w w' :world. step_TR w w' -> indpred w -> indpred w'

  lemma indpred_reachable :
    forall w :world. reachable w -> indpred w
```

Listing 3.3: Steps Theory - Part 2

3.2.3 *Formalization of the Algorithm in Why3 Tool*

The most important property to be proved in this algorithm is that *at most one node will claim to be the leader* and our proofs below are based on the modelMP asynchronous model.

In this example, the first step is to define types for nodes, their states, and messages. Then, we also need a type for identifiers, uniquely associated to nodes by means of the **id** function and the **uniqueIds** axiom. Messages are just node ids. The constant **n_nodes** is the number of nodes in the ring. It is not assigned a value, so the proofs will be *unbounded*, but a minimum of 3 nodes is assumed in the ring.

The initial step in this example is to define types for nodes, states, and messages. Then, using the **id** function and the **uniqueIds** axiom, we'll require a type for identifiers that are uniquely associated with nodes. Messages are nothing more than node ids. The number of nodes in the ring is represented by the constant **n_nodes**. It is not assigned a value, which means the proofs will be unbounded, although the ring is believed to have a minimum of 3 nodes. The constant **maxId_global** corresponds to the (unique) node having the *highest-value id* in the ring.

Node states only have a single field **leader** of Boolean type, which will be true when a node claims to be the leader.

Finally, we define the appropriate notion of *well-formed message* in the ring topology, by means of the **ok_Msg** predicate. All of these definitions may be found in the Listing 3.4 below.

```
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 }

predicate ok_NodeState node state = true

type msg = id

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


let predicate case_node (_node) = true
let predicate case_state (_state) = true
let predicate case_msg (_msg) = true
```

Listing 3.4: Leader Election on a Ring - Basic Setup

There is a remark about the choice of types for the nodes and their identifiers. It could be possible to leave these types undefined which would make the **next** function and the **maxId_global** constant give us a set of axioms instead of being defined. Based on what we know, using concrete types (Why3 library types when appropriate) and definitions of constants, predicates, and functions whenever possible improves provability and minimizes the risk of inconsistencies. Considering the **maxId_global** constant, which is defined constructively here using the let function **maxId_fn**, that recursively calculates the node with the highest id in the first **n** nodes. This function has a contract that could be given simply as an axiom on **maxId_global**, removing like this, the function definition altogether. However, using the let function not only increases our confidence that we are specifying what is meant, but it also generates a logical encoding that may be more beneficial for proofs requiring **maxId_global** due to the nature of let functions in Why3.

Cloning the World Theory

Cloning the theory **modelMP.World** will introduce some new composed types and auxiliary definitions (see Listing 3.5). The system description then goes on to describe the system's initial conditions, using a function called **initState**, which maps every node to an initial state in which **leader** is false, and a constant **initMsgs** corresponding to the list of messages sent by the system's nodes after booting in this example, one message for each node, carrying its id, sent to the next node in the ring. The **initMsgs** is defined by a recursive let function,

however notice that the initial world is described at the logic level, so **initMsgs_fn** is not meant to be executed in the nodes it is written as a let function just to make possible establishing termination through a variant.

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

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

let function initState _node : state = { leader = false }

let rec function initMsgs_fn (n:node) : list packet
    requires { 0 <= n <= n_nodes }
    variant  { n_nodes-n }
    ensures  { forall s d :node, m :msg. mem (d, s, m) result ->
                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) }
  = 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
```

Listing 3.5: Leader Election on a Ring - Cloning World Theory and auxiliary definitions

The message handler definition then follows (see Listing 3.6), according to the description given earlier in this section. The call **handleMsg h src m s** is executed by node **h** with current state **s** when it receives message **m** from node **src**, and returns the new state of **h** and a list of message packets to be sent.

```
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)
```

Listing 3.6: Leader Election on a Ring - Message Handler

The invariant predicate **indpred**, which in this example concerns the state of the nodes and the list of in-flight messages, is the next element in the module (see Listing 3.7). The invariant must be expressed as a let predicate since it will instantiate a **val predicate** declaration in the model, which is another unique feature of our library. However, in order to define it, we must plainly use logic essentials like quantifiers and equality. The **let ghost predicate** provides a solution to these contradictory requirements. The ghost code components are allowed

to activate logic-level functions because they are included in programs for the purpose of specification and are not intended to be executed. As a result, the indpred is specified as a ghost predicate that calls the inv logic predicate.

The invariant states that every inflight message is well-formed; its contents is 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 (in the ring) between i and the message's destination node (an auxiliary predicate between is used to express this notion). Moreover if the message contains its destination's id then that id is the highest in the network. Finally, the invariant expresses the fact that if any node is claiming to be the leader, then that node has the highest id in the ring.

```
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)
```

Listing 3.7: Leader Election on a Ring - Invariant Predicate

Cloning the Steps Theory

The module then clones the Steps module from the modelMP model, and formulates the uniqueLeader proof goal stating that in any reachable world at most one node is claiming to be the leader (see Listing 3.8).

```
clone modelMP.Steps with
    type node, type state, type msg, predicate ok_NodeState, predicate ok_Msg,
    val case_node, val case_state, val case_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 3.8: Leader Election on a Ring - Cloning Steps Theory and definition of the proof goal

The results of running Why3 on this module depend on the provers that are available. In our setup we were able to prove automatically all verification conditions using the Alt-Ergo and CVC4 solvers, but this required the following additional lemma about the between predicate, proved automatically by Alt-Ergo (see Listing 3.9).

```
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
```

Listing 3.9: Leader Election on a Ring - Additional lemma about between predicate

In conclusion, after having all the necessary lemmas and definitions, it was possible to achieve the proof of the Leader Election on Ring Algorithm. We can see its results in the following image:

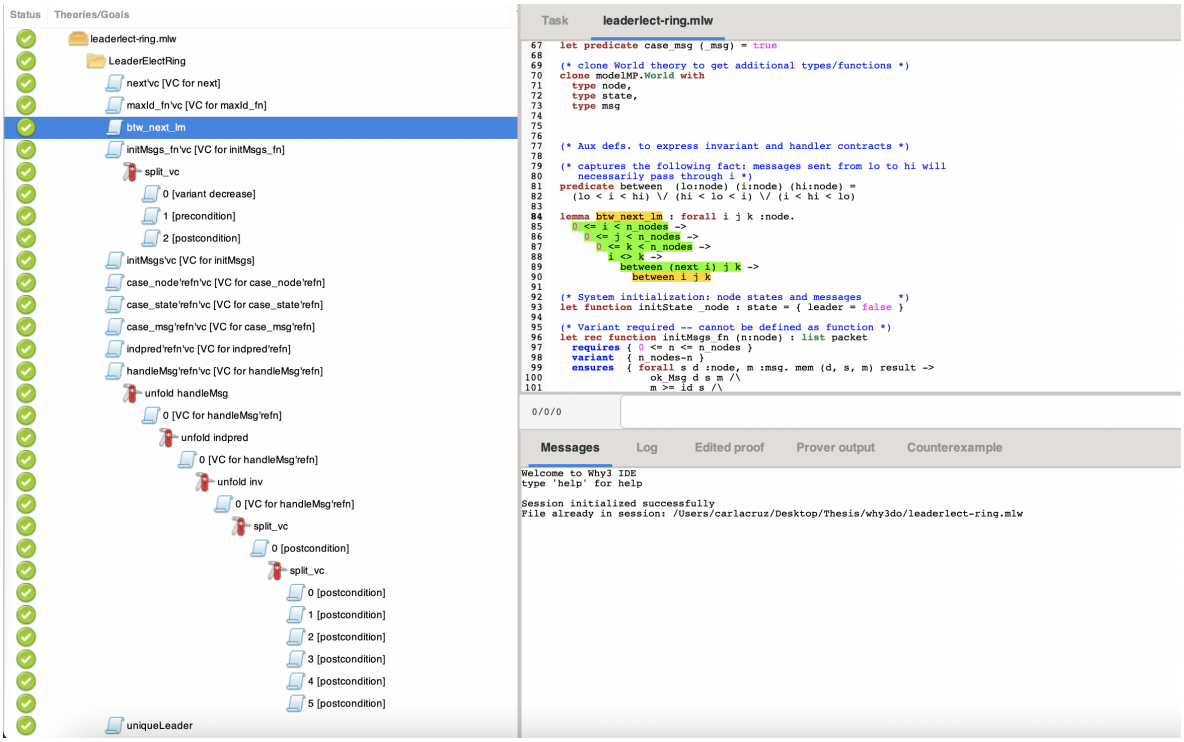

Figure 10: Verification Conditions and transformations proved in Why3 Tool

3.3 DIFFERENCES BETWEEN WHY3 TOOL AND OTHER TOOLS

An important analysis to do at this point is to understand the principal differences between the Why3 and other tools. It is essential to understand why this new approach it has some strengths and more advantages when compared to the examples that were previously mentioned. One of the system's limitations is that complex systems have yet to be verified.

One of the most significant differences between the Why3 and Verdi is that the Why3 tool and its library - Why3 do - uses SMT solvers and on the other side, Verdi is based in Coq. Once we want to privilege the automated tools, here the Why3 was a plus because Verdi is an interactive tool, assisting the user in conducting proofs without much automation offered.

In relation to the Ivy tool, this is more focused to help the user to incorporate the invariants when the Why3 tool does not have this configuration. Despite this, the Ivy only uses one solver - Z3 solver - in comparison to Why3 which can have multiple ones. A disadvantage of Ivy is established with strict first-order logic - EPR logic - and this means that does not use the complete first-order logic.

The Iroonfleet tool only supports one solver - Z3 solver - and the Why3 supports many solvers, helping with the proof of the systems. The Why3do is completed based on a model design by contract, which means that we use the same methodology for the imperative and for the distributed elements, which is not possible in the Ironfleet tool.

The main difference between the Why3 tool and all the tools is that the Why3 provides a rich specification and programming language called WhyML. The specification language of Why3 is an extension of an imperative language. One interesting feature of Why3 is that it can translate and communicate the verification conditions to many external provers. We consider the Why3 tool more versatile and more user-friendly.

4

SELF-STABILIZATION MODELS

Many researchers have been studying self-stabilization in recent years, and they continue to try to validate these algorithms using a variety of tools. Many self-stabilization models have been approached, as well as many other distributed algorithms that have been the focus of the Formal Verification Community. Some of the proofs verify properties of safety or liveness, or in some situations both. Techniques that have been used include Manual Proofs, Model Checking, Theorem Proving, and some of them can be checked with the help of some testing tools are some of the methods used to perform verification. [18]

4.1 WHAT ARE SELF-STABILIZATION MODELS?

Starting from any arbitrary configuration, a self-stabilizing algorithm ultimately achieves a legal configuration, and this property is stable in the sense that it remains in a legal configuration after that. The concept of which configurations are considered legal depends on the task at hand, but the general idea is that an algorithm is self-stabilizing if it can recover from arbitrarily bad faults and stay recovered as long as no new errors occur. [19]

4.2 STABILIZING MUTUAL EXCLUSION ON AN UNIDIRECTIONAL RING - DIJKSTRA ALGORITHM

4.2.1 *The Importance of this Proof*

One of the challenges was to select an example/algorithm and make its proof, similar to the others and so contribute to the growth of this community after researching and becoming familiar with the past work performed so far in the verification of distributed algorithms. We chose this example because, based on previous work, it was critical to attempt to make the proof of this method as simple as possible. We feel that using SMT solvers instead of interactive proofs using proof assistants might be an extremely interesting objective to attain and a significant step forward in this field.

In the work of Shaz Qadeer and Natarajan Shankar [20], the authors mentioned that even simple concerns like how to describe the pigeonhole principle in its most relevant and useful form, how to structure the measure function, and when to utilize measure induction rather than conventional induction took a lot of time and effort to solve. Even simple concerns like how to describe the pigeonhole principle in its most relevant and useful form,

how to structure the measure function, and when to utilize measure induction rather than conventional induction took a lot of effort to solve. Also in the work of Stephan Merz [21], this author writes that the main structure of the argument is not obvious and must be re-established by a careful reader. There isn't much of a reason why a certain lemma is required. It's not clear whether portions of the evidence may be left to an automatic prover and how they should be handled.

This demonstrates how important it was for us to conduct this proof since we wanted to achieve the aim of making the proof of this example as clear and simple as possible while avoiding the issues that had been discovered in prior attempts.

4.2.2 *General Description of The Procedure*

As stated in the preceding section, our goal was to create a clear and succinct proof using SMT solvers. Like this, it is feasible to characterize the complete development of the proof in generic terms:

- The *.mlw* file contains around 500 lines
- The proves only use SMT solvers - Z3, CVC4, Alt-Ergo - and transformations for the indutive proves - split_vc, inline_goal, split_all_full, induction and remove. It is possible to apply a transformation on a goal instead of calling a prover on it. If we have a conjunction, for example, we can split it into subgoals. Sometimes we need a transformation, which we can then can apply more transformations, or we can just create one transformation and prove it immediately with the solvers. We may infer that this proof is not completely automated, because certain transformations are required, but they are almost automatic.
- Let functions are heavily used in the proof and definition. As a result, the majority of lemmas are stated in the contracts of these functions, making formalization more natural.

4.2.3 *Description of The Problem*

When we have failures resulting from corruption of data memory, the self-stabilizing systems were designed to tolerate this type of failure integrating into the code a recovery engine [22].

The system is considered self-stabilizing if:

- It starts from an illegal configuration and eventually converges to a legal configuration - **Convergence Property**
- Legal configurations are closed under normal execution steps. If there is no error occurring in corruption of data, it is not possible to reach an illegal configuration from a legal configuration - **Closure Property**

Dijkstra proposed the following for self-stabilizing mutual exclusion on a unidirectional ring: processes have integer numbers in the range 0 to K as states, with K a constant with a value greater than the ring's size [23]. Each process in the ring checks the state of its predecessor; the process with index 0 is enabled (i.e. 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 differs from that of their predecessor. When enabled, each process can change its state by copying the state of its predecessor; node 0 also increments this state (modulo K) after copying it.

It is possible to understand how the system operates by looking at the figure below. It starts with an illegal configuration in the example (having two tokens - the nodes that are colored). The system reaches a legal configuration in the first transition, and it is possible to comprehend how tokens flow. With this simple illustration, we can get a sense of the system's overall dynamics.

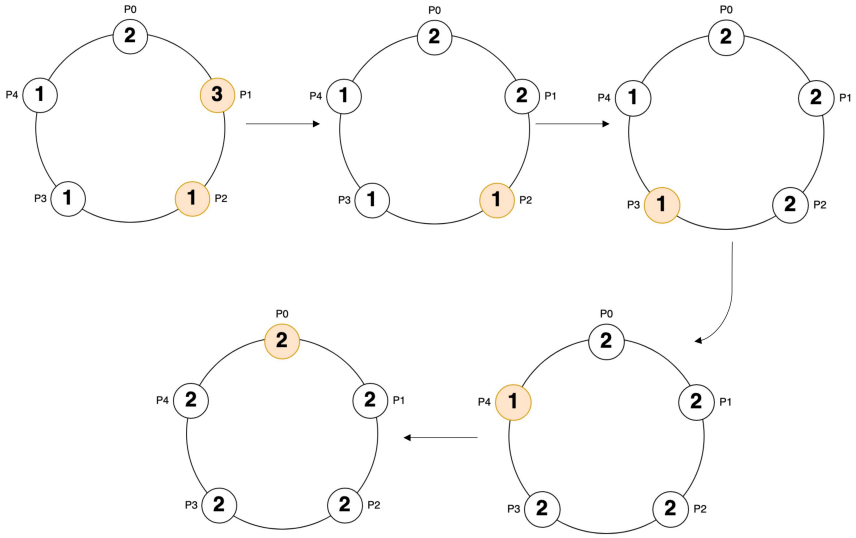

Figure 11: Stabilizing Mutual Exclusion on an Unidirectional Ring - Simple illustration how the system works

To prove the self-stabilization system, the verification was divided into two separate modules, each of which proved a different property. Both modules make a clone of the guarded processes model - ModelReadallEnabled.

ModelReadallEnabled

Dijkstra used a guarded processes model to explain certain distributed systems, particularly self-stabilizing systems like the Stabilizing Mutual Exclusion on an Unidirectional Ring [24]. Instead of exchanging messages, nodes in this paradigm are allowed to view each other's states directly. Only when the guard predicate is true may a process run. This is not a shared-memory paradigm in the strictest sense, but it can run on shared memory if each process has a single-writer multiple-reader data structure.

In our context, we define this as a why3do model, in which worlds are simply of the type map node, with the function node -> state, assigning a state to each node (see Listing 4.1).

```
module World

  type node
  type state

  type world =  map node state

end
```

Listing 4.1: World Theory for Guarded Processes Model - Part 1

When designing a system based on this model, it is required to first establish an enabling predicate on nodes, and then create an enabled handler that may be executed by a node whenever its enabling predicate is true (see Listing 4.2). The handler code and the enabling predicate both have read access to all node states, but the handler can only change the state of the node on which it is now running. The system we are defining will only need access to a limited set of node states - namely, the states of its immediate neighbors. Such constraints, however, will be imposed at the system level for the sake of generality, with the model enabling all node states to be read. The function step_enbld also defined below is important for the enabling predicate - that takes as parameters a node and a global state assigning function- and handler, specifies how the outputs of the handlers are used both in the handler's contract and in the transition relation.

```
 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 }
    requires { enabled lS h}
    requires { indpred lS }
    ensures  { indpred (step_enbld lS h result) }
```

Listing 4.2: Steps Theory for Guarded Processes Model - Part 1

The handleEnbld is the handler to be executed at node h when it is enabled in the world lS. Preservation of the inductive predicate is enforced. The rule can choose any enabled node to run the handler and change its own state. The implementation architecture necessary to run a system based on this paradigm will utilize a mechanism like a readers–writer lock to manage access to each node's state.

Another important predicate, to then be used in the system we are defining next, was the step predicate (see Listing 4.3). This defines transition semantics, in the form of an inductive invariant where we say which was the node where this transition happened. This was very useful for the convergence proof and further, it will be shown its importance.

```
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))
```

Listing 4.3: Steps Theory for Guarded Processes Model - Part 2

4.2.4 *Self-stabilization Properties*

Cloning the guarded processes model described in the previous section and introducing additional definitions, the properties of closure and convergence [25] will be demonstrated. The similarities between the two proofs will be demonstrated in this section.

In these properties, the handler code for both proofs is the same, but the enabling predicate and invariant are not. The enabling predicate is different because in the convergence property we only want transitions because we used a termination strategy to prove convergence: in reality, the transitions do not terminate, but we presume that the first part stops, based on the "converged" predicate, and we prove that termination. This does not happen in the case of closure. The invariant is not the same because, in order to prove the closure property, the invariant assumes that the configurations are legal (in particular, that there are no more than one token), and since it is not possible to start from this principle for convergence, the invariant considers the limits on the values of the states.

Another point worth mentioning is that the convergence proof presupposes that the system nodes have access to processor time - a condition known as "fairness" that is usually required when proving liveness properties. Nothing can be assured if a node in the middle of the ring never executes.

We begin by defining a Basic Setup (see Listing 4.4) for both properties. In which module, the Basic Setup is composed of Nodes, Packets, Inputs, Outputs, and States. Nodes and their states are both defined as integer type; n_nodes is the size of the ring and k_states is the number of different states. It was necessary to define the bounds of these two val constants, so it was declared two axioms: n_nodes_bounds and k_states_lower_bound. We also have the let function incre, that increments the state, returning the new state, after the incrementation.

```
type node = int

val constant n_nodes : int

let predicate validNd (n:node) = 0 <= n < n_nodes

axiom n_nodes_bounds : 2 < 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
```

Listing 4.4: Stabilizing Mutual Exclusion on an Unidirectional Ring - Basic Setup

The let function handleEnbld (see Listing 4.5) is the handler to be executed when a process is enabled, allowing it to copy its predecessor's state. This predicate is defined the same way for both properties.

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

Listing 4.5: Stabilizing Mutual Exclusion on an Unidirectional Ring - Handling Function

Moreover, also in common is the has_token predicate (see Listing 4.6). This takes a state function - maps nodes to states- and specifies when a node in the ring has the token; it is then used to define the enabled guard predicate for both properties.

```
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))
```

Listing 4.6: Stabilizing Mutual Exclusion on an Unidirectional Ring - Has_token Predicate

In addition to these definitions, we will now go through each of the properties in further depth.

4.2.5 *Closure Property*

Besides the common definitions presented before and towards proving the Closure Property, it was necessary to define in order to express the mutual exclusion property two predicates that are applied to the first n nodes, atLeastOneToken and atMostOneToken (see Listings 4.7 and 4.8). These were helper definitions for the invariant predicate, and as the name suggests, these predicates guarantee that the system has at least one token and at most one token, respectively.

```
let rec ghost predicate atLeastOneToken (lS:map node state) (n:int)
    requires { 0 <= n <= n_nodes }
    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))
```

Listing 4.7: Closure Property - atLeastOneToken predicate

```
val ghost predicate atMostOneToken (lS:map node state) (n:int)
    requires { 0 <= n <= n_nodes }
    ensures { result <-> forall i j :int. 0<=i<n -> 0<=j<n -> has_token lS i ->
        has_token lS j -> i=j }
```

Listing 4.8: Closure Property - atMostOneToken predicate

The predicates defined before were essential for the definition of the inductive predicate (see Listing 4.9) because to prove the closure property, the invariant says the configurations are always legal - in particular there is no more than one token.

```
predicate inv (lS:map node state) =
    (forall n :int. 0 <= n < n_nodes -> 0 <= lS n < k_states)
    /\
    atMostOneToken lS n_nodes

let ghost predicate indpred (w:world) = inv ( w)
```

Listing 4.9: Closure Property - Inductive Invariant

The enabling predicate (see Listing 4.10) will be defined as true for a process (a node in the model) exactly when it is carrying a token.

```
let ghost predicate enabled (lS:map node state) (i:node)
  = has_token lS i
```

Listing 4.10: Closure Property - Enabling Predicate

The module concludes with oneToken (see Listing 4.11), which proves that there is exactly one token in all configurations reachable under normal conditions from our specified starting world.

```
predicate oneToken (w:world) = atMostOneToken w n_nodes /\ atLeastOneToken w
    n_nodes

goal oneToken : forall w :world. reachable w -> oneToken w
```

Listing 4.11: Closure Property - One Token goal and predicate

Because we are conducting an unbounded proof, the only way to get this is to include a lemma in the module that says if nearby processes share the same state, then the first and last processes must as well. This has as consequence the fact that in every world there exists at least one token. Note that, since this holds by definition, there is no need to include this information in the inductive invariant.

Interestingly, if *atMostOneToken lS n_nodes* is placed in the invariant, then it is possible to prove the goal without stating the lemma first_last, as long as a fairly low bound is placed on the number of nodes.

```
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
```

Listing 4.12: Closure Property - first_last lemma

To prove this lemma (see Listing 4.12), the induction transformation is used to establish this lemma through induction on n. It enables the aim to be automatically proven with no upper constraint on n nodes and without putting atLeastOnetoken lS n nodes in the invariant - before it was necessary.

Results of Closure Property Proof with the Why3 Tool

In conclusion, after having all the necessary lemmas and definitions, it was possible to achieve the Closure Property Proof with the Why3 Tool of the Stabilizing Mutual Exclusion on an Unidirectional Ring Algorithm. We can see its results in the following image:

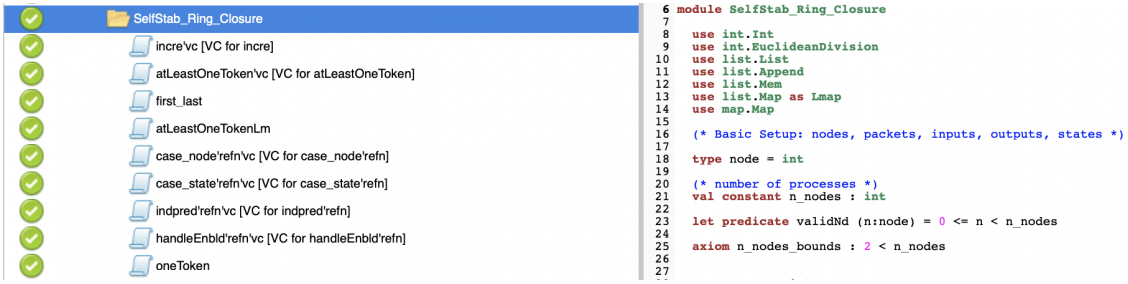

Figure 12: Verification Conditions and transformations proved in Why3 Tool - Closure Property

4.2.6 *Convergence Property*

To understand a little bit the convergence and how this must be proved, we call upon the Hoare Logic [9], that the main idea is the loop that ends based on a variant. This variant should be a natural value in which every iteration of the loop, strictly decreases. This variant should answer a well-founded order. A well-founded order is a relation that doesn't have infinite chains of bigger values. Beyond the condition that enforces that the variant strictly decreasing aspect, there is also a global condition $I \wedge b \rightarrow V > 0$. This means that in each iteration this value needs to be always positive. So in fact the minimal value is the target, and $V = 0 \rightarrow \neg b$.

In the self-converging system, we do not have a termination. A self-converging system is going to try to reach a legal configuration but the system does not stop. This convergence property is not a termination property. In the loop cases, the loop stops but in the convergence property what ends are the illegal configurations, we reach a legal configuration and then we will have the closure property shows that after that is always legal configurations.

In this case, instead of working with a complete transition relation, it is going to be used a transition relation with restrictions that is only going to work while we do not reach a legal configuration, so like this, it is possible to work on this problem as a termination problem, so it will be similiar to the termination of a transition relation. Based on this, we will need to have a boolean condition that says when the system can procede. In the example of the self-stabilization ring, the role of the condition b is similar to the transition-enabling predicate notConverged, which we include in the enabling predicate of every node, and is, in fact, responsible for creating a terminating version of the transition relation. In order to prove the convergence and based on what we presented before, we need to:

- make the transition relation terminating using the notConverged condition, that identifies a target legal configuration (not the first one occurring, but one that will occur for sure in any trace starting from a given initial configuration).
  - This is done by identifying the target state, which will be the same for all nodes,
  - for this we first prove that the target legal configuration is invariant with transitions, from any configuration that is not its target.
- define the measure function, also based on the target legal configuration.
- show that notConverged (if we can still do a transition) implies that the measure function has a positive value [see the Hoare Logic analogy].
- show that the measure function is strictly decreasing.
- show that the target configuration is a legal one, i.e. all executions terminate in a legal configuration [the transition where it stops, it is an legal configuration and after this, starts the closure property].

The transition relation that ends as soon as a legal configuration is established would be difficult to describe, which is a crucial factor to keep in mind. It is easier to think about stopping when you achieve a specified legal configuration, such as when node 0 gets the token, implying that all nodes are in the same state. Because the token constantly rotates to the right after reaching any legal configuration, it inevitably reaches node 0. The following example demonstrates this:

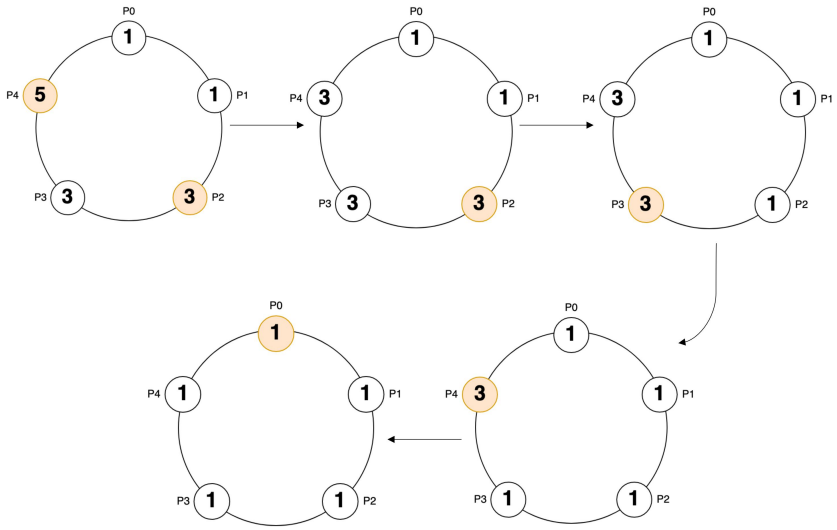

Figure 13: Stabilizing Mutual Exclusion on an Unidirectional Ring - Example that shows the flow of the ring's token

It is important now to show and explain the converged predicate (see Listing 4.13) that says that in the target configuration for the convergence phase all nodes have the same state, thus (only) node zero holds a token.

```
predicate converged (w :world) =
  forall i :node. validNd i -> w i = w 0

lemma converged_0hastoken : forall w :world.
  converged w <-> (has_token w 0 /\ forall i :int. 0<i<n_nodes -> not (
      has_token w i))
```

Listing 4.13: Convergence Property - converged predicate

As it was mentioned before, the enabling predicate and the invariant are different from the closure property. We only want transitions to occur while we have not reached such target configuration where the token is at node 0, so that we can prove the termination of these transitions. For this property, the enabling predicate for a given node ensures that the transition relation being considered here indeed terminates when the target legal configuration is reached.

```
let ghost predicate enabled (lS:map node state) (i:node)
  = has_token lS i && not (converged lS)
```

Listing 4.14: Convergence Property - enabling predicate

About the candidate invariant predicate (see Listing 4.15),it only takes in consideration the limits of the values of the states. Here, we do not consider that the configurations are always legal like we did in the closure property because, we work with illegal configurations - configurations that have more than one token.

```
predicate inv (lS:map node state) =
    (forall n :int. validNd n -> 0 <= lS n < k_states)

let ghost predicate indpred (w:world) = inv (w)
```

Listing 4.15: Convergence Property - invariant predicate

Convergence Measure

We implement a proof approach that is quite similar to the variants proof to accomplish the convergence proof. With this in mind, we needed to create a function that would provide us with a measure of convergence. The premise that it must be a non-negative integer number and that it must satisfy two requirements, which are the same as the versions in Hoare's reasoning, is crucial. The two requirements are then:

- The measure must have a value greater than 0 whenever a configuration can execute a step (i.e. the execution has not yet ended). This ensures that if the value is 0, the execution is finished.
- The measurement value must be decreased strictly during the execution of all steps.

```
let ghost function measureNode (w:world) (i:int) : int
    requires { indpred w }
    requires { validNd i }
    ensures  { result >= 0 }
    ensures  { i=0 -> convState w >= w i -> result = n_nodes * (convState w - w i
        ) }
    ensures  { i=0 -> convState w <  w i -> result = n_nodes * (k_states - w i +
        convState w) }
  = let p = if i=0 then diffZero w (convState w)
                   else if has_token w i then 1 else 0
    in (n_nodes-i) * p

let rec ghost function measureNodes (w:world) (n:int) : int
    requires { indpred w }
    requires { 0 <= n <= n_nodes }
    ensures  { result >= 0 }
    ensures  { result = 0 -> forall i :int. 0<=i<n -> measureNode w i = 0 }
    variant  { n }
  = if n=0 then 0
    else measureNode w (n-1) + measureNodes w (n-1)

let ghost function measureAllNodes (w:world) : int
```

```
    requires { indpred w }
    ensures  { result >= 0 }
    ensures  { not (converged w) -> result > 0 }
  = measureNodes w n_nodes
```

Listing 4.16: Convergence Property - measureNode, measureNodes and measureAllNodes predicates

```
goal step_decreasesMeasure : forall w w' :world, k :node.
    indpred w -> step w k w' -> measureAllNodes w' < measureAllNodes w
```

Listing 4.17: Convergence Property - step_decreasesMeasure goal

It is clear that the previously indicated criteria are incorporated into the contract of the measureAllNodes function (see Listing 4.16) being this the definition of our convergence measure function, respecting the premises that need to respect. So here it was proved one of the first properties, that before convergence, the measure (which is always non-negative) has a positive value. The measureNodes function performs a sum, and it does this using the measureNode predicate, which specifies what each node is added to. The functioning of the measure of nodes 0 to n-1 and also the measure of a single node will be explored in significant detail in a section after.

After this is used to prove our goal step_decreasesMeasure (see Listing 4.18), which indicates that this measure is decreasing with each step taken in our world - the value in the new world (w') is smaller than the previous one (w) after taking one step. We can prove a second property that says that the measure strictly decreases with every step of the system.

The measureDeltaNodes function is used to demonstrate the step decreasesMeasure goal. The following are the two post-conditions:

- ensures { result = measureNodes w' n - measureNodes w n }
- ensures { n = n_{nodes} -> result < 0 }

Together, they ensure that the difference between the measures of two configurations (when one transitions to the other) is always negative, implying that the measure decreases. We need to include "n = n nodes," which is when the measure applies to the complete ring, since this method only calculates it for the first n nodes.

```
let rec ghost function measureDeltaNodes (w:world) (w':world) (n:int) (k:node) :
    int
    requires { indpred w }
    requires { 1 <= n <= n_nodes }
    requires { validNd k }
    requires { step w k w' }
    ensures  { result = measureNodes w' n - measureNodes w n }
    ensures  { 0 <= n-1 < k -> result + n_nodes * mod (convState w - convState w'
        ) k_states = 0 }
    ensures  { n>1 -> k = n-1 -> result + n_nodes * mod (convState w - convState
        w') k_states = n-1-n_nodes }
```

```
  ensures  { n>0 -> k = 0 -> convState w' <> convState w -> result + n_nodes *
      (diffZero w (convState w)) <= 0 }
  ensures  { n>0 -> k = 0 -> convState w' =  convState w -> result < 0 }
  ensures  { 0 <= k < n-1 -> result < 0 }
  ensures  { n = n_nodes -> result < 0 }
  variant  { n }
= if n=1 then measureDeltaNode w w' 0 k
  else measureDeltaNode w w' (n-1) k + measureDeltaNodes w w' (n-1) k
```

Listing 4.18: Convergence Property - measureDeltaNodes function

Measure Function

To complete and explain the calculation of the measure, we used the calculation of the convergence state, i.e the state in which all nodes are at the end of this phase, as a starting point. There are two scenarios:

- The convergence state is defined as the value of the nodes' state of a prefix in which all nodes have the same state and that state does not recur in the ring. The following example, includes numerous tokens however, it must converge in the following configuration.

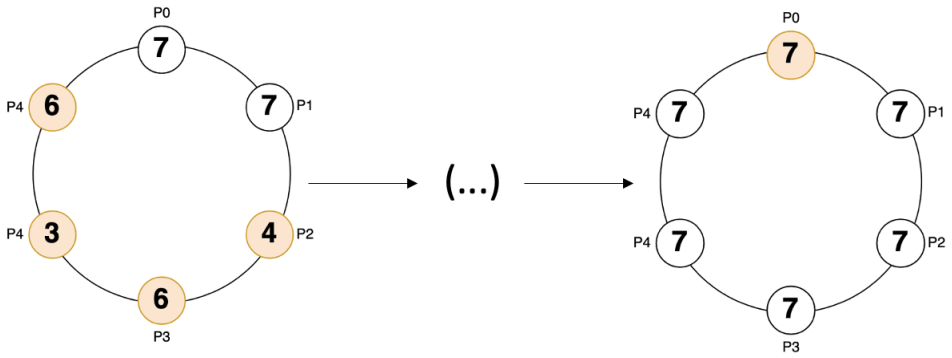

Figure 14: Stabilizing Mutual Exclusion on an Unidirectional Ring - Convergence state defined as a prefix

- Otherwise, the convergence state will be a value not found in the ring: the one with the smallest difference from node 0's present state. For instance, in the first example, we have scenario 1 occur again, and it converges to 777777. The only state after 6 that does not occur in this example is 7, however since the states were only 0 to 6, that state would be 0 in this case (the increment is mod K). There are two instances of 6, but regardless of the execution order, a 6 will always wind up in the final place, at which time node 0 will transition to 7 (or to 0 if 7 does not exist).

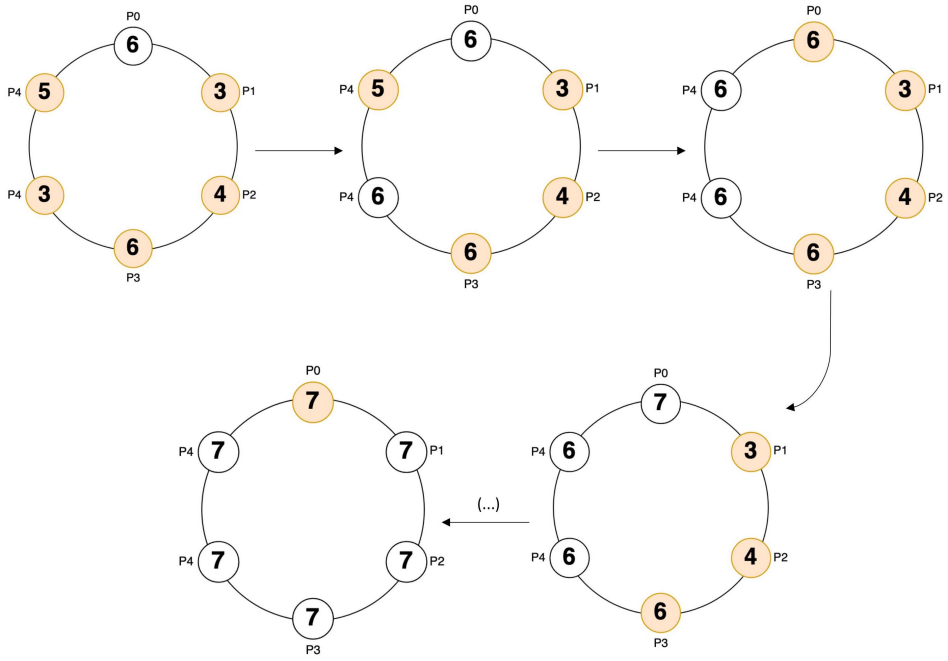

Figure 15: Stabilizing Mutual Exclusion on an Unidirectional Ring - Convergence state is a value not found in the ring

Now we will have a look at the functions initConv, diffZero, and convState, which translate the convergence state calculation and its properties (see Listing 4.19). The first is a predicate that returns true for the configurations where the state of node 0 only appears in a prefix. Thus, this function returns false for the previous figure - Figure 15 - and returns true for the other example - Figure 14.

The function diffZero calculates the "distance" between a given state s and the state of node 0. This distance is then the difference mod the number of states. This function is used in the convState function. In the case where the state 0 occurs in the middle of the ring (not just in a prefix), the convergence state is the "smallest" - the least distant from the 0 state, which does not occur in the ring.

The convState function returns the convergence state , which is the state of node 0, if it only occurs in a prefix of the ring (and thus the initConv predicate is true), or else it is the non-occurring state that is closer to w 0. For this to be proven uses in its contract the diffZero, ensuring the properties of the convergence state calculation.

```
predicate initConv (w:world) =
  exists j : int. 0<=j<n_nodes /\ (forall k :int. 0<k<=j -> w k = w 0) /\
      noOcc_from w (w 0) (j+1)


let ghost function diffZero (w :world) (s :state)
  requires { 0 <= w 0 < k_states /\ 0 <= s < k_states }
  ensures  { 0 <= result < k_states }
  ensures  { w 0 <= s < k_states -> result = s - w 0 }
```

```
  ensures  { 0 <= s < w 0 -> result = k_states + s - w 0 }
= mod (s - w 0) k_states

let ghost function convState (w:world) : state
  requires { forall n :int. validNd n -> 0 <= w n < k_states }
  ensures  { 0 <= result < k_states }
  ensures  { not (initConv w) -> noOcc w result }
  ensures  { noOcc_from w (w 0) 1 -> result = w 0 }
  ensures  { initConv w <-> result = w 0 }
  ensures  { w 0 <= result < k_states ->  forall s :state. w 0<=s<result -> not
       (noOcc w s) }
  ensures  { 0 <= result < w 0 -> forall s :state. w 0<=s<k_states \/ 0<=s<
      result -> not (noOcc w s) }
  ensures  { forall s :state. 0<=s<k_states -> s<>result -> noOcc w s ->
      diffZero w result < diffZero w s }
= if (initConv w) then (w 0) else convState_hlpr w (w 0) 1
```

Listing 4.19: Convergence Property - initConv, diffZero, and convState functions

As previously said, measuring the convergence - measureNodes function (see Listing 4.16) - was required to prove this approach. To formally comprehend how this was accomplished, we used the formula below:

$$\boxed{\Phi(l) = n((c - l_0) mod(k)) + \sum_{i=1}^{n-1}(n-i)u_i}$$

In this formula, we have many components:

- On the right side of the formula, the n represents the number of nodes
- Where we have (c-l_0), the c represents the reference state - this value is defined as the next missing value of l0 or the smallest number that does not occur. l_0 is the current value of the state of node 0.
- It is necessary and important to have the mod k - where k is a value chosen by us and needs to be at least equal to n - because we are working with the ring, so reaching the last value, the next is going to be in the beginning
- We have a sum on the left side of the formula. This is determined by the node's location: the last node has the value 1, which increases by one until the node in position zero (this value is assigned from the end to the beginning). This number is then multiplied by zero or one (value of u_i), depending on whether or not the node in question contains a token - it is tested to see if $l_{i-1} = l_i$.

An example will now be shown in practice in order to better grasp how this formula works, with the example from Figure 15:

```
Considering that k = 8, the convergence state has the value 7.
(note: The states with *, means that have a token)

 6 3* 4* 6* 3* 5*   Value = 6 x (7 - 6) mod 8 + (5 + 4 + 3 + 2 + 1) = 6 + 15
                    = 21
(6  5  4  3  2  1)  where n = 6, c = 7, state of node 0 = 6 and elements of the
                    sum have value when it has token


 6 3* 4* 6* 6 5*    Value = 6 x (7 - 6) mod 8 + (5 + 4 + 3 + 0 + 1) = 6 + 13
                    = 19
                    where n = 6, c = 7, state of node 0 = 6 and elements of the
                    sum have value when it has token



 6 3* 4* 6* 6 6     Value = 6 x (7 - 6) mod 8 + (5 + 4 + 3 + 0 + 0) = 6 + 12
                    = 17
                    where n = 6, c = 7, state of node 0 = 6 and elements of the
                    sum have value when it has token



 7 3* 4* 6* 6 6     Value = 6 x (7 - 7) mod 8 + (5 + 4 + 3 + 0 + 0) = 0 + 12
                    = 12
                    where n = 6, c = 7, state of node 0 = 6 and elements of the
                    sum have value when it has token


                    (...)


 7* 7 7 7 7 7       Value = 6 x (7 - 7) mod 8 + (0 + 0 + 0 + 0 + 0) = 0 + 0
                    = 0 -> we reach the value 0 because the system converged
                    where n = 6, c = 7, state of node 0 = 6 and elements of the
                    sum have value when it has token
```

Listing 4.20: Calculus of the Formula

After understanding how this calculus works and having the formulation defined in a formal way, it was possible to translate it to the Why3 tool like we already showed before - see Listing 4.16.

Pigeonhole Principle

The demonstration of the pigeonhole principle [26] is another crucial feature. We must show that at least one state between 0 and K-1 does not exist in any configuration, i.e., it is not a state of any node. This is guaranteed

by N<K, which is critical since it is the only method to be certain that the convergence state exists - like in the figure 15.

The system might not converge if there was no "free" state (and we wouldn't be able to establish the convState function's full correctness).

The way we prove the pigeonhole principle is very elegant in comparison to the very complicated proofs done in proof assistants. The setUnused function (which uses the recursive setUnused_from function) calculates the set of states that do not occur in a configuration, and we prove two post-conditions:

- The first claims that the result is equal to the set of states that do not occur.
- The other says that the cardinal of this set is always greater than one. This is enough no additional lemma is required, because these post-conditions already imply that there is a free state.

```
let rec ghost setUnused_from (w :world) (i:node) : fset node
    requires { forall n :int. validNd n -> 0 <= w n < k_states }
    requires { 0 <= i <= n_nodes }
    ensures  { forall st :state. mem st result <-> 0<=st<k_states /\ noOcc_from w
         st i }
    ensures  { cardinal result >= k_states - (n_nodes-i) > 0 }
    variant  { n_nodes-i }
  = if i=n_nodes then interval 0 k_states
    else let r = setUnused_from w (i+1) in
           if mem (w i) r then diff r (singleton (w i))
     else r

  let ghost function setUnused (w :world) : fset node
    requires { forall n :int. validNd n -> 0 <= w n < k_states }
    ensures  { forall st :state. mem st result <-> 0<=st<k_states /\ noOcc w st }
    (*Proof of Pigeonhole Priciple with this pos-condition*)
    ensures  { cardinal result > 0 }
  = setUnused_from w 0
```

Listing 4.21: Convergence Property - Pigeonhole Principle Proof

The trick to prove the pigeonhole principle is not to include the existential quantification in the result to be proved by induction, because it would be necessary to express "there exist at least x unused states". If the statement uses a single existential quantifier, the single witness may be consumed in the inductive step.

So instead we construct the set of unused states (see Listing 4.21) and prove a lower bound on its cardinality. This entails that when the entire ring is considered there is at least one unused state since k_states > n_nodes, which we formulate as a lemma function.

The contract of the setUnused function will be used as a lemma in the Pigeonhole principle to prove convergence, and in particular to prove that the convState function is well defined, i.e. that the convergence state always exists (the convState_hlpr only ends because a state that does not occur in the ring is guaranteed to exist).

Results of Convergence Property Proof with the Why3 Tool

In conclusion, after having all the necessary lemmas and definitions, it was possible to achieve the Convergence Property Proof with the Why3 Tool of the Stabilizing Mutual Exclusion on an Unidirectional Ring Algorithm. We can see its results in the following image:

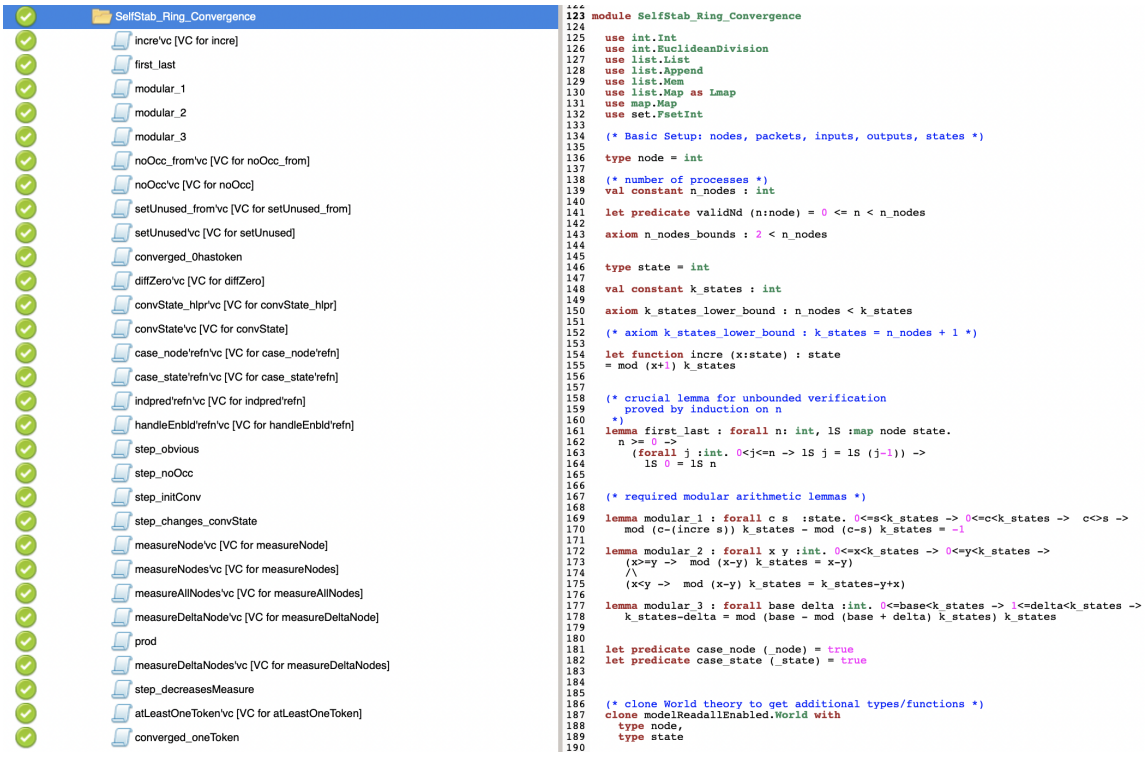

Figure 16: Verification Conditions and transformations proved in Why3 Tool - Convergence Property

5

CONCLUSION

As we have mentioned in the begining of the dissertation, many different program verification tools are currently available. However, they are mostly limited to the verification of sequential code.

In this dissertation, we had as our main goal the verification of distributed algorithms because this is a challenging area that is growing in importance, and is at present an important target for the program verification community. We truly believe that with the work that was developed and presented in this dissertation, we have made a relevant contribution to this community. We were able to attain our main goal and verify one algorithm – Dijkstra's Algorithm for Stabilizing Mutual Exclusion on an Unidirectional Ring – using deductive verification, a technique that offers the highest degree of assurance. As far as we are aware this is was the first verification of this algorithm based on the use of SMT solvers, with a development that is substantially more compact than previous approaches based on proof assistants.

This dissertation was really challenging but in the end we feel that we have accomplished what we were proposed to do. We explored the Why3 deductive verification tool, and contributed to the development of a dedicated Why3 library. Another goal that we had, and again we believe that we have achieved it, was to provide evidence that Why3 is a privileged tool for this task, that stands at a sweet spot regarding expressive power and practicality.

For a future work, we want this subject to keep being explored, and that this library will keep being extended with new models and examples, contributing towards a mature framework for the verification of protocols and algorithms.

BIBLIOGRAPHY

[1] Mark Utting, Bruno Legeard, Fabrice Bouquet, Elizabeta Fourneret, Fabien Peureux, and Alexandre Vernotte. Recent advances in model-based testing. *Advances in computers*, 101:53–120, 2016.

[2] Hung-Pin Charles Wen, Li-C Wang, and Kwang-Ting Tim Cheng. Functional verification. *Electronic Design Automation*, pages 513–573, 2009.

[3] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. The why3 platform. Number 1. 2010–2020 University Paris-Saclay, CNRS, Inria, 2020.

[5] Doron A Peled, Patrizio Pelliccione, and Paola Spoletini. Model checking., 2008.

[6] Leslie Lamport. *Specifying systems*, volume 388. Addison-Wesley Boston, 2002.

[7] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.

[8] Jean-Christophe Filliâtre. Deductive software verification. International Journal on Software Tools for Technology Transfer, 2011.

[9] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[10] Reiner Hähnle and Marieke Huisman. Deductive software verification: from pen-and-paper proofs to industrial tools. In *Computing and Software Science*, pages 345–373. Springer, 2019.

[11] Prime I.T. The euclidean division.

[12] Cláudio Belo Lourenço, Si-Mohamed Lamraoui, Shin Nakajima, and Jorge Sousa Pinto. Studying verification conditions for imperative programs. *Electronic Communications of the EASST*, 72, 2015.

[13] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

[14] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[15] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 614–630, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.

[17] Min-You Wu and Daniel D Gajski. Hypertool: A programming aid for message-passing systems. *IEEE transactions on parallel and distributed systems*, 1(3):330–343, 1990.

[18] Jingshu Chen, Fuad Abujarad, and Sandeep Kulkarni. Towards scalable model checking of self-stabilizing programs. *Journal of Parallel and Distributed Computing*, 73(4):400–410, 2013.

[19] James Aspnes. Notes on theory of distributed systems. CPSC 465/565: Spring 2019, May 2019.

[20] Shaz Qadeer and Natarajan Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In *Programming Concepts and Methods PROCOMET'98*, pages 424–443. Springer, 1998.

[21] Stephan Merz. On the verification of a self-stabilizing algorithm. 1998.

[22] Aly M Farahat. *Automated design of self-stabilization*. Michigan Technological University, 2012.

[23] EW Dijkstra. ªself-stabilizing systems in spite of distributed control, º comm, 1974.

[24] Shlomi Dolev and Ted Herman. Dijkstra's self-stabilizing algorithm in unsupportive environments. In *International Workshop on Self-Stabilizing Systems*, pages 67–81. Springer, 2001.

[25] Lakshmi Visvanathan. *Self-stabilizing sorting on linear networks*. University of Nevada, Las Vegas, 2006.

[26] Miklós Ajtai. The complexity of the pigeonhole principle. *Combinatorica*, 14(4):417–433, 1994.

A

APPENDIX

In this appendix is presented all the code used in this project.

A.1 EUCLIDEAN DIVISION

```
module Division

  use int.Int
  use ref.Ref

  let division (a:int) (b:int) : int
    requires { 0 <= a && 0 < b }
    ensures  { exists r: int. result * b + r = a && 0 <= r < b }
  =
    let q = ref 0 in
    let r = ref a in
    while !r >= b do
      invariant { !q * b + !r = a && 0 <= !r }
      variant   { !r }
      q := !q+1;
      r := !r-b
    done;
    !q

end
```

A.2 FIBONACCI ALGORITHM

```
theory 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 }
    ensures { 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

end

theory 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
| others : 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 }
      invariant { !x >= 0 /\ forall r. fibpred !n r <-> r = !x }
      invariant { !y >= 0 /\ forall r. 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
```

A.3 MODELMP

```
(** {1 World Theory}*)

module World
  use int.Int
  use map.Map
  use list.List
  use list.Append
  use list.Mem
  use list.Map as Lmap

  (* types to be defined at the system level,
     to be instantiated when cloning *)
  type node
  type state
  type msg

  (* product types and their selectors *)
  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


(** {2 Network Theory}*)

module Steps
  use int.Int
  use map.Map
  use list.List
```

```
use list.Mem
use list.Append
use list.Map as Lmap

(* types to be defined at system level, to be instantiated when cloning *)
type node
type state
type msg

(* case analysis predicates are always true *)
val predicate case_node (node)
  ensures { result }

val predicate case_state (state)
  ensures { result }

val predicate case_msg (msg)
  ensures { result }

(* product types *)
type packet = (node, node, msg)
type world = (map node state, list packet)

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

function localState (w:world) : map node state
= let (lS,_) = w in lS
function inFlightMsgs (w:world) : list packet
= let (_,ifM) = w in ifM

(* equality of packets *)
predicate eq_pckt (p:packet) (q:packet) =
  let (dp,sp,mp) = p in
  let (dq,sq,mq) = q in
  dp=dq /\ sp=sq /\ mp=mq

(* well-formedness predicates for packets and node states,
   to be instantiated when cloning *)
predicate ok_NodeState node state
predicate ok_Msg node node msg

(* components of the initial world, defined for each system *)
```

```
val function initState node : state
val constant initMsgs : list packet

constant initWorld : world = (initState, initMsgs)

(* candidate inductive predicate
   contract ensures well-formedness of message and node states,
   and initial world satifies the predicate *)
val ghost predicate indpred (w:world)
  ensures { w = initWorld -> result }
  ensures { result -> forall n :node. ok_NodeState n (localState w n) }
  ensures { result -> forall p: packet.
              mem p (inFlightMsgs w) ->
                ok_Msg (dest p) (src p) (payload p) }

(* helper function for step_message *)
let rec ghost function remove_one (x:packet) (l:list packet) : list packet
  ensures { forall y :packet. mem y result -> mem y l }
= match l with
  | Nil -> Nil
  | Cons h t -> if (eq_pckt x h) then t
                else Cons h (remove_one x t)
  end

(* function specifying how the outputs of the handlers are used
   both in the handler's contract and in the transition relation *)
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_one p (inFlightMsgs w)) in
  (localState, inFlightMsgs)

(* message handler to be executed at node h with state s.
   well-formedness of packet and the state s of h are required; case
   analysis predicates are introduced in the context and
   preservation of the inductive predicate is enforeced in every
   world where handler is executed *)
val function handleMsg (h:node) (src:node) (m:msg) (s:state)
  : (state, list packet)
  requires { ok_NodeState h s }
  requires { ok_Msg h src m }
  requires { case_node h }
  requires { case_node src }
  requires { case_msg m }
  requires { case_state s }
  ensures  { forall w :world. indpred w ->
               mem (h, src, m) (inFlightMsgs w) ->
```

```
                    s = localState w h ->
                      indpred (step_message w (h, src, m) result) }


  (* Transition semantics, in the form of an inductive invariant *)
  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 predicate is preserved by world transitions as
     described by the semantics.  *)
  lemma indpred_step :
    forall w w' :world. step w w' -> indpred w -> indpred w'

  (* many steps relation [reflexive transitive closure]
     and reachable worlds *)
  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 predicate is preserved by many steps and holds
     in reachable worlds *)
  lemma indpred_manySteps :
    forall w w' :world. step_TR w w' -> indpred w -> indpred w'

  lemma indpred_reachable :
    forall w :world. reachable w -> indpred w
end
```

A.4 LEADER ELECTION ON A RING

```
(** {Leader Election}*)

module LeaderElectRing

  use int.Int
  use int.EuclideanDivision
```

```
use list.List
use list.Append
use list.Mem
use list.NthNoOpt
use list.Length
use map.Map
use ref.Ref

(* Basic Setup: nodes, packets, inputs, outputs, states *)
type node = int

(* number of processes *)
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
(* nodes are integers, used for addressing and for constructing
   network topologies, and they have ids that are also integers
   given by the following function *)

val function id node : id

axiom uniqueIds : forall i j :node. id i = id j <-> i=j

(* returns the node between 0 and n-1 with highest id *)
let rec function maxId_fn (n:int) : id
  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

(* Every network has a unique maximum-id node *)
constant maxId_global : id = maxId_fn n_nodes

(* state of a node is a Boolean, true when node claims to be leader *)
type state = { leader : bool }

(* all node states are well-formed *)
predicate ok_NodeState node state = true

(* exchanged messages are node ids *)
```

```
type msg = id

(* well-formed messages in the ring topology *)
predicate ok_Msg (dest:node) (src:node) msg =
  0 <= dest < n_nodes /\ 0 <= src < n_nodes /\ dest = next src

(* case analysis predicates *)
let predicate case_node (_node) = true
let predicate case_state (_state) = true
let predicate case_msg (_msg) = true

(* clone World theory to get additional types/functions *)
clone modelMP.World with
  type node,
  type state,
  type msg



(* Aux defs. to express invariant and handler contracts *)

(* captures the following fact: messages sent from lo to hi will
   necessarily pass through i *)
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

(* System initialization: node states and messages      *)
let function initState _node : state = { leader = false }

(* Variant required -- cannot be defined as function *)
let rec function initMsgs_fn (n:node) : list packet
  requires { 0 <= n <= n_nodes }
  variant  { n_nodes-n }
  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) }
= 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

(* Message  handling function *)
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)

(* Definition of candidate invariant predicate *)
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)



(* Cloning the Steps module will generate VCs to ensure that indpred
   is an inductive invariant *)

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



(* SYSTEM PROPERTIES TO BE PROVED FROM INVARIANT *)
```

```
  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

end
```

A.5 MODELREADALLENABLE

```
(** {1 World Theory}*)

module World
  use int.Int
  use map.Map
  use list.List
  use list.Append
  use list.Mem
  use list.Map as Lmap

  (* types to be defined at the system level,
     to be instantiated when cloning *)
  type node
  type state

  (* product types and their selectors *)
  type world =  map node state

end



(** {2 Network Theory}*)

module Steps
  use int.Int
  use map.Map
  use list.List
  use list.Mem
  use list.Append
  use list.Map as Lmap

  (* types to be defined at system level, to be instantiated when cloning *)
```

```
type node
val predicate validNd (n:node)
type state


(* case analysis predicates are always true *)
val predicate case_node (node)
  ensures { result }
val predicate case_state (state)
  ensures { result }

(* product types *)
type world = map node state

(* components of the initial world, defined for each system *)
val function initState (node) : state

constant initWorld : world = initState

(* candidate inductive predicate
   contract ensures initial world satifies the predicate *)
val ghost predicate indpred (w:world)
  ensures { w=initWorld -> result }


(* handleEnbld-enabling predicate, to be instantiated when cloning *)
val ghost predicate enabled (map node state) (i:node)
  requires { validNd i }


(* function specifying how the outputs of the handlers are used
   both in the handler's contract and in the transition relation *)
function step_enbld (w:world) (n:node) (st:state) : world =
  set w n st

(* handler to be executed at node h when it is enabled
   in the world lS. Preservation of the inductive predicate
   is enforced *)
val function handleEnbld (h:node) (lS:map node state) : state
  requires { validNd h }
  requires { enabled lS h}
  requires { indpred lS }
  requires { case_node h }
  ensures  { indpred (step_enbld lS h result) }
```

```
  (* Transition semantics, in the form of an inductive invariant *)
  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))







  (* inductive predicate is preserved by world transitions as
     described by the semantics. *)

  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


  (* many steps relation [reflexive transitive closure]
     and reachable worlds *)
  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 noNegative_step_TR : forall w w' :world, steps :int.
    step_TR w w' steps -> steps >= 0

  predicate reachable (w:world) = exists steps :int. step_TR initWorld w steps

  (* inductive predicate is preserved by many steps and holds
     in reachable worlds *)

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

  lemma indpred_reachable :
    forall w :world. reachable w -> indpred w

end
```

A.6 STABILIZING MUTUAL EXCLUSION ON AN UNIDIRECTIONAL RING

```
(** {Selfstab Ring}*)


module SelfStab_Ring_Closure

  use int.Int
  use int.EuclideanDivision
  use list.List
  use list.Append
  use list.Mem
  use list.Map as Lmap
  use map.Map

  (* Basic Setup: nodes, packets, inputs, outputs, states *)

  type node = int

  (* number of processes *)
  val constant n_nodes : int

  let predicate validNd (n:node) = 0 <= n < n_nodes

  axiom n_nodes_bounds : 2 < 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

  let predicate case_node (_node) = true
  let predicate case_state (_state) = true

  (* clone World theory to get additional types/functions *)
  clone modelReadallEnabled.World with
    type node,
    type state


  (* System initialization: node states and messages      *)
  let function initState (n:node) : state
```

```
= if n=n_nodes-1 then 1 else 0

(* defining when a node in the ring has the token *)
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))

(* enabling predicate *)
let ghost predicate enabled (lS:map node state) (i:node)
= has_token lS i

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

(* helper definitions for invariant predicate *)
let rec ghost predicate atLeastOneToken (lS:map node state) (n:int)
  requires { 0 <= n <= n_nodes }
  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))

val ghost predicate atMostOneToken (lS:map node state) (n:int)
  requires { 0 <= n <= n_nodes }
  ensures { result <-> forall i j :int. 0<=i<n -> 0<=j<n -> has_token lS i ->
      has_token lS j -> i=j }


(* crucial lemma to achieve an unbounded proof *)
(* of the atLeastOneTokenLm lemma              *)
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



(* candidate invariant predicate *)
predicate inv (lS:map node state) =
  (forall n :int. 0 <= n < n_nodes -> 0 <= lS n < k_states)
  /\
  atMostOneToken lS n_nodes
  (* /\ *)
```

```
    (* atLeastOneToken lS n_nodes *)

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


  (* Cloning the Steps module will generate VCs to ensure that indpred is an
     inductive invariant *)
  clone modelReadallEnabled.Steps with
    type node,
    type state,
    val validNd,
    val case_node,
    val case_state,
    val initState,
    val indpred,
    val enabled,
    val handleEnbld

  (* SYSTEM PROPERTIES TO BE PROVED FROM INVARIANT *)
  predicate oneToken (w:world) = atMostOneToken w n_nodes /\ atLeastOneToken w
      n_nodes

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

end




module SelfStab_Ring_Convergence

  use int.Int
  use int.EuclideanDivision
  use list.List
  use list.Append
  use list.Mem
  use list.Map as Lmap
  use map.Map
  use set.FsetInt

  (* Basic Setup: nodes, packets, inputs, outputs, states *)

  type node = int

  (* number of processes *)
```

```
val constant n_nodes : int

let predicate validNd (n:node) = 0 <= n < n_nodes

axiom n_nodes_bounds : 2 < n_nodes


type state = int

val constant k_states : int

axiom k_states_lower_bound : n_nodes < k_states

(* axiom k_states_lower_bound : k_states = n_nodes + 1 *)

let function incre (x:state) : state
= mod (x+1) k_states


(* crucial lemma for unbounded verification
   proved by induction on n
 *)
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


(* required modular arithmetic lemmas *)

lemma modular_1 : forall c s  :state. 0<=s<k_states -> 0<=c<k_states ->  c<>s
    ->
   mod (c-(incre s)) k_states - mod (c-s) k_states = -1

lemma modular_2 : forall x y :int. 0<=x<k_states -> 0<=y<k_states ->
   (x>=y ->  mod (x-y) k_states = x-y)
   /\
   (x<y ->  mod (x-y) k_states = k_states-y+x)

lemma modular_3 : forall base delta :int. 0<=base<k_states -> 1<=delta<k_states
     ->
   k_states-delta = mod (base - mod (base + delta) k_states) k_states


let predicate case_node (_node) = true
let predicate case_state (_state) = true
```

```
(* clone World theory to get additional types/functions *)
clone modelReadallEnabled.World with
  type node,
  type state



(* System initialization: node states and messages      *)
let function initState (n:node) : state
= if n=n_nodes-1 || n=0 then 0
  else n

(* defining when a node in the ring has the token *)
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))




 (* Expressing that there exist no occurrences of a state
    from a given node on
  *)
let rec ghost predicate noOcc_from (w:world) (a:state) (i:node)
  requires { forall n :int. validNd n -> 0 <= w n < k_states }
  requires { 0 <= a < k_states }
  requires { 0 <= i <= n_nodes}
  ensures { result <-> forall j :int. i<=j<n_nodes -> w j <> a }
  variant { n_nodes-i }
= i=n_nodes || (w i <> a && noOcc_from w a (i+1))


let ghost predicate noOcc (w:world) (a:state)
  requires { forall n :int. validNd n -> 0 <= w n < k_states }
  requires { 0 <= a < k_states }
  ensures { result <-> forall j :int. 0<=j<n_nodes -> w j <> a }
= noOcc_from w a 0



(* PIGEONHOLE principle
   we construct the set of unused states and prove a lower
   bound on its cardinality. This entails that when the entire ring
```

```
   is considered there is at least one unused state since k_states > n_nodes,
   which we formulate as a lemma function.
 *)

let rec ghost setUnused_from (w :world) (i:node) : fset node
  requires { forall n :int. validNd n -> 0 <= w n < k_states }
  requires { 0 <= i <= n_nodes }
  ensures  { forall st :state. mem st result <-> 0<=st<k_states /\ noOcc_from w
       st i }
  ensures  { cardinal result >= k_states - (n_nodes-i) > 0 }
  variant  { n_nodes-i }
= if i=n_nodes then interval 0 k_states
  else let r = setUnused_from w (i+1) in
         if mem (w i) r then diff r (singleton (w i))
   else r

let ghost function setUnused (w :world) : fset node
  requires { forall n :int. validNd n -> 0 <= w n < k_states }
  ensures  { forall st :state. mem st result <-> 0<=st<k_states /\ noOcc w st }
  ensures  { cardinal result > 0 }
= setUnused_from w 0


(* It is not required to formulate the lemma; the lemma function will do
 *)
(* lemma pigeonHole_lm : forall w :world. *)
(*   (forall n :node. validNd n -> 0 <= w n < k_states) -> exists st :state.
    0<=st<k_states /\ noOcc w st *)




(* In the target configuration for the convergence phase
   all nodes have the same state, thus (only) node zero holds a token
 *)
predicate converged (w :world) =
  forall i :node. validNd i -> w i = w 0

lemma converged_0hastoken : forall w :world.
  converged w <-> (has_token w 0 /\ forall i :int. 0<i<n_nodes -> not (
      has_token w i))

predicate initConv (w:world) =
  exists j : int. 0<=j<n_nodes /\ (forall k :int. 0<k<=j -> w k = w 0) /\
      noOcc_from w (w 0) (j+1)
```

```
let ghost function diffZero (w :world) (s :state)
  requires { 0 <= w 0 < k_states /\ 0 <= s < k_states }
  ensures  { 0 <= result < k_states }
  ensures  { w 0 <= s < k_states -> result = s - w 0 }
  ensures  { 0 <= s < w 0 -> result = k_states + s - w 0 }
= mod (s - w 0) k_states



let rec ghost function convState_hlpr (w:world) (base:state) (delta:int) :
    state
  requires { forall n :int. validNd n -> 0 <= w n < k_states }
  requires { 0 <= base < k_states /\ 1 <= delta < k_states }
  requires { base + delta <  k_states ->
             (forall s :state. base<=s<base+delta -> not (noOcc w s))
             /\
           exists c :state. (base + delta <= c < k_states \/ 0 <= c < base) /\
               noOcc w c }
  requires { base + delta >= k_states ->
             (forall s :state. base<=s<k_states -> not (noOcc w s))
             /\
             (forall s :state. 0<=s<base+delta-k_states -> not (noOcc w s))
             /\
           exists c :state. base+delta-k_states <= c < base /\ noOcc w c }
  ensures  { base+delta < k_states -> base+delta <= result < k_states \/ 0<=
      result<base }
  ensures  { base+delta < k_states -> base+delta <= result < k_states -> forall
       s :state. base<=s<result -> not (noOcc w s) }
  ensures  { base+delta < k_states -> 0<=result<base -> forall s :state. base<=
      s<k_states \/ 0<=s<result -> not (noOcc w s) }
  ensures  { base+delta >= k_states -> base+delta-k_states <= result < base  }
  ensures  { base+delta >= k_states -> forall s :state. base<=s<k_states -> not
       (noOcc w s) }
  ensures  { base+delta >= k_states -> forall s :state. 0<=s<result -> not (
      noOcc w s) }
  ensures  { noOcc w result }
  variant  { k_states-delta }
  = let s = mod (base + delta) k_states
  in if delta+1 = k_states then s
    else if (noOcc w s) then s
      else convState_hlpr w base (delta+1)



(* The convergence state is the non-occurring state that is closer to w 0
 *)
```

```
let ghost function convState (w:world) : state
  requires { forall n :int. validNd n -> 0 <= w n < k_states }
  ensures  { 0 <= result < k_states }
  ensures  { not (initConv w) -> noOcc w result }
  ensures  { noOcc_from w (w 0) 1 -> result = w 0 }
  ensures  { initConv w <-> result = w 0 }
  ensures  { w 0 <= result < k_states ->  forall s :state. w 0<=s<result -> not
       (noOcc w s) }
  ensures  { 0 <= result < w 0 -> forall s :state. w 0<=s<k_states \/ 0<=s<
      result -> not (noOcc w s) }
  ensures  { forall s :state. 0<=s<k_states -> s<>result -> noOcc w s ->
      diffZero w result < diffZero w s }
= if (initConv w) then (w 0) else convState_hlpr w (w 0) 1



(* enabling predicate for a given node
   ensures that the transition relation being considered here
   indeed terminates when the target legal configuration is reached
 *)
let ghost predicate enabled (lS:map node state) (i:node)
= has_token lS i && not (converged lS)

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



(* candidate invariant predicate *)
predicate inv (lS:map node state) =
  (forall n :int. validNd n -> 0 <= lS n < k_states)

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



(* Cloning the Steps module will generate VCs to ensure that indpred is an
    inductive invariant *)
clone modelReadallEnabled.Steps with
  type node,
  type state,
  val validNd,
  val case_node,
  val case_state,
  val initState,
```

```
  val indpred,
  val enabled,
  val handleEnbld


(* A few obvious facts when node 0 steps:
     surely w is not an initConv config
     the state of node 0 is not the convergence state.
     Thus, the latter may not occur in the configuration.
 *)
lemma step_obvious : forall w w' :world .
  indpred w -> step w 0 w' -> not (initConv w) /\ convState w <> w 0 /\ noOcc w
       (convState w) /\
    w' 0 = incre (w 0) /\ w' 0 = incre (w (n_nodes -1))

lemma step_noOcc : forall w w' :world, k :node, s :state.
  indpred w -> step w k w' -> 0<=s<k_states -> w k <> s -> noOcc w' s -> noOcc
      w s

(* Important lemma to prove the next one.
   Requires use of interactive proof transformations:
   unfolding of initConv predicate; case analysis; existential witnesses
 *)
lemma step_initConv : forall w w' :world, k :node.
  indpred w -> step w k w' -> initConv w -> initConv w'

lemma step_changes_convState : forall w w' :world, k :node.
  indpred w -> step w k w' -> convState w' <> convState w -> noOcc w' (
      convState w)




(* measure of a single node
 *)
let ghost function measureNode (w:world) (i:int) : int
  requires { indpred w }
  requires { validNd i }
  ensures  { result >= 0 }
  ensures  { i=0 -> convState w >= w i -> result = n_nodes * (convState w - w i
      ) }
  ensures  { i=0 -> convState w <  w i -> result = n_nodes * (k_states - w i +
      convState w) }
= let p = if i=0 then diffZero w (convState w)
                 else if has_token w i then 1 else 0
  in (n_nodes-i) * p
```

```
(* measure of nodes 0 to n-1
 *)
let rec ghost function measureNodes (w:world) (n:int) : int
  requires { indpred w }
  requires { 0 <= n <= n_nodes }
  ensures  { result >= 0 }
  ensures  { result = 0 -> forall i :int. 0<=i<n -> measureNode w i = 0 }
  variant  { n }
= if n=0 then 0
  else measureNode w (n-1) + measureNodes w (n-1)


(* measure of entire ring
   This function is a fundamental part of the specification
   PROPERTY 1 - Before convergence, the measure (which is always non-negative)
       has a positive value
 *)
let ghost function measureAllNodes (w:world) : int
  requires { indpred w }
  ensures  { result >= 0 }
  (* ensures  { result = 0 -> converged w } *)
  ensures  { not (converged w) -> result > 0 }   (* equivalent to the previous
      formulation *)
= measureNodes w n_nodes


(* The above definitions are sufficient to express ou final goal
    step_decreasesMeasure below.
   However, rather than write lemmas to allow us to prove the goal, we will
       define a let function
   measureDeltaNodes that calculates the decrement of the measure when a node
       steps, and tie it
   to the above defs with the postcond  { result = measureNodes w' n -
       measureNodes w n }, and write
   lemmas about this function in the form of postconditions.
   measureDeltaNodes in turn uses another let function measureDeltaNode, again
       annotated with
   appropriate postconditions.
*)


(* delta of a single node i, when k is the stepping node
 *)
let rec ghost function measureDeltaNode (w:world) (w':world) (i:int) (k:node) :
     int
```

```
  requires { indpred w }
  requires { validNd i /\ validNd k }
  requires { step w k w' }
  ensures  { i = 0 -> k = 0 -> convState w' = convState w -> result + n_nodes =
       0 }
  ensures  { i = 0 -> k = 0 -> convState w' <> convState w -> result + n_nodes
      * (diffZero w (convState w)) = 0 }
  ensures  { i = 0 -> k > 0 -> result + n_nodes * mod (convState w - convState
      w') k_states = 0 }
  ensures  { i > 0 -> k = i-1 -> i-n_nodes <= result <= n_nodes-i }
  ensures  { i > 0 -> k = i -> result = i-n_nodes }
  ensures  { i > 0 -> k<>i-1 -> k<>i -> result = 0 }
= assert  { 0 <= convState w' < k_states} ;
  measureNode w' i - measureNode w i


 lemma prod : forall x y :int. x > 0 -> y > 0 -> x * y >= x /\ x * y >= y

(* delta of first n nodes, when k is the stepping node
 *)
let rec ghost function measureDeltaNodes (w:world) (w':world) (n:int) (k:node)
    : int
  requires { indpred w }
  requires { 1 <= n <= n_nodes }
  requires { validNd k }
  requires { step w k w' }
  ensures  { result = measureNodes w' n - measureNodes w n }
  ensures  { 0 <= n-1 < k -> result + n_nodes * mod (convState w - convState w'
      ) k_states = 0 }
  ensures  { n>1 -> k = n-1 -> result + n_nodes * mod (convState w - convState
      w') k_states = n-1-n_nodes }
  ensures  { n>0 -> k = 0 -> convState w' <> convState w -> result + n_nodes *
      (diffZero w (convState w)) <= 0 }
  ensures  { n>0 -> k = 0 -> convState w' =  convState w -> result < 0 }
  ensures  { 0 <= k < n-1 -> result < 0 }
  ensures  { n = n_nodes -> result < 0 }
  variant  { n }
= if n=1 then measureDeltaNode w w' 0 k
  else measureDeltaNode w w' (n-1) k + measureDeltaNodes w w' (n-1) k


(* PROPERTY 2 - the measure strictly decreases with every step of the system
 *)
goal step_decreasesMeasure : forall w w' :world, k :node.
  indpred w -> step w k w' -> measureAllNodes w' < measureAllNodes w
```

```
  (* helper definitions for legal configurations *)
  let rec ghost predicate atLeastOneToken (lS:map node state) (n:int)
    requires { 0 <= n <= n_nodes }
    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))

  val ghost predicate atMostOneToken (lS:map node state) (n:int)
    requires { 0 <= n <= n_nodes }
    ensures { result <-> forall i j :int. 0<=i<n -> 0<=j<n -> has_token lS i ->
        has_token lS j -> i=j }

  predicate oneToken (w:world) = atMostOneToken w n_nodes /\ atLeastOneToken w
      n_nodes


  (* PROPERTY 3 - final confuguration of convergence phase is legal
     and first node holds the token
  *)
  goal converged_oneToken : forall w :world. converged w -> oneToken w /\
      has_token w 0


end
```