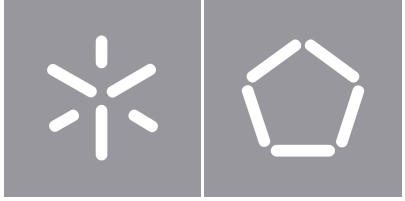
University of Minho
School of Engineering

António Manuel Carvalho Gonçalves

Distributed Systems Verification using Why3

january 2024



University of Minho
School of Engineering

António Manuel Carvalho Gonçalves

Distributed Systems Verification using Why3

Master's Dissertation in Informatics Engineering

Dissertation supervised by
Jorge Miguel Matos Sousa Pinto

january 2024

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Jorge Miguel Sousa Pinto, for all the assistance provided during the development of this thesis. This work would not be possible without his passion for this subject and his attentiveness with my progress.

I would also like to thank my parents, Aníbal Gonçalves and Maria Eulália Gonçalves, for allowing me to follow this path and having enough patience to wait for me to finish my studies. I also would like to thank my brother, Marco Rafael Gonçalves, for all the support he gave me throughout my life.

Finally, I would like to thank my closest friends, who are always there for me, every since the first day I came to Braga to study at the university. Thank you, Pedro Henriques, Gonçalo Quesado, Ricardo Costa, Eduardo Araújo, Pedro Freitas, Gonçalo Esteves, Guilherme Araújo, João Araújo, Daniela Arruda, Rui Oliveira, Tiago Rodrigues, Rita Rosendo, João Fernandes, Eduardo Conceição, Ricardo Machado and Paulo Cruz.

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.

University of Minho, Braga, january 2024

António Manuel Carvalho Gonçalves

Abstract

There has been a greater focus on Distributed Systems lately, creating new ways to approach how information is handled in between systems. As expected, these new methods bring a number of possible problems such as errors that might corrupt the data or even cause critical malfunctions in the program that change the entire information over time. As such, the area of Formal Verification needs to evolve accordingly, in order to keep up with the advancements of Distributed Systems and ensure the functionality and well-being of these systems. The goal of this work is to study the state-of-art of Distributed Systems Formal Verification, specifically on State Machines, as well as the the use of the tool *Why3* in order to model and study a specific type of algorithms used in Distributed Systems called Self-Stabilizing.

Keywords Distributed Systems, State Machines, Formal Verification, *Why3*, Self Stabilizing, Refinement.

Resumo

Recentemente, tem-se observado um maior foco na área dos Sistema Distribuidos, criando novas abordagens sobre como lidar com a informção presente nos sistemas. Como seria de esperar, isto cria também novos problemas, como erros que podem corrumper a data, ou mesmo causar falhas criticas que alteram a informação do sistema ao longo do tempo. Como tal, a área da Verificação Formal precisa também de evoluir no mesmo caminho de forma a conseguir acompanhar estas mudanças, conseguindo assim assegurar o bom funcionamento dos sistemas. O objetivo deste trabalho é estudar o *state-of-art* relacionado com a Verificação Formal de Sistemas Distribuidos, em específico Máquinas de Estado, bem como a utilização da ferramenta *Why3* para modelar e estudar um tipo de algoritmos específico utilizado em Sistemas Distribuidos chamandos Auto-Estabilizantes.

Palavras-chave Sistemas Distribuidos, Máquinas de Estado, Verificação Formal, *Why3*, Auto-Estabilizantes, Refinamento.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This document will start by giving an explanation about the motivation and reason why this subject is currently relevant. After that, we will point out the objectives of this work, as well as how it is related to the current state-of-the-art. Finally, we will present the structure of this document.

1.1 Context and Motivation

The idea of Distributed Systems implies that different systems or components can communicate between themselves in order to fulfil a specific task. As such, the data needed to complete these tasks will, most times, not be in a single component of the system (Dijkstra (1974)). With this in mind, there is a need to assure that, when needed, the data can be moved from one point to another, in the most efficient way and without breaking its own structure or rules. With these, problems are bound to appear if the rules are not always strictly followed. Synchronization errors can lead to different information being given to the users. Besides this, a recently started or rebooted Distributed System might not have the information needed from all the other components. These are only some of the consequences that a badly managed system can cause, which may lead to data loss or even a higher cost of maintenance and performance.

The Formal Verification of Distributed Systems aims to assess if the system achieves its original objectives, while maintaining an over-all state that is not against its own rules throughout the system's life. This way, we can prevent possible failures that could endanger the whole system. To aid the process of Formal Verification, tools like *Why3* can be used, which allow an easier readability and modeling of the systems in order to check intended properties, as well as the creation of new rules to further improve the algorithms used to share data between components, which can then be more easily transcribed onto the actual system implementation.

1.2 Objectives

The main objective of this thesis is the development of a formal verification model for distributed systems using the *Why3* tool, having as main case studies the Self-stabilizing ring system, already studied by Jorge Sousa Pinto, and the Self-stabilizing bidirectional array. We will also present the idea of Refinement and how it can be applied to simplify the process of formal verification of these type of systems.

For that, we will first analyze work related to Distributed Systems, as well as some advances made related to its Formal Verification. These will contain different systems and even other tools besides *Why3*. After that, we will present how both of the systems behave. Following that, we will use the *Why3* tool to study how to model Self-Stabilizing Systems, which are used as a Fault Tolerance technique, using as an example the Self-Stabilizing Ring.

Finally, we will present the concrete work, an implementation of the Self-Stabilizing Bidirectional Array, based on the example of the Self-Stabilizing Ring, using *Why3*. Also, we will introduce the idea of Refinement, and how it can help simplify the proving process of similar systems, by dividing them into parts that can be shared between them, which can reduce the amount of work needed when proving. Finally, we will end this part of the thesis with an implementation of both examples using the Refinement technique.

1.3 Research Hypothesis

With this thesis we expect to show the usefulness of the *Why3* tool, specifically on work done regarding Formal Verification of Distributed Systems and on self-stabilizing systems, as these will be the ones this thesis will study. By modeling and testing this system using *Why3* we will add to an ever-growing area of study, as well as present a different tool that can be useful to other related work. We will also present the idea of refinement and show how it can be used to lighten work that has repetitive code between examples, and allows an easier proof process.

1.4 Document Structure

The chapter "State of the Art" will present all the information needed to begin the study, starting with some examples of how formal verification is being applied to Distributed Systems. After that, there will be a section presenting the *Why3* tool, as well as show a simple example of an algorithm modeled in *Why3* syntax and how the proving process is made. Following that there is a explanation of what a State Machine is and how they work, as well as an introduction of the idea of Fault Tolerance, since the Self-

Stabilizing technique can be seen as one. Following that, there is a section for both self-stabilizing ring and bidirectional array, stating how they are structured, as well as their main properties. Finally, and to end the State-of-the-Art chapter, it will be shown how *Why3* can be used to model the Self-Stabilizing Ring.

The "Formalization of Bidirectional Array" chapter will start by presenting the implementation of the Bidirectional Array using *Why3*, and how each of the previously explained properties can be translated to *Why3* syntax.

Following that, the "Refinement-based Formalization" chapter will explained how the idea of refinement works, followed by both examples of systems presented before written using this technique.

Finally, in the last chapter there will be some conclusion regarding the results reached as well as some difficulties that appeared during this thesis. After that, there are some final ideas of what could be done in the future to enrich this thesis in terms of possible future work.

Chapter 2

State of the Art

This chapter will present the State of the Art related to Distributed Systems and their Formal Verification.

Firstly, it will propose some tools other than *Why3* used in Formal Verification of Distributed Systems. After that it will explain the main functionalities of the *Why3* tool, as well as showing a small example of its uses. Following that, it will explore the idea of State Machines, which are the main focus of this thesis, as well as present some examples of Fault Tolerance techniques, since Self-stabilizing systems have been designed precisely to address this requirement. Finally, there will be a description of both systems used in this work, the Self-Stabilizing Ring and Bidirectional Array, as well as an implementation of the first one using the *Why3* tool.

2.1 Related Work

The area of Distributed System's formal verification is not a new idea, as there are already some examples of works using other tools besides *Why3*. Some examples are the TLA+, Ivy, Ironfleet and Verdi.

TLA+ allows the user to model software using a specific language based on mathematics only. Just like *Why3*, it can work with various backends such as solvers, theorem provers and proof assistants(Kaustuv Chaudhuri (1993)). It is widely used on distributed algorithms in order to check that properties are maintained throughout the system, like invariants and liveness properties, specially on state-machine protocols. By specifying the initial condition (initial state) and the next-state relation (possible steps) it is possible to limit the behaviors of the variables in the system.

Verdi (James R. Wilcox) runs directly on the Coq proof assistant. It allows the user to create models of networks, such as group of nodes that can share messages between them.

IronFleet works based on 3 layers (Chris Hawblitzel). The first specifies the system structure and behavior, the second, the distributed protocol and in the third, the actual implementation of the system. There is communication between the layers in order to secure that the data is not changed throught them.

These layers use a separate tool to connect with Z3 solver, called *Dafny*, which has its own logic language, as a verification tool to check the correctness of the program. This tool is very similar to *Why3*.

Finally, Ivy is also regularly used to work with distributed systems. It can be used to model the protocol as well as finding invariants, using first order logic. It allows a more hands-on approach since the user can add restrictions in order to reach an unbounded system to aid the creation of the invariant. With this objective in mind, *Ivy* has a more transparent process during the development, since it allows the user to see the models in action, as well as errors that appear in their execution.

All of these can be used to explore the Formal Verification of Distributed Systems, but we will focus on the tool *Why3*, which also has some examples already studied.

2.2 Why3

Why3 (Andrei Paskevich (1993)) is a logic tool used to prove certain properties of systems modeled with its own language, using deductive predicates, polymorphic types, recursive definitions, algebraic data types and inductive predicates. It is possible to translate most structures used in other programming languages to *Why3* syntax, which allows us to model the intended system completely using this tool.

Furthermore, it can also use external satisfiability modulo theories Solvers to determine whether or not a formula is satisfiable, or in other words, if there is a way in the program for the given property to be true. However, *Why3* is used as an intermediary to use these external solvers, so it can not prove anything that the solvers themselves could not. Multiple solvers can be used simultaneously, even if they have a different syntax, since *Why3* can translate its own language to the one used by the different solvers. For this, each solver needs a configuration file where the transformations needed to successfully translate *Why3*'s language to the solver's are specified. This file is called a driver, and each Solver available to use in *Why3* needs one.

Why3 comes with three main components: the *why3-ide*, a command line interface, and *why3bench*. The command line interface allows the user to use a given solver to prove properties in a file directly from the command line. The *why3-ide* can be used to open a graphical interface, which is simpler and more user-friendly. With it, the user can choose which solver to use, edit the file with the goals, and directly apply transformations to the proof tasks. It is possible for some provers to easily reach a result while others can not. If a prover cannot verify a goal, *Why3* will tell the user that the prover timed out or that the goal could not be proven. Besides, *Why3* allows the splitting of goals, automatically identifying pre and post-conditions. The *why3bench* is used to compare different results between solvers, or even between

different configurations of the same problem.

A small example of what can be done using this tool's syntax is presented here (Filliâtre).

```
    let function max_sum (a: array int) (n: int) : (sum: int, max: int)
        requires { n = length a }
        requires { forall i. 0 <= i < n -> a[i] >= 0 }
        ensures  { sum <= n * max }
    = let ref sum = 0 in
        let ref max = 0 in
        for i = 0 to n - 1 do
            invariant { sum <= i * max }
            if max < a[i] then max <- a[i];
            sum <- sum + a[i]
        done;
    sum, max
end
```

Given an array, the program will return a pair containing the sum and the higher number. The inputs are specified, as well as their types (an array *a* of ints and an int n), and the same happens to the output (a pair of ints (*sum*,*max*)). The pre and post-conditions appear right after that. The pre-conditions are defined using the keyword `requires`, which in this case limit the values that the program can accept to work. So the first one requires the value of `n` to be the length of the array, and the second says that in all indexes of the array, from index 0 to `n-1`, there needs to be a positive number or 0. If these pre-conditions are true then the program is allowed to start, changing the values needed to reach the result. After that, the post-condition, which use the keyword `ensure`, will be tested on the results. In this case, *Why3* will check if the sum of the array is smaller or equal than the length of the array times the maximum value found, since *sum* can either be equal if all the values in the array are the same, or smaller if there are other values. With these, it is possible to create a *lemma* to test this function.

```
  lemma max_sum_ex :
    forall a:array int, n:int, sum:int, max:int, i:int.
      (0 <= i < n) /\ 0 < a[i] ->
        n < 0 ->
          (sum,max) = max_sum a n ->
            0 <= sum /\ sum <= n * max
```

With the *lemma* above, we are trying to check if for any array with at least one element and with only positive values, if the pair (*sum*,*max*) is the result of applying the *max_sum* function to the array, then

sum should be equal or higher than 0, and *sum* must always be equal or lower than the *max* value times the number of elements of the array. Since this example is so simple, any solver is able to assert its correctness. *Why3* also presents how long the proof took to complete, as well as how many steps were needed.

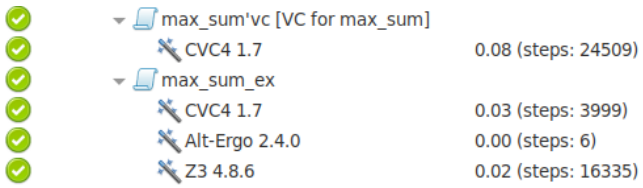


Figure 1: *Why3* ide while proving `max_sum`

The context used by the solver to prove the function *max_sum* is shown next.

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

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

goal max_sum'vc :
  forall a:array int, n:int.
   n = length a /\ (forall i:int. 0 <= i /\ i < n -> a[i] >= 0) ->
   (let o = n - 1 in
    (0 <= (o + 1) ->
     0 <= (0 * 0) /\
     (forall max:int, sum:int.
       (forall i:int.
         (0 <= i /\ i <= o) /\ sum <= (i * max) ->
         (0 <= i /\ i < length a) /\
         (if max < a[i]
          then (0 <= i /\ i < length a) /\
               (forall max1:int.
                 max1 = a[i] ->
                 (0 <= i /\ i < length a) /\
                 (forall sum1:int.
                   sum1 = (sum + a[i]) -> sum1 <= ((i + 1) * max1)))
          else (0 <= i /\ i < length a) /\
               (forall sum1:int.
                 sum1 = (sum + a[i]) -> sum1 <= ((i + 1) * max)))) /\
       (sum <= ((o + 1) * max) -> sum <= (n * max)))) /\
    (0 > (o + 1) -> 0 <= (n * 0)))
```

Figure 2: Context of *Why3* proving *max_sum*

Why3 also allows the application of specific transformation to the proving process, either manually or automatically, which will divide and simplify this context. For example, it is possible to split the proof of the `max_sum` function using the transformation *split_vc*, which separates each individual component of the proof into each own branch. Besides being able to prove more easily, these might also indicate small errors in the construction of the model, since they can show which part is not doing what it is supposed to, for example, which branch of the proof in the `max_sum` is not working.

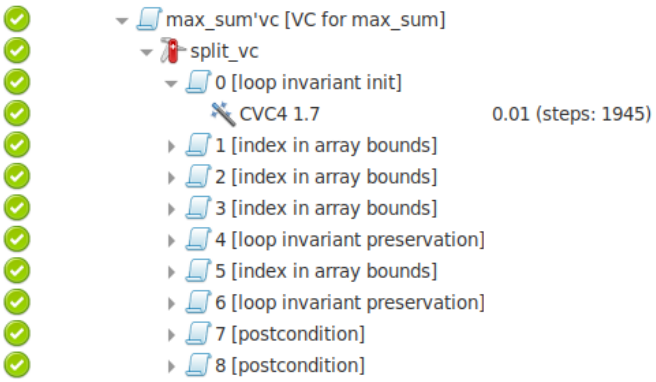


Figure 3: *Why3* ide proving `max_sum` after splitting

For example, lets look at a portion of this split, that represents the seventh branch, which is a verification condition of a post-condition:

```
------------------------- Local Context -------------------------
constant a : array int
constant n : int
Requires1 : n = length a
Requires : forall i:int. 0 <= i /\ i < n -> a[i] >= 0
H : 0 <= ((n - 1) + 1)
constant max : int
constant sum : int
LoopInvariant : sum <= (((n - 1) + 1) * max)
------------------------------ Goal -----------------------------
goal max_sum'vc : sum <= (n * max)
```

Figure 4: Context of *Why3* for the Goal number 7 of the split

Why3 will divide the proof of the function as much as possible, and only add to the context of each proof the information needed to complete it. It is fairly easy to see that this approach allows an easier processes when proving bigger and more complex systems when we compare the local context of the verification condition with and without split.

In this case, the split was made manually because the automatic strategies can prove the function just by using the provers directly, but *Why3* can also apply this and other transformations automatically when needed in other systems that are more complex. The automatic strategies usually start by trying to prove using only the solvers, and if that does not work, they try to simplify the problem by applying transformations.

Besides these, there are a lot more functionalities that can be seen in the Why3 manual, as well as many other examples available in the website Tocatta, created by a French team of researchers.

2.3 State Machine Specification

A typical Distributed System is usually formed by multiple components, each with specific data that can be passed on through connections between one or more of the other parts of the system. These values are the ones that can cause the system to change and evolve to a different state, following the rules of the protocol.

The main focus of this thesis will be State Machines, which in conjunction with Distributed System create states spread across multiple components. The set of states is what is called a world. These will then suffer changes and evolve to other worlds. These systems can be specified by a set of initial worlds (Σ^0), as well as a set of possible worlds (Σ) and a transition relation ($\rightsquigarrow$), such as:

$$\Sigma^0 \subseteq \Sigma \tag{2.1}$$

$$\rightsquigarrow \subseteq \Sigma \times \Sigma \tag{2.2}$$

The set of worlds must contain every initial state given, and the transition relation must transform a world *w* into a different one *w'*. This transition relation can be used multiple times, allowing other worlds to be reached.

$$\forall w \in \Sigma : w \rightsquigarrow^* w \tag{2.3}$$

$$\forall w w' w'' \in \Sigma : w \rightsquigarrow^* w' \implies w' \rightsquigarrow^* w'' \implies w \rightsquigarrow^* w'' \tag{2.4}$$

To this new relation ($\rightsquigarrow^*$) we call *reflexive transitive closure* of ($\rightsquigarrow$). By applying zero transition steps, the world w is reachable from itself (reflexive case). On the other hand, if w can reach w' in a number n1 of steps, and w' can reach w" in n2 steps, then w should reach w" in n1+n2 steps (transitive case).

If for all worlds $w \in \Sigma$ a property (Φ) holds, than it can be called an *invariant* of the system. So, and in conjunction with the previous definition of these systems, it is easy to see that to be an *invariant* the property must follow some rules:

- $\Phi(w_0)$ holds for all $w_0 \in \Sigma^0$;
- $\forall w, w' \in \Sigma$, if Φ holds for w and $w \rightsquigarrow w'$, then Φ should also holds for w', and the same happens for any reachable world.

Functional State Machine Specification

Since this work is going to be focused on the tool *Why3*, some changes are needed. *Why3* is based on Functions, so these state machines need to change accordingly. For that, these characteristics are going to change into action functions and guard predicates. The base of the system is now the tuple F= $\langle \Sigma, \Sigma^0, \{f^i\}, \{P^i\}, N \rangle$ where:

- Σ and Σ^0 are still the same, the sets of worlds and initial worlds;
- N is the number of components in each world, such as nodes;
- $\{f^i : \Theta^i \times \Sigma \to \Sigma\}_{i \in \{1,..,N\}}$ is the family of action functions used to evolve the state, where Θ^i is the given component that is going to evolve;
- $\{P^i\}_{i \in \{1,..,N\}}$ are predicates tested on each node and world, working as guards to take actions or use the functions. With this, the function f^i will only be used if the predicates P^i are valid.

Finally, just like in the normal definition of state machines, there can be a predicate that is maintained through the different states and evolution of the worlds, which is called *invariant*. In order to be considered an *invariant* of the specification tuple F= $\langle \Sigma, \Sigma^0, \{f^i\}, \{P^i\}, N \rangle$, the following conditions must be met:

- $\Phi(w_0)$ holds for all $w_0 \in \Sigma^0$;
- If $P^i(t^i, w)$ and $\Phi(w)$ hold, which means that the guard and the predicate both hold, then $\Phi(f^i(t^i, w))$ for $i \in \{1, .., N\}$, which in turn means that the *invariant* also applies to the world after using the transition function on the node *i*, and the same happens for any reachable world.

2.4 Fault Tolerance

Fault tolerance is the idea that a system should be able to make small mistakes or errors, but still maintain part of its functionalities working. These are specially important on critical systems where even a small error can cause data to be lost or even endanger human lives, like on medical facilities where a simple shutdown that should not happen can lead to loss of human lives.

For this, there are various techniques that allow different grades of security when something goes wrong. For example, simply having copies of the data or backups can be seen as way to secure that if the main data is deleted, either manually, by mistake or by an error that is impossible to predict, it is still possible to recover most of it. A similar technique is to have multiple instances of the same component in a system in order to make sure that, if one of them stops working, at least the other one will still be able to continue as intended, even if with a higher work load. These techniques are specially important in data base where it is extremely important that no data is lost, no matter what happens to the system.

Testing and monitoring the system to find bottlenecks and errors is also a technique that allows the user to find and correct problems before the actual losses happen. Monitoring can identify components of the system that are close to causing problems and need to be replaced or corrected. Testing can simulate the use of the actual systems close to real use scenarios, so it can recognize which parts of the system are most likely to cause problems. Even though this can be seen closer to prevention than to actual fault tolerance, by identifying the possible fails it is possible to implement specific techniques to deal with errors that are observed.

The focus of this work is also on a fault tolerance technique. Self-Stabilizing systems can identify errors on their data and gradually correct them in order to maintain their data structure and availability (Sukumar Ghosh; Schneider (1993)). Besides that, they can also start from a random state, like after a complete shutdown or a reboot, where variables and other data have random values, and slowly turn them into values that are needed so that the behavior of the system does not change or fail. This allows the system to converge into a stable state following a number of steps, that are actions that change a small part of the system. Theoretically, after reaching this state, the system should be able to maintain it throughout its life, as the system does not, in normal circumstances, allow steps that do not follow its rules. If by any chance the system state is changed into an unstable one, it should be able to correct itself again.

Some of the more studied examples of Fault Tolerance Techniques are the *Paxos* (Lamport (2001)) protocol and the simpler *Raft* protocol (Diego Ongaro). These are consensus algorithms used to secure the

good behavior of distributed systems where there is a need to secure that it is possible to make decisions between components, like electing a leader in case of *Raft*, even when taking in consideration possible malfunctions.

Taking a look at the *Raft* algorithm, a set number of processes will need to agree on a leader between themselves, which will then need to make sure that all the followers receive the same actions with the same order. All of these actions can and will cause small errors through the system's life, so multiple fault tolerance techniques can be applied in various places. Having multiple servers that work simultaneously with the same instructions allows the system to most likely always have some servers working correctly. The election of the leader is also made in regular intervals or when the leader stops working. For that, the leader sends *heartbeats* periodically in order to let the other nodes know that there is no need to start a different election. When these are not received after some time, the nodes will start a new election. Also, the leader is elected by communicating with all the components, just like in any other leader election problem. This time measure used to elect the leaders is called *term*, and it is accounted for on each component, so that the system can automatically synchronize the elections, logs stored and other actions made. Besides, each component will keep track of the actions issued by the leaders on their logs, by creating a copy of all client requests that the leader has received. Only components with completed log are allowed to be elected as leaders. If by any reason a follower fails, the requests will be repeated until it restarts and is able to fulfil its work, which does not change how the system behaves overall.

These are just some of the many design aspects used in the *Raft* protocol. All of these allow the system to be more secure and robust, since most possible errors or faults have already been accounted for, and it will be possible for the system to either correct them by itself or just continue working anyway.

2.5 Self Stabilizing Systems

Self-Stabilizing algorithms are part of the fault-tolerance strategies explained previously, and will be the main focus of this thesis. These were brought to attention by Dijkstra (1974), where the system would evolve, from a random state, in order to reach a stable configuration, which is called a "legitimate state". This configuration should not change into an "iligitimate state", no mater how many changes occurred on the system. This state was dependent on relation between nodes and their neighbours.

There are two main properties that can be seen in their behavior, when described as transitions systems: *Convergence* and *Closure*. These allow them, from any configuration, to gradually evolve to reach a specific classification of states, and once reached, secure that they will not change to any incorrect state if the system does not malfunction.

Convergence: These systems are named after the transitions that can occur in their data. These change the overall configuration of the system, and in the case of the self-stabilizing algorithms, approach a stable state where the system will behave as supposed. The Convergence property is the idea that the system will for sure evolve into a correct and stable state, even when starting with a random one. This property is easier to prove with the examples that we will work on, since all the steps allowed in them can only evolve in to a more stable state, or just stay at the same level overall, so eventually, with enough steps, it will reach a stable state.

Closure: After reaching a stable state it is important to maintain it for as long as the system does not encounter unexpected errors, like hardware problems. The Closure property is the idea that once the system reaches a stable state, if it keeps following its rules and no error or miss-input happens, then it will stay in a stable state no matter how many transitions are made. Again, since the steps programmed into these systems do not allow the world to evolve into a "less" stable world, once it is reached, the possible states should all be stable as well.

In this thesis we will present two examples of Self Stabilizing Distributed Systems: the Self Stabilizing Ring and the Self Stabilizing Bidirectional Array. Some of the ideas presented in the first one will be kept in mind for the Stabilizing Array, since it inherits some of its properties.

We will start by defining the systems, followed by how they evolve over time, as well as the rules that need to be followed throughout the protocols' life (Merz (1998); Dijkstra (1974)) for both examples.

2.5.1 Self Stabilizing Ring

As described in Dijkstra (1974), this system is composed by N nodes, and each of them has connections to the previous and the next node. Node 0 has connections to the node N-1 and 1. In this case, we will consider that inside each node there is a value that represents its state that can go from 0 to to any values lower than the total number of nodes. In this work we will consider the higher value possible for the nodes the value N-1. The connections between nodes are used to pass their own value to the next node, and to read from the previous one. To the group of nodes and their respective states we will call Configuration.

Each node can have a token, which allows it to change its state following a set of rules of the system. These rules are based on the values of nodes that the node can read, which are its own and the following one. In a ring with N nodes, node 0 will have a token if its value is equal to the one from the last node (N-1). For any other node, the token is created if the value that it reads is not the same as its own.

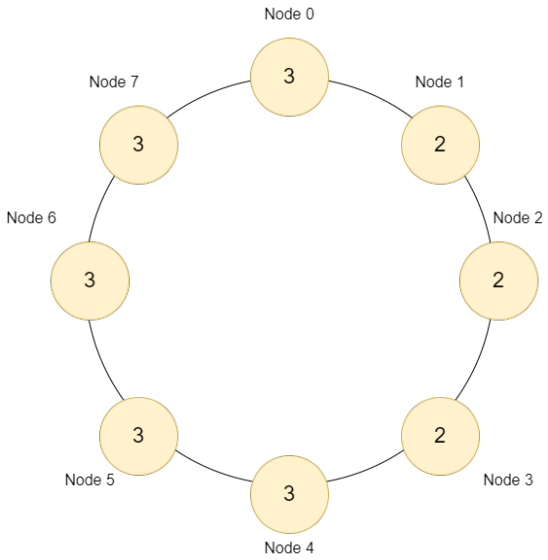


Figure 5: Possible Configuration of a Self Stabilizing Round-Array

In the image above we can see a possible state for a round-array. Nodes 0, 1 and 4 have tokens since node 0 and 7 have the same state, and the other nodes have a different value to the one in the node before them. This could easily be a possible configuration of a newly started system, since all nodes appear to be random.

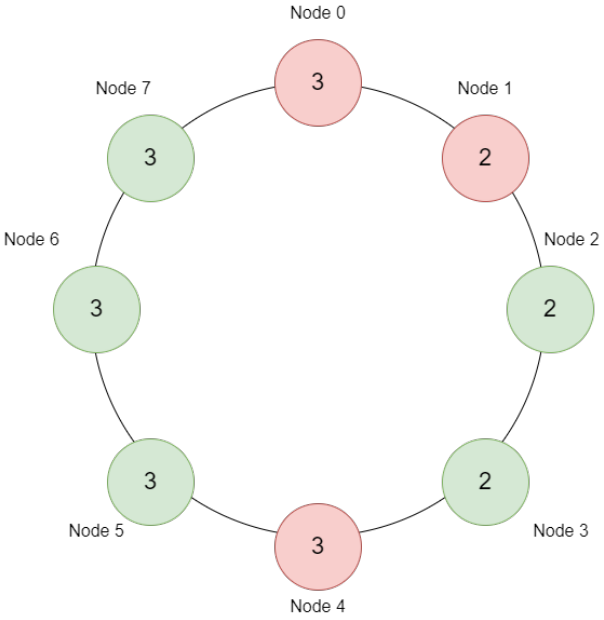


Figure 6: Tokens in the Round Array

In the image we can see the nodes that have a token marked red, and the ones that do not green. Nodes that have a token can take specific actions (steps) in order to evolve to a different state. In this case, the action taken is to increase the present value in the node 0 if it has a token, maintaining it in the range allowed by the system $(0 \geq n \geq N-1)$ by using `n' = mod (value+1) N-1`, or change its value to the one received from the previous node, for any other node.

Only one step is allowed to be taken at the time, which means that only one of the states will change for both examples. The node that is chosen for the next step is random, from the list of nodes with tokens.

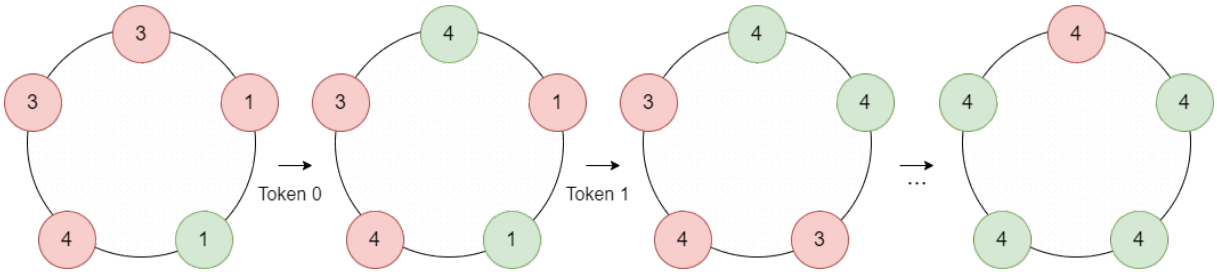


Figure 7: Evolution of the Round-Array Configuration

The objective is that, after a finite number of steps, only one token is active throughout the life of the system. It is easy to see that this is possible since, no matter which step is taken, the number of tokens either decreases or stays the same, which will gradually cause the system to decrease its total number of tokens to a minimum of one. After reaching this state, the number of tokens will not increase as the values of each node will all be the same, triggering the node 0 token rule, increasing its values and creating a

token on node 1, and so on. Once the system reaches this state we can say that it has reached a Stable State, as in theory it should maintain these properties throughout its life.

During the stable configuration it is possible to assert some properties about the states. Since there is only one token, if the node with token is the node 0 then all the nodes in the ring should have the same state, or there would be another token present. If any other node has the only token then there is also a similar conclusion to make. The nodes after it must have a different state than the one with token, but equal between them, until the last node (N-1), and the ones behind, including node 0, must have the same state as the node with token. This way, only one token can exist.

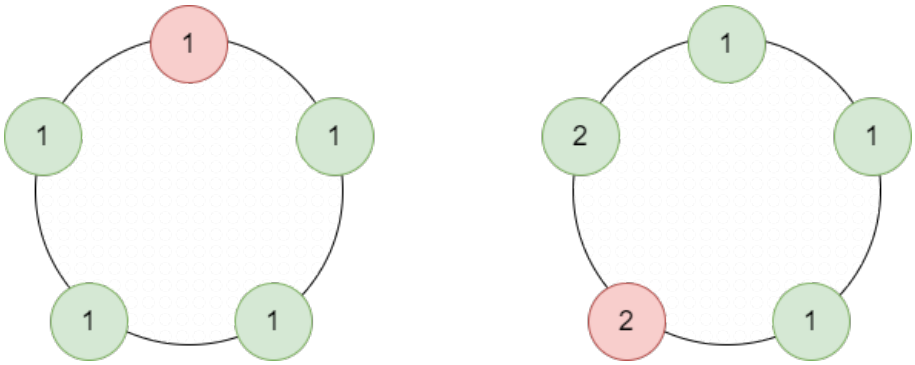


Figure 8: Both cases of Tokens on a Stable Configuration

2.5.2 Self Stabilizing Bidirectional Array

The Self-Stabilizing Bidirectional Array is composed by a number of nodes (N nodes), each with an identifier that goes from 0 to N-1 and also a specific state, represented by an *int*. In this case, this value can be read by the previous and the following node in the array, but there is no connection between the first and last node. To the group of the nodes and states of each one we will call Configuration.

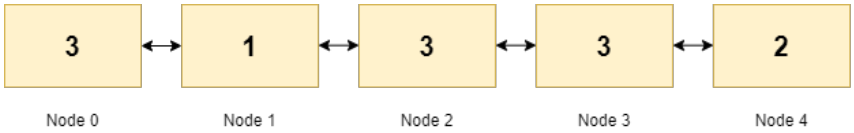


Figure 9: Example of a Bidirectional Array Configuration

The state of each node ranges from 0 to 3, not being limited by the total length of the array. Since this is a Self Stabilizing System, these values will progressively change in order to achieve a stable configuration. As there is no communication between the first and last node, like in the Ring example, the strategy used is slightly different.

So, to reach a stable configuration, the first node can only have the values 1 or 3, and the last 0 or 2.

```
(a[0] = 1 \/ a[0] = 3) /\ (a[N-1] = 0 \/ a[N-1] = 2)
```

With this in mind, each node can change its state when it has a token, that is, if its state satisfies a specific condition when compared with the value of its neighbors. In this example, a node has a token when a neighbor has a value that is the increment of the node's state, which means:

```
( a[n] + 1 ) % 4 = a[n-1] \/ ( a[n] + 1 ) % 4 = a[n+1]
```

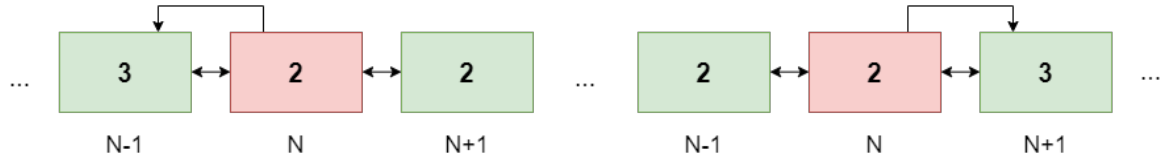


Figure 10: Nodes with token

In the image above, we have two portions of two arrays, with three visible nodes on each (N-1, N and N+1). Looking at the first array, node N has a token because node N-1 has a state of 3. In the second one, node N has token because node N+1 has a state of 3. The `mod` operator allows the system to check only values in the range allowed. Using this, a node with the state 2 will create a token if either the previous or the next node has value 3, and the state 3 will do the same with the value 0, and so on.

As an example, the following array has the extremity nodes with correct values, but some nodes have a token, which are represented by the red boxes. The green boxes do not have a token. The arrows leaving each red box represent the previous formula, showing the comparison of the states on each node and which of its neighbours causes the token to exist:

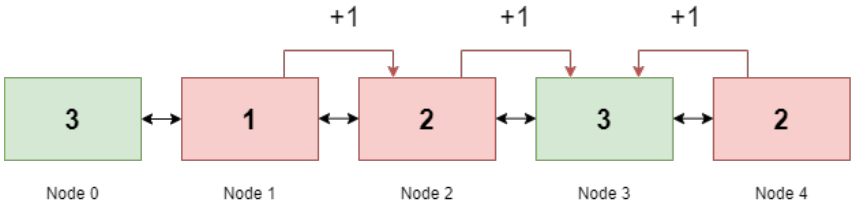


Figure 11: Example of nodes that have a token in an array

Here, each node with a token will change its value in order to get closer to a stable configuration. Each of these changes are a system step. When a node has a token it will increment its value in order to mimic the value that caused it to have a token (s[n]' = (s[n] + 1) % 4). Since the edge nodes can only have

specific values, these will evolve differently from the rest. If node 0 or node N-1 have a token, then their state will increment twice (s[n]' = (s[n] + 2) % 4).

It is possible to identify three types of states for the bidirectional array: the *Base Configuration*, the *Stable Configuration* and the *Complete Stable Configuration*.

Base Configuration

Here all nodes have random values between 0 and 3, with the respective extremities having the correct states. The number of tokens is impossible to predict, as the system is behaving like a recently initialized array, having close to no steps taken. From this state, the system will follow its rules to evolve, reducing the number of existing tokens, turning into the next type of array, the stable array.

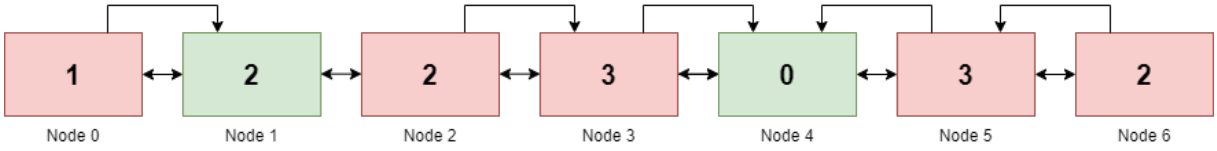


Figure 12: Random Base Configuration

Stable Configuration

A stable array is reached when, from the base one, a number of steps is taken and the result is an array where only one token is active. This does not mean that all values need to be the same. As long as the values to the right and to the left of the token follow a simple rule, then it is not possible to have more than one token with any step taken.

When the node has a token created by the node to its left, then all the states to the left are either its increment or three times the increment. This way no node to the left can create a token between them. In the same way, the nodes to the right can only have the same state or be incremented twice. These rules are clear to see in the image below

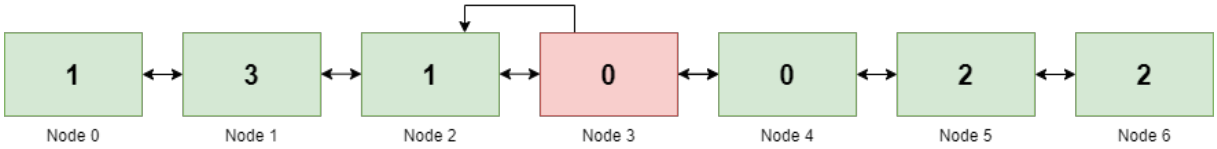


Figure 13: Stable Configuration with token from the Left

When the node has token from the value on the right the relations of the remaining nodes are the

same, but on the opposite side. All the nodes to the left need to be either the same value or incremented twice, and the ones to the right incremented once or three times.

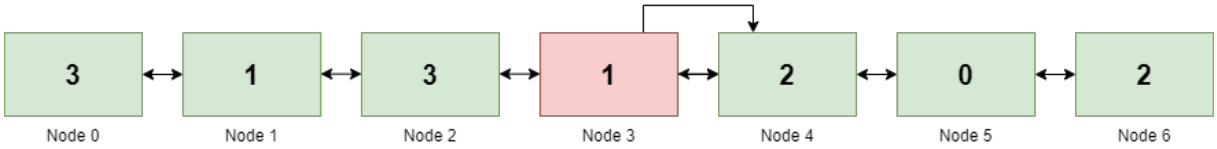


Figure 14: Stable Configuration with token from the Right

Both of these rules also limit the states of nodes with token on these simple stable states, since these also limit the states of the extremity nodes. Let us look at an example of an array where the node n has a value of 1 and a token created by the node to its right. Just as explained before, this means that all the nodes to the right of the node n must be its increment or incremented three times, which are either values 2 or 0. Following this, the values to the left must be either 1 or 3. In this case these values are acceptable values for the extremity nodes, but this is not necessarily the case.

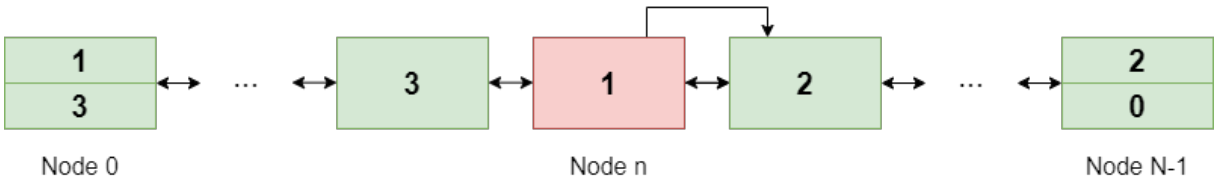


Figure 15: Stable Configuration with valid values on the extremities

The image below shows how these rules can limit the values depending on the token that exists on a stable array. Here, node n with value 0 has a token created by its right neighbour, which limit the values to the right to either 1 or 3, and the ones to the left to 0 or 2. With this, the extremities can only have values that go against its own rules, since node 0 will either have value 1 or 3, and node N-1 either 2 or 0, which is impossible, since these states should not be accepted in this system.

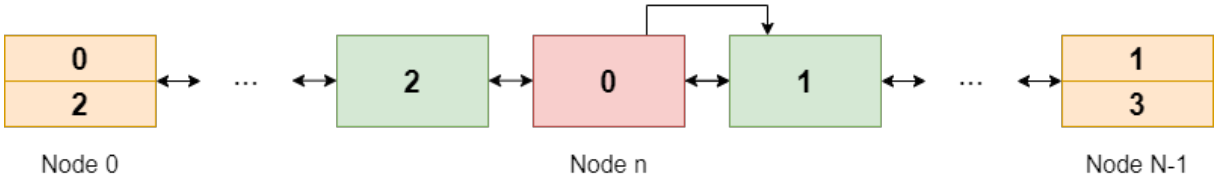


Figure 16: Stable Configuration with invalid values on the extremities

So, and in order to maintain the extremities with the correct values, if in a stable configuration there is a node with token from the left, then the value of that node must always be either 0 or 2. The same

applies to the token from the right, which limit the values of the node with token to either 1 or 3.

Besides this, in any stable configuration, all states to the left and to the right of the token are also limited. Since the left extremity can only take the values 1 or 3, and the left 2 or 0, then in order to not create more than one token all nodes to the left of the existing token must either be 1 or 3, and to the right 2 or 0. This also creates an additional property. Since nodes to the right and left of the token on a stable configuration always have different states, it is impossible for any configuration to create a token from both sides. This means that if a token is created from the right, then it is impossible to also be another one created on the same node from the left.

Complete Stable Configuration

Finally, there is the complete stable configuration. Just like in the previous one, there can only be one token at any time, but all states to the right or to the left of the token are now equal between them.

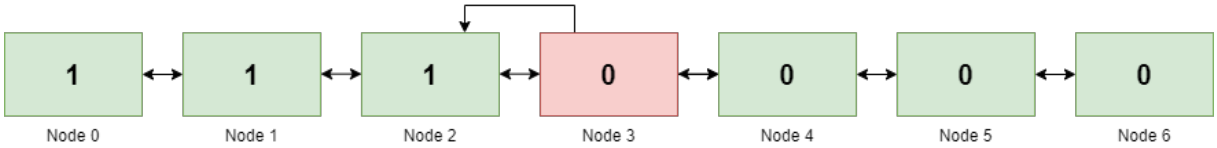


Figure 17: Complete Stable Configuration

Here, each step will create the token in the opposite direction of the node that created it in the first place. So, if the token is created by the node on the left, after taking a step a new token will be created on the node to its right, and the old one will be removed. This changes when the node with token is an extremity node, since there is no additional node to the right of node n-1 or the left of node 0. In these cases, since these increment their values twice, the token will then change to the opposite direction. So, if node 0 has token from the right, the step will create a token from the left on node 1 next.

As an example, the system from the next image will always maintain one token if there are no faults in its behavior. Once the system stabilizes, the one token will travel from edge to edge of the array, since, in this case, both node 0 and 4 always increment twice. This does not happen to the other type of configurations, where each step can cause the token to swap directions, since not all nodes have the same state.

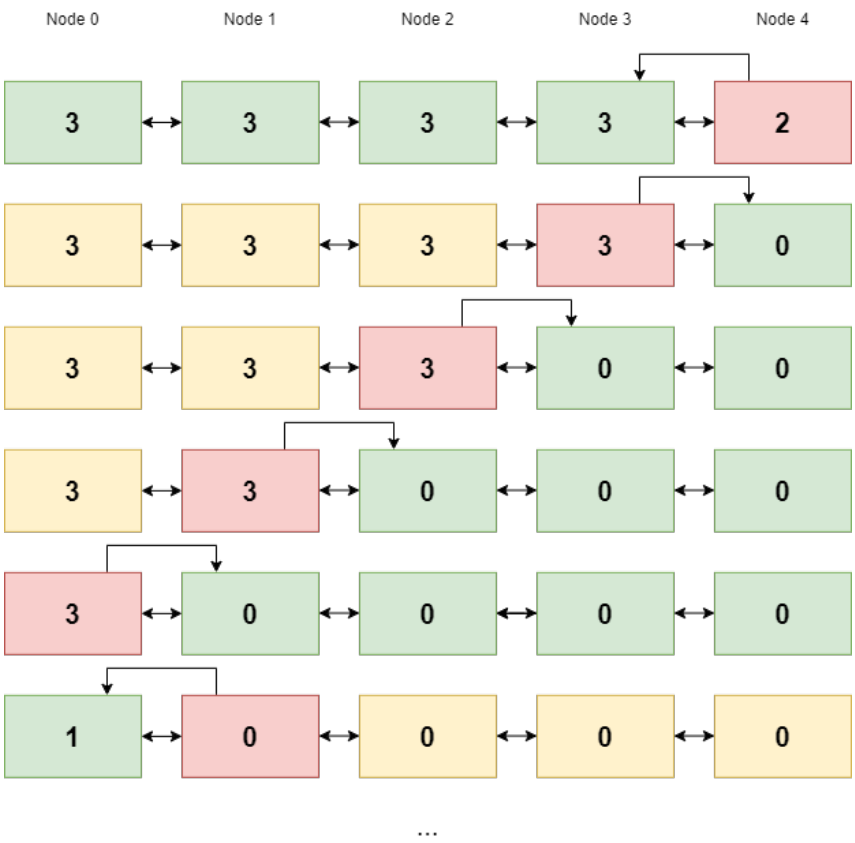


Figure 18: Stable Configuration over the time

Also, all the nodes marked orange will change to the value of the extremity marked with green. When only the opposite extremity has a different state it creates a token to the opposite direction, repeating the wave-like movement of the token through the array.

Even though both the complete and the normal stable configuration are considered stable, the second one will always evolve into the complete. Every step taken on a stable configuration will gradually make the values equal between themselves, to the right and the left of the token, which eventually leads it to a complete stable configuration.

Possible Steps

After describing how this system works, it is possible do identify all the possible steps taken throughout its life, just as they are presented in Ghosh (2014):

Case	World	Step	World'
a)	x+1 <- tx x	x+1	x+1 <- tx
b)	x+1 <- tx -> x+1	x+1	x+1 x+1
c)	x+1 <- tx x+2	x+1	tx+1 -> x+2
d)	x+1 <- tx x+3	→ x+1	x+1 x+3
e)	x tx -> x+1	tx -> x+1	x+1
f)	x+2 tx -> x+1	x+2 <- tx+1	x+1
g)	tx+3 tx -> x+1	x+3	x+1 x+1
h)	tx -> x+1 ...	x+2 <- tx+1	...
i)	... x+1 <-tx	...	tx+1 -> x+2

Table 1: Table with all possible steps

Each case shows the behavior of three nodes, being marked with a "t" if the given node has a token, and an arrow to represent which of his neighbours are creating the token, before and after taking a step.

For example, the property a) can be represented by the following arrays:

Figure 19: Representation of Step a)

Here, the system has somewhere on the array three consecutive nodes, `n-1`, `n` and `n+1`, each with the states `x+1`, `x` and `x` respectively. The nodes on `n-2` and `n+2` can cause tokens on `n-1` and `n+1`, but in these cases the main focus is the token on `n`. After taking a step on node `n`, its value becomes `x+1`, moving its token to the next node.

The cases a) through g) are the ones used when a node is not an extremity node, and h) and j) are specific for those cases. Looking at the table, it is easy to see that a step cannot increase the number of tokens present in the system, only allowing to either maintain or reduce it. Cases a), c), e) and f) do not decrease the number of nodes, but can change the node that has a token (a and e), or the direction from

which the token is created (c and f). Cases b) and d) reduce the number of tokens from one to zero, and g) from two to zero.

Considering more than three nodes, it is possible that a), c), e) and f) could decrease the total number of tokens in the array. For example, if the step is moving the token from n to n+1, just like in property a), but n+1 already has a token from n+2, creating the array on property e) on the nodes n, n+1 and n+2, then the total number of token decreases by one with the step, as represented by the image below:

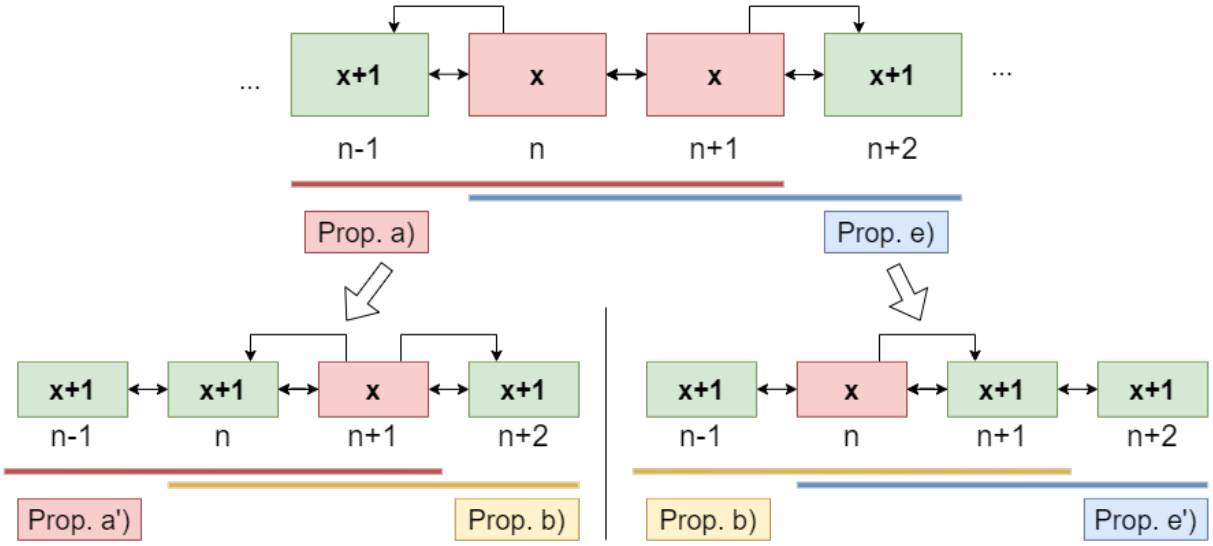


Figure 20: Steps reducing the total number of tokens

Here, a portion of the array has both property a) and e), with two tokens visible. If either of them is updated by a step, the result is an array with one token less, even though the type of steps a) and e) should not reduce the total number of tokens when only looking at node n and its neighbors.

Finally cases h) and i) are used when node 0 and node N-1 have a token, respectively, and transfer the token to the next node if it is the first, or to the previous if it is the last. Just like explained previously, these can also reduce the total number of tokens considering more nodes, since the nodes 2 and N-3 can cause tokens to be created as well.

2.6 Self Stabilizing Ring using Why3

After presenting all the rules for both algorithms and their structure, as well as the tool we will be using in this work, we will now show an example of a system that combines the Self-Stabilizing Ring and *Why3* notation (Jorge Sousa Pinto), in order to prove the properties presented previously. All the code presented can be consulted in Github repository of the project (Git).

As a base, we firstly need the variables that represent the various nodes, as well as the whole system:

```
type node = int
val constant n_nodes : int
axiom n_nodes_bounds : 2 < n_nodes

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

type state = int
val constant k_states : int
axiom k_states_lower_bound : n_nodes < k_states
```

First we declare a new type called `node` which is an int that identifies the number of the node. We also create a new variable, `n_nodes`, that will keep the total number of nodes in the system. After that, we create an *axiom*, a rule that needs to be followed throughout the system, to secure that there are always at least three nodes in the system. This is used so that systems with two nodes are not considered, since it would always have a token with any values. Finally, we need to make sure that if given an int it represents a node in the desired interval, so we create the predicate `validNd`, which checks if the number matches a node inside the system. A new type is created as well, in order to keep the data that is inside each node, called `state`. Besides that, the number of different states cannot be lower than the number of nodes (*k_states*), which is given by definition of the system. So the state of each node is now limited by how many nodes there are.

Finally, the main structure of the system is going to be a map that pairs the number of the node and the state inside it.

```
type world =  map node state
```

We then define how the system decides if a given node has a token or not, as mentioned in a previous section: If node 0 has the same state as node N-1, or if any other has a different state from the previous node.

```
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))
```

Next we need to define the transitions between states in order to make the system evolve.

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

inductive step world node world =
| step_enbld : forall w :world, n :node.
    validNd n ->
      has_token w n ->
        step w n (set w n (if n = 0 then incre (w (n_nodes-1)) else w (n-1)))
```

The function `incre` is going to be used to change the value of the node 0 when it has a token. After that, the predicate `step` is going to be used to simulate the step by giving the present world and the node that is going to change, and returning the new world where the value of the node `n` has been updated. For this we first check if the node is valid and has a token, and then we apply the change, depending on whether it is the node 0 or a different node. This predicate is an *inductive* predicate, which is going to be explained later.

Besides defining how the world evolves we need to be sure that all the other variables are not changed when a step is made. For that we use the following:

```
lemma step_preserves_states :
   forall w w' :world, k i :node. step w k w' -> i<>k -> w i = w' i
```

The *lemma* `step_preserves_states` secures that if a step is made using the node k, the new world should not change the value of any other node besides k.

After that, we have the *inductive* predicate `step_TR`. This is mainly used to check the reflexive transitive closure of these systems, where the step function can be applied multiple times to reach other worlds.

```
inductive step_TR world world int =
| base : forall w :world. step_TR w w 0
| step : forall w w' w'' :world, k :node, steps :int.
         step_TR w w' steps -> step w' k w'' -> step_TR w w'' (steps+1)
```

```
5
6  lemma noNegative_step_TR : forall w w' :world, steps :int.
7    step_TR w w' steps -> steps >= 0
```

So, `step_TR` shows that if a world w can reach a world w' in n steps, and world w' can reach world w" in one, then world w can reach world w" in n+1 steps. Being a *inductive* predicate means that it is based on using multiple constructors, in this case, `base` for when w'=w, and `step` for any other. These are also most of the times used for recursive predicates, such as this one, or when working with data structures. It is also necessary to state that the number of steps to reach a world can never be negative.

Finally, just as stated before, the main property that needs to be checked is that the system must always have at least and at most one token at each given time, and for that we use the predicate `atLeastOneToken`.

```
1  predicate atLeastOneToken (lS:map node state) (n:int) = exists k :int. 0<=k<n /\
     has_token lS k
2
3    val ghost predicate atMostOneToken (lS:map node state) (n:int)
4    requires { 0 <= n <= n_nodes }
5    ensures { result <-> forall i j :int. 0<=i<n -> 0<=j<n -> has_token lS i ->
     has_token lS j -> i=j }
```

The idea that there is always at least one token is just a direct implication of the system, since there is no possible configuration that does not have at least one token.So now it is possible to create the invariant that is going to be used to prove the correctness of this transition system using both `atLeasOneToken` and `atMostOneToken`.

```
1  predicate inv (lS:map node state) =
2    (forall i :int. validNd i -> 0 <= lS i < k_states)
3    /\
4    atLeastOneToken lS n_nodes
5    /\
6    atMostOneToken lS n_nodes
```

This invariant ensures that there are no nodes with incorrect states, and that both of the properties showed previously are maintained. To prove this for any world we use the following:

```
1  lemma inv_full_step_unbounded :
2    forall w w' :world, n :node. step w n w' -> inv w -> inv w'
```

In order to prove this invariant, which is not dependent on the total number of nodes of the ring the creation of an additional *lemma* was necessary, which translates one of the properties of the ring that was stated when describing the system.

```
let rec lemma first_last (n :int) (lS :map node state)
  requires { validNd n }
  ensures { (forall j :int. 0<j<=n -> lS j = lS (j-1)) -> lS 0 = lS n }
  variant { n }
= if n=0 then ()
  else first_last (n-1) lS
```

This *recursive lemma* states that for any ring, if all nodes but node 0 and n-1 have the same state, that means that node 0 and n-1 also have the same value. With all of this, it is possible to prove the protocol with any number of nodes. Before adding this *lemma*, higher number of nodes would normally imply more variables and more time to processes which might cause the provers to time out.

Chapter 3

Formalization of Bidirectional Array

This chapter will present the work done regarding the modeling of the Bidirectional Array using *Why3*. This is going to focus mainly on the Closure property, which states that once the system reaches a stable configuration, no mater the transitions made, it will always stay stable. It should also be possible to do a similar work about the Convergence property, but it would focus more on a variant-like approach, in which the closer the system is to the stable state, the lower that variant would be, reaching the minimum when it reaches a stable configuration.

3.1 Bidirectional Array Implementation

The base structure of the Bidirectional array is very similar to the Ring example. So, with this in mind, the main component is an *int* called `node`, each with a `state` represented by an `int`, organized on a map called `world`, which has pairs of `node state`. The total number of nodes is kept in the constant `n_nodes`, since it will be needed for future queries, such as the predicate `validNd`, that checks if a given node is within the nodes of the array. Besides that, we will limit the number of nodes to at least three, since a lower number would cause the system to only have extremity nodes, not representing the entirety of the system capabilities, even though in a world with two nodes it would still be possible to take steps, but these would only cause the token to jump from one node to the other repeatedly.

```
1 type node = int
2 val constant n_nodes : int
3 axiom at_least_two : 2 < n_nodes
4 let predicate validNd (n:node) = 0 <= n < n_nodes
5 type state = int
6
7 type world = map node state
```

Next, it is necessary to limit the states within the nodes themselves. For that we use an axiom that only allows values between 0 and 3 on the nodes.

```
axiom state_limit: forall w:world, s:state. 0 <= w s <= 3
```

Following that, the system needs to know how the node can check if it has a token. For that we use the function `incr` and the predicate `has_token`.

```
function incr  (x:state) : state = mod (x+1) 4
function incr2 (x:state) : state = incr (incr x)
function incr3 (x:state) : state = incr (incr (incr x))
```

These are going to be used to compare values between nodes, as well as to evolve the world after a step.

To check if a node has a token, we use both `has_token_L` and `has_token_R` to see if any of the neighbors, the left one, which is the previous, or the right one, which is following, have a state that is the increment of the given node. Some later properties became easier to state by separating how the system checks if the token is created by the right or left node. These do not check to the left of node 0 or to the right of node `n_nodes-1` since these are the extremity nodes, so there should be no node there.

```
predicate has_token_L (w:map node state) (n:node) = (n > 0 ∧ n <= n_nodes-1
                                                    ∧ w (n-1) = incr(w n))
predicate has_token_R (w:map node state) (n:node) = (n >= 0 ∧ n < n_nodes-1
                                                    ∧ w (n+1) = incr(w n))
predicate has_token (w:map node state) (n:node) = has_token_L w n
                                                ∨  has_token_R w n
```

It is important to note that at the beginning we were separating the token on the first and last node from these, but by doing this it would create additional conditions to test, so we decided on this approach to make the code more readable. With these, when it is needed to separate the cases, for example, when we need the case of node 0 with token, we just need to use: `has_token_R w 0` and for any other node `has_token_R w n` and `n!=0`. The same happens with the case of token on `n_nodes-1` and `has_token_L`.

Next, we create a function to initialize a world. This is going to be used to check later properties, such as the reachability of a given world starting from this one. This initial world is already in a stable configuration, with only one token on node 0, and since the steps taken cannot add tokens, it should always maintain only one token.

```
let function initState (k:node) : state
```

```
  = if k=0 then 1 else 2

  constant initWorld : world = initState
```

The function `initState` is called on the `initWord`, which works as a guard function to create the *constant* used as the initial world for future predicates.

Next, we have some predicates that are going to be used to check the number of tokens inside the array.

```
let rec ghost predicate noTokens (w:map node state) (n:int)
  requires { 0 <= n <= n_nodes }
  ensures { result <-> forall k :int. 0<=k<n -> not(has_token w k) }
  variant { n }
= n = 0 || (not(has_token w (n-1)) && noTokens w (n-1))
```

First, `noTokens` checks if there are no Tokens inside the given world and interval. There should be no array with no tokens, but this is going to be used as an auxiliary predicate for others. First, the `n` given must be between 0 and `n_nodes`, to check only nodes that are inside the array. The result of this predicate checks, if for all nodes between 0 and `n`, no node in the array has token. With this, and using the given `n` as a variant, the predicate observes if the `n` node does not have token, and calls itself recursively with `n-1`, until it reaches the end of the array, were `n` is equal to 0. This uses the keyword `ghost` to indicate that this predicate is used only to express logical values. In this case, it should be possible to create a predicate that proves that there is at least one token in the array using only logic. Instead, this predicate only applies the logic on the post-condition, and there is an additional recursive code, since it also uses the keyword `rec`, to check the property of the number of tokens in the array. By grouping logic and code, using the `ghost` keyword, we get stronger specification, since the logic post-condition is going to be applied in conjunction with the code. This also allows the user to specify the intended properties in two different ways, and if one is not correct then the other will not work as well.

Next, the predicate `atLeastOneToken` follows the same strategy, but instead of checking if there is no token in the array, it will see if there is at least one token.

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

Finally, the predicate `atMostOneToken` checks if there is only one token in the array.

```
let rec ghost predicate atMostOneToken (w:map node state) (n:int)
  requires { 0 <= n <= n_nodes }
  ensures { result <-> forall i j :int. 0<=i<n -> 0<=j<n -> has_token w i ->
  has_token w j -> i=j }
  variant { n }
= n = 0 || (has_token w (n-1) && noTokens w (n-1))
  || (not(has_token w (n-1)) && atMostOneToken w (n-1))
```

For that, the predicate `noTokens` is used so that it can secure that when a token is found no other is present in the array. As a post-condition, if an array checks the predicate `atMostOneToken`, then if two random nodes are chosen, and both have token, then the two of them must be the same, since there should only be one token in the array.

These three predicates are going to be used to prove the correctness of the system. `atLeastOneToken` can be proven using only the system definition, since any system that has all values within the interval of 0 and 3, as well as the extremity nodes with the correct values will necessarily have at least one token active. On the other hand,`atMostOneToken` is not a direct implication of the system definition, since it is possible that a world has more than one token. So with this, the first one can be proven using a *lemma*, since the information needed to complete the proof is already present, and the second will be used as an invariant to the system, which will be used to prove that the predicate is maintained with each step.

```
lemma atLeastOne : forall w :world, n :int.  2 <= n <= n_nodes
  -> (w 0 = 1  \/  w 0 = 3)  /\ (w (n-1) = 0  \/  w (n-1) = 2)
  -> atLeastOneToken w n
```

In most tested properties it is necessary to secure the values of both node 0 and node `n_nodes-1` are correct. Also, when testing this *lemma* a problem appeared, since *Why3* was trying to prove this property with a base case for n = 0, because it is the generic base case for induction on *ints*. This would always be false, since it should be between 2 and `n_nodes`, so to make it understand that the base case should be 2 it was needed to apply a transformation `induction n from 2` to only verify from 2. This, and some others inductive predicates where the only ones that needed additional transformations to complete the proof of the system, and could not be proven by the automatic strategies of *Why3*.

The invariant is going to be used to prove that, after reaching a stable configuration where there is always only one token active, the number of tokens never increases or decreases.

```
predicate inv (w:map node state) =
```

```
  (w 0 = 1  ∨  w 0 = 3) ∧ (w (n_nodes-1) = 0  ∨  w (n_nodes-1) = 2)
  ∧
  (forall n:node. has_token w n ->
    (has_token_R w n ∧ not has_token_L w n)
     ∨
    (has_token_L w n ∧ not has_token_R w n))
  ∧
  atMostOneToken w n_nodes
```

Since the *lemma* `atLeastOne` already proves that all worlds have at least one node, that part of the proof is already confirmed and it is not needed in the invariant, so the only part left is to test that there is always a maximum of one token. Here we also check that no node with token can have both tokens from left and right at the same time. Since this rule is something that was added because of the limitation of only allowing stable configurations we decided to only add it to the invariant portion of the code instead of changing the system code directly, as it would change how the system behaves as a whole, instead of affecting only the stable configuration behavior.

Finally, in order to evolve a given world it is necessary to define the `step` behavior.

```
let ghost function evolve_value (h:node) (w:map node state) : state
  requires { validNd h }
  requires { has_token w h}
  requires { inv w }
  ensures { inv (set w h result) }
= if h = 0 || h = n_nodes-1 then incr2 (w h)
                            else incr (w h)

inductive step world node world =
  | step_enbld : forall w :world, n :node.
    validNd n ->
      has_token w n ->
        step w n (set w n (evolve_value n w))
```

The `evolve_value` function checks if all the requirements needed to take a step on a given node `n` are met and says how the state on the node is going to change. For that, the node needs to be inside the array (`validNd n`), it needs to have a token (`hasToken w n`) and the world needs to be already in a stable configuration where the invariant is true (`inv w`). Beside these, after changing the state of node n, the invariant still needs to hold true on the updated world (`inv (set w n result)`).

With this, it is important to define some *lemmas* about how the step behaves in order to check if the system is working as intended, just like in the previous example.

```
lemma indpred_step :
  forall w w' :world, n :node. step w n w' -> inv w -> inv w'
```

First, the *lemma* `indprep_step` ensures that no step creates a new world where the previously defined invariant does not hold. `step_preserves_states` makes sure that only one node changes with each step.

```
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
```

Next, `step_TR` is going to be used to represent reflexive transitive closure, where more than one step is taken, and `noNegative_step _TR` makes sure that, in order to reach another world `w'` from `w`, there cannot be a negative number of steps, there can be either 0, if `w = w'`, or a number bigger than 0 for any other case.

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

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

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

Finally, the predicate `reachable` is going to be used to check if, starting on an initial world, it is possible to reach the world `w` in a number of steps. `indpred_manySteps` is a version of `indpred_step` that can take in consideration more than one step by using `step_TR`, and can secure that the invariant is maintained through any number of steps. Additionally, `indpred_reachable` says that any world that is reachable from an initial one should still hold the invariant, no matter how many steps it takes.

`noNegative_ste_TR` and `indpred_manySteps` needed the *induction_pr* transformation to complete their proof because they use the inductive predicate `step_TR` in their definition. In conjunction with

`atLeastOne` explained previously, these where the only *lemmas* that needed additional transformations to prove.

At first, it was needed to limit the total number of nodes in the array to reach any conclusion about the preservation of the invariant after steps. This was happening because the *Why3* context was not complete, as there were some *lemmas* missing. The properties about the bidirectional array presented previously were translated into *lemmas* to aid the verification process, allowing *Why3* to reach the conclusion even without a limit in the number of nodes.

As a final goal, the system must be able to prove that, for any reachable world, there is always at least one token and at most one token at all times, after reaching a stable state.

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

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

The keyword *goal* normally identifies the main property to proof, since it is not added to the *Why3* context like *lemmas* do. This way, any other proof made after it will not know that the system can prove this goal. If that was the objective, then it should be coded as a *lemma* instead. This *goal* is where the main property of convergence is being tested. By proving that any reachable world holds the predicate `oneToken` we are proving all legitimate worlds have a single token present, no matter the steps taken.

With this, all of the system related rules are ready, and the system should be able to check some other small examples of worlds and steps being taken. For example, by giving *Why3* a world we can specifically check if the system is working as intended for that given case:

```
lemma token_w2s :
  forall w w':world. (w 0 = 1 ∧ w 1 = 2 ∧ w 2 = 2) ∧ n_nodes = 3
                       -> has_token w 0
                       -> step w 0 w'
                       -> has_token w' 1
```

Giving *Why3* the world [1,2,2], which has a token in node 0, after taking a step in that node, the token should have moved to node 1, since the new world will be [3,2,2]. This can easily be proved by *Why3*.

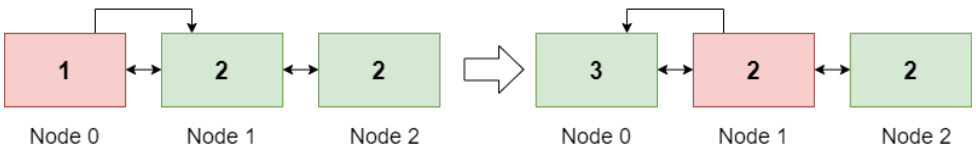


Figure 21: Example of the *lemma* `token_w2s` Step

It is also possible to show other properties being tested on these examples such as `atLeastOneToken` and `atMostOneToken`.

```
lemma token_w5_atleast :
  forall w :world. (w 0 = 2 ∧ w 1 = 2 ∧ w 2 = 1 ∧ w 3 = 0)
                   ∧ n_nodes = 4
                   -> atLeastOneToken w 4

lemma token_w5_atmost :
  forall w :world. (w 0 = 2 ∧ w 1 = 2 ∧ w 2 = 1 ∧ w 3 = 0)
                   ∧ n_nodes = 4
                   -> not atMostOneToken w 4
```

Here the world [2,2,1,0] is used to prove that the system can identify both of these properties, as it can prove that there is at least one token, and that the `atMostOneToken` is not true, since there are two tokens in the array, on node 2 and 3.

As a more generic approach, it is also possible to model the table 1 presented in the earlier chapter with all the possible moves.

Case	World	Step	World'
c)	x+1 <- tx	x+2 x+1	tx+1 -> x+2

For example, the case shown can be written as:

```
lemma prop_c : forall w w':world, n:int. 0 < n < n_nodes-1
                                      -> inv w
                                      -> w (n-1) = incr (w n)
                                        ∧
                                          w (n+1) = incr2 (w n)
                                      -> has_token_L w n
                                      -> step w n w'
                                      -> has_token_R w' n
```

By manually comparing the states of the neighbour nodes, it is possible to check if the node has a token from the left side. If a step is taken, the value in the middle node will change, and the token in the new world will be in the same node, but coming from the right.

It is also possible to translate some of the conclusions made when analysing the System on the previous chapters. These allowed *Why3* to add to its context when trying to prove the final *goal*, eventually

allowing an unbound proof, where the total number of nodes is not limited. For example, in a stable configuration, all the values to the left of the token must be 1 or 3, and all to the right must be 2 or 0. This will be implemented in a different invariant presented ahead.

```
lemma token_LR:
  forall w:world. inv w ->
    forall n:node. validNd n /\ has_token w n ->
      (forall l:node. 0 <= l < n -> w l = 1  \/  w l = 3)
      /\
      (forall r:node. n < r <= n_nodes-1 -> w r = 2  \/  w r = 0)
```

In any stable world `w` if node `n` of the same world has a token then all states before it (`0<=l<n`) should either have the state 1 or 3, and all the values to its right (`n<r<=n_nodes-1`) should have state 0 or 2.

It is also possible to prove that every token state is dependent on which type of token it is. If it is a token from the left then its value must be 1 or 3, and if it is from the right it must be 0 or 2.

```
  lemma val_l_stable:
    forall w:world. inv w ->
        forall n:node. 0 < n <= n_nodes-1 /\ has_token_L w n ->
            w n = 0  \/  w n = 2

  lemma val_r_stable:
    forall w:world. inv w ->
        forall n:node. 0 <= n < n_nodes-1 /\ has_token_R w n ->
            w n = 1  \/  w n = 3
```

All of these are easily proved with *Why3*, and help in proving some of the main properties faster, since some of the verification conditions, like the unbounded proof of the invariant, need to use part of these to prove the correctness of the system.

3.2 Alternative Invariants

After proving all of these properties, some different invariants came to mind. The invariant used before (`atMostOneToken`) only limits the total number of tokens in the array, but it can be changed to also limit the states of the nodes. With the information presented before about the three possible configuration types in the array, the Base, Stable and Complete Stable configuration. The later two can be used to create a different invariant.

3.2.1 Stable Configuration Invariant

In this configuration, the array is already stable, but the values to the right and to the left of the token can be two different states. These are affected by what type of token is present, if it is either `has_token_L` or `has_token_R`.

So, just like it was explained previously, for a *world* `w`, this invariant can be written as:

```
...
∧
 (forall i :int. 0<i<=n_nodes-1 ∧ has_token_L w i ->
    forall k :int. (i<k<=n_nodes-1 ->
        (w[k] = w[i]  ∨  w[k] = incr2 w[i]) ∧ not has_token w k)
    ∧
    (0<=k<i ->
        (w[k] = incr w[i]  ∨  w[k] = incr3 w[i]) ∧ not has_token w k))
∧ (*-----------------------------------------------------------*)
 (forall i :int. 0<=i<n_nodes-1  ∧ has_token_R w i ->
    forall k :int. (i<k<=n_nodes-1 ->
        (w[k] = incr w[i]  ∨  w[k] = incr3 w[i]) ∧ not has_token w k)
    ∧
    (0<=k<i ->
        (w[k] = w[i]  ∨  w[k] = incr2 w[i]) ∧ not has_token w k))
```

This invariant is supposed to replace only the part corresponding to `atMostOneToken`, not the limitations of the extremity states or that a node cannot have token from both sides. This predicate limits the states of all nodes, for each case of token. The first part is used when there is a token from the left (`has_token_L`). This causes all the states to the right (`i<k<=n_nodes-1`) of the one with token to either have the same value or twice its increment, as well as all the nodes to the left (`0<=k<i`) to be its increment or increment three times. Since the configuration is already stable, no other node besides it should have a token. The second part is just like the first one, but for nodes with token from the right. This also causes the possible states to flip from the right to the left, as explained before.

In addition to this, using the *lemma* `token_LR`, that limits the values to the left and to the right of the token to 1 or 3, and 0 or 2 respectively, it is possible to simplify this invariant even further. Since we already know what the possible states of the nodes are, there is no need to compare them with the state of the token, or separate both types of tokens.

```
...
```

```
  ∧
   (forall i :int. has_token w i ->
      forall k :int.
        (0 <= k < i -> (w[k] = 1  ∨  w[k] = 3) ∧ not has_token w k)
          ∧
        (i < k <= n_nodes-1 -> (w[k] = 2  ∨  w[k] = 0) ∧ not has_token w k))
```

This is the main invariant used in this thesis, since not only is it easier to understand but it can also be used in a wider variety of cases, since the complete stable configuration only accepts arrays that are in a more advanced stable configuration than the simple stable does. This will also be the invariant used to test the refinement technique later.

3.2.2 Complete Stable Configuration Invariant

Finally, when the array is completely stable, all states to the right or to the left of the token should have the same value, and only one token is present. So, and following a structure close to the previous one, this can be written as:

```
  ...
  ∧
   (forall i :int. 0<i<=n_nodes-1 ∧ has_token_L w i ->
      forall k :int.
        (i<k<=n_nodes-1 -> w[k] = w[i] ∧ not has_token w k)
          ∧
       (0<=k<i -> w[k] = incr w[i] ∧ not has_token w k))
   ∧ (*--------------------------------------------------------*)
   (forall i :int. 0<=i<n_nodes-1  ∧ has_token_R w i ->
      forall k :int.
     (i<k<=n_nodes-1 -> w[k] = incr w[i] ∧ not has_token w k)
          ∧
     (0<=k<i -> w[k] = w[i] ∧ not has_token w k))
```

When compared to the previous invariant it is easy to see that this is just a version where instead of two possible states, there is only one for all nodes that do not have token. Also, in this case the states of other nodes cannot be directly compared since different tokens will only allow one state to the right and one to the left. It is still possible to use the simple stable invariant to these type of worlds, but this version limits the type of stable configurations a bit more.

3.2.3 Comparison of Invariants regarding Proving Time

It is possible to compare the proving time when using these different invariants in the system. Since the invariant for stable configurations and complete stable configurations are limiting the actual state of the nodes, the time needed is probably going to be different when compared with `atMostOneToken`, which only checks if they have tokens or not. With this in mind, it is not clear weather or not using these invariants is going to decrease or increase the time needed to prove the properties of the system. The solver used for a specific part of the program can have a harder time proving only if the node has token, or it can take longer to compare the states of the array.

Using various predicates with each invariant, and using the simpler version of the stable configuration invariant, these are the average measurements, using the solver *Z3* for `evolve_value` and `OneToken`, for the time needed in seconds and the total number of steps needed to complete the proof:

	evolve_value	OneToken
atMostOneToken	1.12 s 3 194 642 steps	0.12 s 53 372 steps
Stable	0.71 s 1 790 052 steps	0.04 s 58 622 steps
Complete Stable	0.71 s 1 543 031 steps	1.82 s 3 124 600 steps

Table 2: Times in seconds and number of steps for the different Invariants, using the *Z3* solver

The time needed can easily change with how much work the machine is doing at the time, as in these the measurements would be inconsistent between themselves, but the number of steps would still be maintained. This is because no matter the work load, the solver must reach the same conclusion in the same number of steps, but it can change how much time it takes to reach the conclusion, so we will use the number of steps to compare the results.

The time needed to prove `indprep_manyStep` was also checked, but it is not presented because it was very similar with every invariant used (around 0.07 seconds and 21 000 steps). As a reminder, this *lemma* checks if the invariant is maintained through many steps, and uses `evolve_value` to do so. The main part of the time needed is in the `evolve_value`, so it is only normal that changing the invariant does not change the time for `indprep_manyStep`. The same happened when measuring the work for the propositions of the possible steps, where the solvers had relatively close results for all, no matter the invariant used (between 300 and 2200 steps needed).

The same does not happen to `evolve_value`, where there is a clear difference in the observed values. This might happen because `atMostOneToken` is being called recursively to all nodes of the

array , while both of the other invariants are called only once for each. Between the Stable and Complete the difference is probably because the last one is just simpler, since the states can only be one on each side of the token.

Finally, we have `OneToken`, that tests if all worlds reachable have at least and at most one token. Using the *atMostOneToken* invariant this conclusion is almost direct, since the system already proved that there is always at least one token from the *lemma* `atLeastOne`, and it is using the `atMostOneToken` as an invariant, which is true for all reachable worlds, so the proof is already completed. The time for the Stable invariant is very close to this one. This might happen because the system already knows the values of the nodes through the invariant, and these only allow one token to be created. Lastly, the invariant of the Complete Stable state is the slowest.

Since we presented two different options for the invariant of stable configurations, we also decided to compare the results between them:

	evolve_value	OneToken
Stable	3.10 s 5 539 256 steps	0.24 s 404 135 steps
Simpler Stable	0.71 s 1 790 052 steps	0.04 s 58 622 steps

Table 3: Comparison of results for both versions of Stable configuration Invariant

The basic Stable invariant limits the states to the right and the left by comparing them to the state of the node with token, while the second one directly says what the states are. Just by looking at the code it is easy to see that the simpler version should work faster. Both types of tokens are tested once, which cuts the work in half, and there are no comparisons with the token value, which leads to a much faster proving processes.

Chapter 4

Refinement-based Formalization

In this chapter we will talk about the Refinement technique and how it can be used in conjunction with *Why3* to prove similar algorithms in a quicker and simpler way, exemplifying with both examples already explained.

It is fairly easy to see that there is a good amount of redundant work between the two systems presented. Most of the code used to create the base structure is almost identical, as is the code regarding the transition rules of both systems. Besides, the code of most of the auxiliary predicates, like `noTokens` and `atLeastOneToken`, is the same as well. Some related works (Muhammed Basheer Jasser) already identified this situation and presented the idea of Refinement in order to facilitate the construction of similar systems.

The main objective of the Refinement technique is to reduce the amount of code needed to work with similar systems where, just like with the Self Stabilizing systems, some parts of the code are reusable for multiple examples. Besides, it also allows an easier formal verification process, since proving the code that is shared between examples only needs to be done once. From a basic specification, if it is possible to prove the behavior of a system built on it, then it should be possible to prove another one that is built on the same base in an easier way, since part of the verification process is already done from the first example.

4.1 Refinement Implementation

With this in mind, the specification of both examples can start with a simpler world, that is called *Abstract World*. In this case, the abstract world will be a map of ints and bools, representing weather the node has token or not.

The actual specification of each example will be made on top of this abstract map using a *Concrete World* that will have the rules about the states of the nodes and how they evolve with each transition.

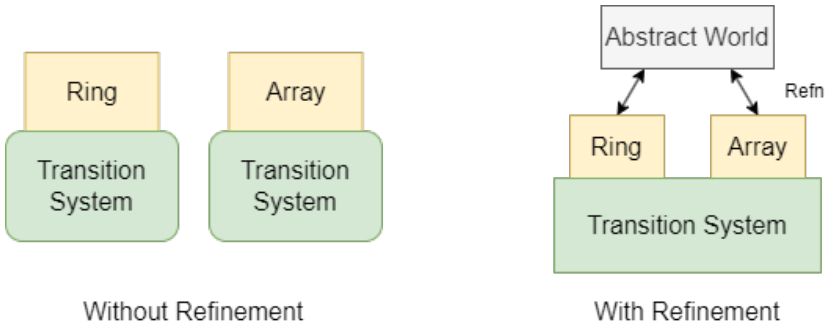


Figure 22: No Refinement versus Refinement

This way, when creating the code for both examples, the part about the transition system rules, like properties related to the invariants and steps are only written once, and only need to be proven once. Besides, since the abstract world is separate from the concrete world all the code about the tokens and their properties, like the *lemmas* `atLeastOneToken` an others, are also separated from the code of the specific system.

4.1.1 Inductiveness Module

To start, the module `inductiveness` defines the base structure of the system. Since both examples are state machines, using a separate module to define how their base is structured allows a quicker and easier construction.

For that, the module begins with a simple definition of an Invariant, as well as *lemmas* to specify the initial world and the step predicate.

```
1  type world
2  predicate inv (w:world)
3
4  val ghost predicate initWorld (w:world)
5    ensures { result -> inv w }
6
7  val ghost predicate step (w1:world) (w2:world)
8    ensures { result -> inv w1 -> inv w2 }
```

Besides this, it also has some small definition about how the steps work to check if the invariant is maintained through the evolution of the system, just like in the previous examples (`step_TR`, `inv_manySteps`, `reachable` and `inv_reachable`). These are going to be used to guarantee that any world created using this module can take steps without creating a world that does not follow the given

invariant.

This module is going to be cloned when the abstract world is created in order to allow it to use these properties. By changing this module to a different one it is possible to change the type of systems used to other types different from transition machines.

4.1.2 Refinement Module

Since the refinement is going to be made between the concrete and the abstract worlds, it is important to define their relations with one another.

For this, the `refn` function is used to transform a concrete world (`worldC`) into an abstract world (`worldA`).

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

Just like we explained in the chapter about transition systems, to be able to apply the refinement with the *inductiveness* module, each world will need to be able to transform the following from the concrete to the abstract world:

1. Invariant - Predicate that is maintained through all the possible worlds;
2. Step Function - Function used to evolve the world, changing some of its values;
3. An initial World - Possible initial configuration from which the system will evolve.

Each of these will be transformed from the concrete to abstract world in order to complete the refinement process, and allow to check properties of both types of worlds.

With an invariant of the concrete and abstract worlds, `invC` and `invA` respectively, it is possible to define a generic initial configuration.

```
1  val ghost predicate initWorldA (w:worldA)
2    ensures { result -> invA w }
3
4  val ghost predicate initWorldC (w:worldC)
5    ensures { result -> invC w }
6    ensures { result -> initWorldA (refn w) }
```

With these, we ensure that the initial abstract world follows the respective invariant. Besides that, the initial concrete world follows the concrete invariant and its refinement must be equal to the result of

`initWorldA`, since the `refn` function transforms the `worldC` into the respective `worldA`. So, for any initial correct concrete world, the respective transformed abstract world must follow the abstract invariant, so that any initial world created is correct for both the concrete and abstract worlds.

After the initial world, it is necessary to write how the transitions and their refinement should behave.

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

val ghost predicate stepC (w1:worldC) (w2:worldC)
  ensures { result -> invC w1 -> invC w2 }
  ensures { result -> invC w1 -> refn w1 = refn w2
                               \/  stepA (refn w1) (refn w2) }
```

Just like with the initial world, a step in the abstract world should maintain the invariant, and the same should happen in the concrete world. Besides that, using the `refn` function on the `w1` and `w2` of the concrete world should prove that the transition made on the concrete world does not change the respective abstract world into one that does not follow the `invA`, or it simply does not change the refinement of the abstract world. This can happen in the previously presented examples when a step changes the direction from where the token is made, but not the node that has the token.

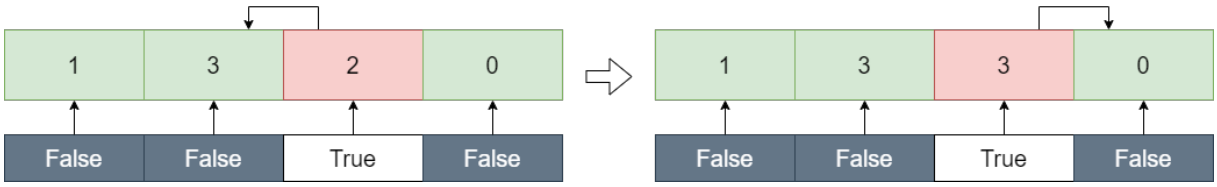


Figure 23: Abstract World not changing with a step

Following this, it is important to define the reachability properties that have been explained previously, such as `inv_manySteps_C` and `inv_manySteps_A` that prove that the invariant is maintained through steps, and `inv_reachable_C` and `inv_reachable_A` that prove that if a world is reachable, than it still checks the given invariant.

Finally, and in order to check all of these using the refinement technique, it is needed to prove that the relations between the abstract and concrete world are maintained when we use the `refn` function.

```
lemma manySteps_refinement : forall w w' :worldC.
  invC w -> stepC_TR w w' -> stepA_TR (refn w) (refn w')

lemma reachable_refn : forall w :worldC.
```

```
    reachableC w -> reachableA (refn w)

 lemma invariance_refn : forall w :worldC. reachableC w -> invA (refn w)
```

For that, the lemma `manySteps_refinement` is used to prove that in a concrete world that follows the `invC`, if it is possible to go from the configuration `w` to `w'`, then it is also possible to do the same applying the `refn` function to both worlds on the abstract world. The same can be applied to the reachability property, since a reachable concrete world must have its respective reachable abstract world. The lemma `invariance_refn` is used as a conclusion that any concrete world that is transformed into a abstract one still maintains its abstract invariant.

Why3 allows the user to clone different modules as templates for others, and this is what allows us to clone this refinement process using different concrete and abstract worlds.

Cloning using Why3

By cloning a module, *Why3* can import a copy of all the information present in a different module to itself, be it predicates, *lemmas* or any other definition. Besides that, it is possible to give specific instances to the cloned module, which creates additional verification conditions. As an example, we have the following modules:

```
module Level1
use export int.Int
    val function f (x:int) : int
        requires { x > 20 }
        ensures  { result > x+100 }
end
(*-------------------------*)
module Level2
use export int.Int
    val function f (x:int) : int
        requires { x > 5 }
        ensures  { result > x+200 }

clone Level1 with val f
end
```

These two modules, `Level1` and `Level2`, define a different function each, that adds at least 100 or

200 to the *int* `x`, which needs to be higher than 20 and 5, respectively. `Level2` then clones `Level1` by instantiating this new function. By doing this, `Level2` will now inherit the information present on `Level1`, but using the function instanced when the cloning happened.

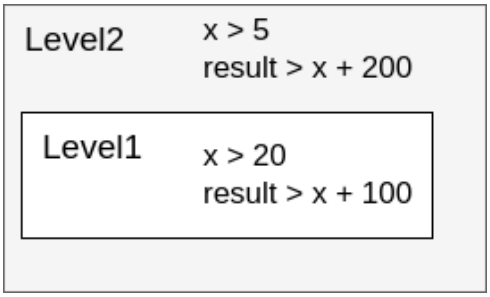


Figure 24: Information present at *Level2*

With this, *Level2* will now try to check if their post-conditions include the *Level1* post-conditions, which in this case is true, since x+200 > x+100. It also checks if the pre-conditions of *Level1* include all the values of the pre-condition of *Level2*, since 5<20. Grouping this two relations, *Why3* creates the following verification condition automatically, after cloning:

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

function f int : int

f'spec : forall x:int. x > 5 -> f x > (x + 200)

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

goal f'refn'vc :
  forall x:int.
   x > 20 ->
   x > 5 /\ (let result = f x in result > (x + 200) -> result > (x + 100))
```

Figure 25: Context of the new refinement verification condition

With this, *Why3* is able to prove the *Level2* using as an abstract base *Level1*. In more complex systems this allows the user to create generic rules, for example, `Level1`, which will then be cloned in other modules (`Level2`) to check if it follows what was defined on the first one.

4.1.3 Abstract World Module

As stated before, this module will focus on controlling the tokens, keeping track of where they are in the system and changing their location when a transition is made.

The base structure is a map of nodes and bools, which will keep track of which node has the token by marking it with `true` if they do, or `false` when they do not. It is not important for the abstract world to

know the specific state of each node, since their transformations will be given later by the concrete world, and here it is only needed to change the map of bools.

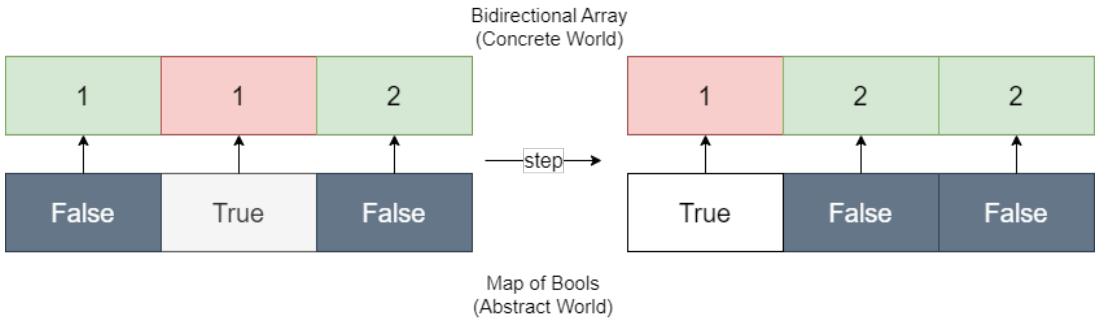


Figure 26: Example of Abstract and Concrete World Configurations

Just like the concrete world, this one is limited to at least three nodes, having a total of `n_nodes`. In this module there are also some generic definitions of properties to check on the token map, such as `noTokens`, `atLeastOneToken` and `atMostOneToken`. Each is going to be used to check properties on the abstract world, in this case, about the number of tokens present in the array.

As we explained previously, throughout the refinement work we will consider that the configuration is already stable, which means that only one token will be present at the time, since the rules do not allow the number of tokens to increase, and it is also impossible to have 0 tokens. The same work could be done about reaching a stable configuration using refinement, but it would need a different approach and it is not going to be studied in this thesis. With this, the invariant that is going to be used is the junction of both properties, `atMostOneToken` and `AtLeastOneToken`.

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

With this predicate we ensure that the abstract world must always have only one token, which means, only one true bool on the map, representing a stable world.

```
predicate initWorld_p (w:world) =
  w.token[0] /\
  forall n :int. validNd n /\ n > 0 -> not w.token[n]

let ghost predicate initWorld (w:world) = initWorld_p w
```

The initialization of the world is also fairly simple. The bools will all be false except for node 0, since it needs have one token to follow the invariant. Finally, the step function:

```
predicate trans_enbld (w:world) (c:node) (n:node) =
  validNd c ∧ validNd n ∧ c<>n ∧ w.token[c]

let ghost function trans (w:world) (c:node) (n:node) : world
  requires { trans_enbld w c n }
  ensures  { inv w -> inv result }
= { token = w.token[c<-false][n<-true] }

inductive stepind world world =
| trans : forall w :world, c n :node.
          trans_enbld w c n -> stepind w (trans w c n)

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

The predicate `trans_enbl` is used to check if the nodes that will be changed are in the map range, if they are not the same token and if the first one was set to true before the step, which means that it had a token. There is no need to check if the other node used has token since the invariant says that there is only one token at any time. Next, the `trans` function changes the states on the map in order to set the old token to false and the new one to true. And lastly, the predicate `step` is used to change the world using all the above.

Since this module is based on a transition system, it will clone the `inductiveness` module in order to instance this type of systems, instantiating the parameters `world`, `inv`, `step` and `initWorld` created here.

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

With this, the module *oneToken* is now a transition system, which has access to both definitions from its own and the *inductiveness* module, and can now be used as the abstract world for the refinement.

4.1.4 Concrete World Module

Having the main refinement module and the respective abstract world defined, it is now time to work on the concrete world. This is where there can be some deviations in behaviors between different systems. We will present the same examples from before, the Self Stabilizing Ring and Bidirectional Array, but it should be possible to create other variations that could be built on top of the same abstract world using the refinement technique.

In these examples, the concrete world says how the system evolves, sending the message to the abstract world about how tokens are changed.

Just like with the abstract world, these worlds need an invariant, an initial world and a transition function. The invariant used should focus on the values of the concrete world, so the previously used `atMostOneToken`, which focuses more on the token present in the abstract world, is not the most useful. So the invariant of the stable configuration, which limits the states of the nodes without token to the right and the left, is the most appropriate.

Even with refinement, we will only test the formal verification of the system after reaching a stable configuration. Just like explained previously, it is also possible to prove the processes of stabilization using a variant.

Even though the code of these modules will be similar to the ones created without refinement in mind, the main difference will be in how it accesses the states, since they are now stored inside a different structure.

Besides, it is important that any transformations made on the concrete world changes the abstract world to the respective one, since both of them need to be synchronized in order to be able to tell which node has token or not.

Finally, and with the objective of connecting the concrete and the abstract worlds, the `refinement` module needs to be cloned using the respective parameters.

4.2 Self-Stabilizing Ring's Refinement

As mentioned before, the states are now inside a map structure of `nodes` and `states`, which represent the number of the node and the value inside it. With this, it is necessary to change all the accesses made to these states throughout the module.

```
1 type world = { ring : map node state }
```

Since the idea is to prove this module using the refinement technique, the main function needed is the function `refn` that transforms the concrete world, a map of `nodes` and `states`, into a abstract world, a map of `nodes` and `bools`.

For that, it is needed to check which node has token to create the corresponding abstract world.

```
 predicate has_token (w:world) (i:node) =
  (i = 0 &&  w.ring[i] =  w.ring[n_nodes-1])
   ||
  (i > 0 && i < n_nodes && w.ring[i] <> w.ring[i-1])

let ghost function refn (w:world) : OneToken.world
= { token = has_token w }
```

Now, and following the rules of the ring algorithm explained before, given a concrete world it is possible to create the abstract world to test the desired properties about tokens instead of testing them on the concrete world. Since `refn` calls the `has_token` just by giving it the world as a parameter, it will be tested for all nodes on the world, creating the map of `nodes` and `bools` just as explained in the abstract world module.

```
predicate inv (w:world) =
  forall n :int. validNd n -> 0 <= w.ring[n] < k_states
  /\
  (w.ring[0] = w.ring[n_nodes-1] ->
    forall k :int.
      (0<k<n_nodes -> w.ring[k] = w.ring[0] /\ not has_token w k))
    /\ (*-----------------------------------------------------------*)
  (forall i :int. 0<i<n_nodes /\ w.ring[i] <> w.ring[i-1] ->
     forall k :int.
       (i<k<n_nodes -> w.ring[k] = w.ring[i] /\ not has_token w k)
         /\
       (0<=k<=i-1 -> w.ring[k] = w.ring[i-1] /\ not has_token w k))
```

The invariant's main focus will be to check if a node having a specific token causes all other nodes to not have it, just like in a stable configuration. If the node 0 has token, then all other nodes must have the same value as it, causing no other node to have token. In the other case, if any other `i` node has a token, because the state of the previous node (`i-1`) is different, then all the nodes after `i` (`i<k<n_nodes`) must have the same value that `i`, and all the previous must have the same value of `i-1` ($0 \leq k < i$).

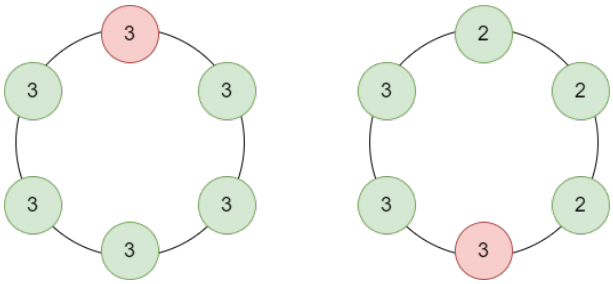


Figure 27: Cases of the Ring Invariant

The initial world is just a simple initialization of the map to have token on the first node, which is the same node with token on the initialization of the abstract world (node 0).

```
predicate initWorld_p (w:world) =
  forall n :node. validNd n -> w.ring[n] = 0

let ghost predicate initWorld (w:world) = initWorld_p w
```

Finally, the `trans` function must check if the properties of the abstract world are kept or not after evolving the concrete world.

```
predicate trans_enbld (w:world) (h:node) =
  validNd h ∧ has_token w h

let ghost function trans (w:world) (h:node) : world
  requires{trans_enbld w h}
  ensures {inv w -> inv result}
  ensures {inv w -> h=n_nodes-1 -> refn result = OneToken.trans (refn w) h 0}
  ensures {inv w -> h<n_nodes-1 -> refn result = OneToken.trans (refn w) h (h+1)}
  ensures {inv w -> refn result = refn w  ∨  OneToken.stepind (refn w) (refn result)
  }
= let st = if h = 0 then incre (w.ring[n_nodes-1])
                    else w.ring[h-1]
  in
  { ring = w.ring[h<-st] }
```

As a pre-condition the system needs to try to evolve from a valid node that has token. There are four post-conditions. These will work almost as *lemmas* when proving the refinement verification conditions. Directly specifying how both worlds behave in any case of token will aid the proof when testing the refinement relations, since it already states how the concrete and abstract world evolve.

The first post-condition is easy to understand, since the idea is basically the same of the work without refinement. A step made from a valid world should never be able to create an invalid world, so the invariant should be maintained.

The second and third post-condition are the start of the relations between the two worlds. From the first explanation of the ring algorithm, it is known that when node 0 has token, the step made will remove it by taking the state of the node `n-1` and increasing it once, creating a token on node 1. So the token needs to move from node 0 to node 1 in the respective abstract world. When it is any other node but the first, it will take the state received from the previous one, creating a token on the following node, since this world should already be in a stable configuration as explained in the invariant part.

With this, and looking at the invariant, the token moves from the node N-1 to node 0 in the first case, and from node h to node h+1 in the second, so the respective abstract worlds should represent this new configurations when the step is taken, which is tested in the the following line of code:

```
refn result = OneToken.trans (refn w) h 0
```

The result of a step in the world *w*, in this case, when the node with token is node n_nodes-1, should make the respective abstract world (refn w) change the token from node n_nodes-1 to node 0.

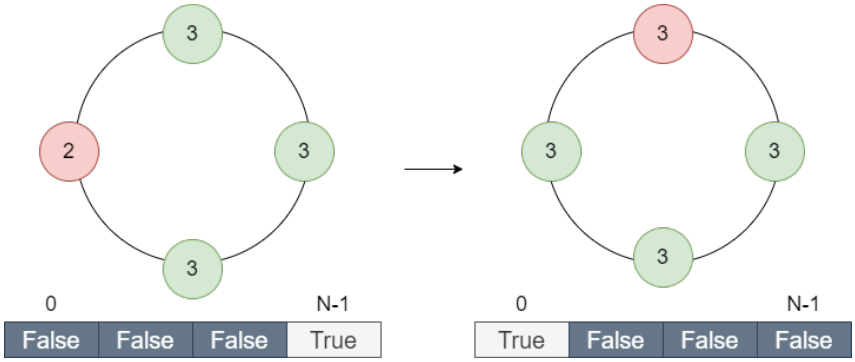


Figure 28: Evolution of Ring Worlds with token on N-1

In the same way, the result of a step in the world *w*, when the node with token is node h, which is any other but n_node-1, should make the respective abstract world (refn w) change the token from node h to h+1, since the token is always passed to the next node.

```
refn result = OneToken.trans (refn w) h (h+1)
```

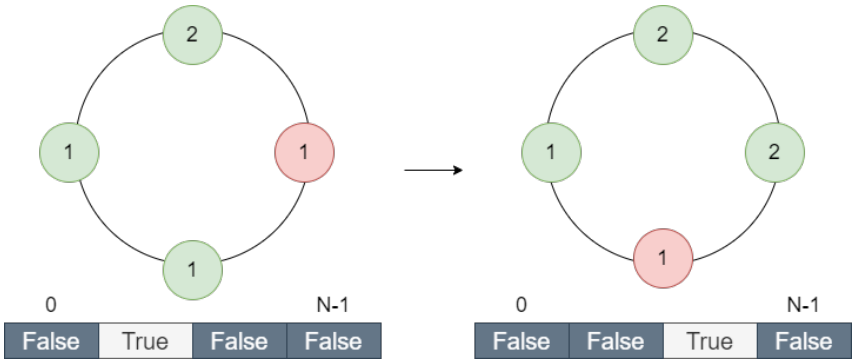


Figure 29: Evolution of Ring Worlds with token on other node

The last post condition says that it is possible to have a step not change the respective abstract world with a step on the concrete. This does not happen in this example after the system reaches a stable configuration, since all possible transformations must change the token from a node to another.

Finally, and to reach a conclusion about how the refinement of this system works with the abstract world module, we need to clone the refinement module, giving it both the abstract and concrete worlds.

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

goal oneToken : forall w :world. reachableC w -> oneToken (refn w)
```

By cloning the refinement module with both world types, the function that transform one into another, their invariants, initial worlds and step functions, it is possible to test if the all properties explained on the refinement module are maintained, when using the given values and predicates. This creates additional verification conditions that check that the properties stated in this concrete module follow the ones created on the abstract one.

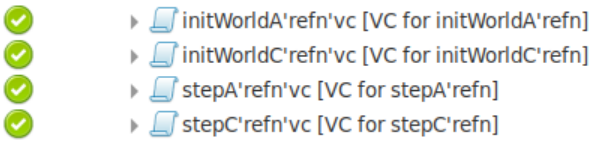


Figure 30: New Verification Conditions created by the cloning

With these, we know that the relation between the concrete and abstract world are correct, since it was possible to check all the refinement properties, but now with the respective abstract and concrete

worlds present in the context of the proof. Finally, the `oneToken` goal checks if all reachable concrete worlds have their respective abstract world, using the properties in the abstract module. This would not be possible if the verification conditions of the refinement cloning were not hold.

4.3 Self-Stabilizing Bidirectional Array's Refinement

Finally, and as a last example, we will present the bidirectional ring system proven using the refinement technique. Most of the code is the same as in the example without the refinement idea, but with the changes needed because of the new main structure `world`, just like in the previous system.

So, just like in the Self Stabilizing Ring, the transition function is going to be used to change the states of the concrete world, as well as move the token to the correct node when needed.

```
let ghost function trans (w:world) (n:node) : world
  requires { trans_enbld w n }
  ensures  { inv w -> inv result}
  ...
= let st = if n = 0 then incr2 (w.arr[n]) else
           if n = n_nodes-1 then incr2 (w.arr[n]) else
           incr w.arr[n]
  in
  { arr = w.arr[n<-st] }
```

For that, the node that is going to change needs to have token (`trans_enbld w n`), and the state is going to change according to the normal rules. Also, the new world must still follow the invariant of the concrete world. For this we need to add different post-conditions to make sure that the new token is being passed correctly to the abstract world. Like in the Ring example, there is going to be a post-condition for each type of token possible, where the behavior of the token needs to be specified, and those are the cases of token on node 0, on node `n_nodes-1`, and for any other node. These will assist *Why3* when proving the relations between worlds, after cloning the *refinement* module.

For when the node 0 has token, just as explained previously, the node is going to increment its state twice, and pass the token to node 1. The same happens when the node `n_nodes-1` has token, but the one that is going to have token after the step is node `n_nodes-2`. So this can be seen as:

```
ensures  { inv w -> has_token_R w 0 -> refn result =
    OneToken.trans (refn w) n (n+1) }
```

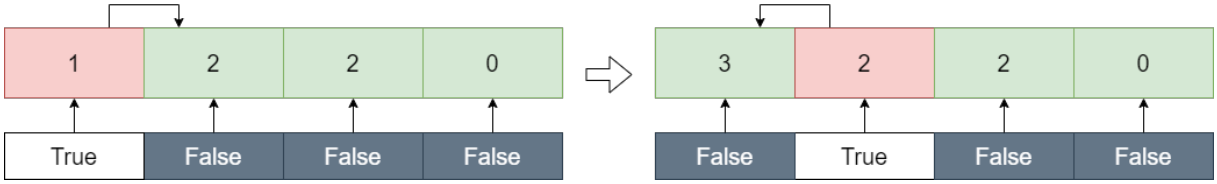


Figure 31: Evolution of Bidirectional Worlds with token on node 0

```
ensures  { inv w -> has_token_L w (n_nodes-1) -> refn result =
           OneToken.trans (refn w) n (n-1) }
```

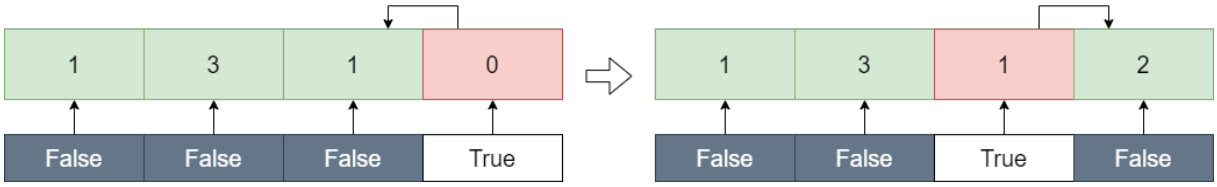


Figure 32: Evolution of Bidirectional Worlds with token on node n_nodes-1

With this we ensure that after the step is made, the respective abstract world is going to transform the transformed concrete world (`refn w`) using the `trans` function defined on `OneToken`, that changes two nodes on the map, the first to false, and the second to true. This way, for the case of `n=0`, the token is being passed from node 0 to node 1, and in the case of `n=n_nodes-1` it is being passed to `n_nodes-2`.

For the rest of the cases where the nodes are not 0 or `n_nodes-1` the process is similar. Depending on which type of token is present, either `has_token_L` or `has_token_R`, the token is going to be passed to the right or the left respectively on the abstract world.

```
ensures  { inv w -> has_token_L w n ∧ n<>(n_nodes-1) ->
   refn w = refn result  ∨  refn result = OneToken.trans (refn w) n (n+1) }
```

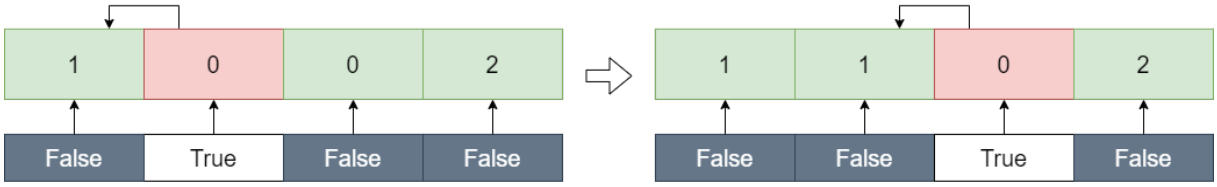


Figure 33: Evolution of Bidirectional Worlds with token from the Left

```
ensures  { inv w -> has_token_R w n ∧ n<>0 ->
   refn w = refn result  ∨  refn result = OneToken.trans (refn w) n (n-1) }
```

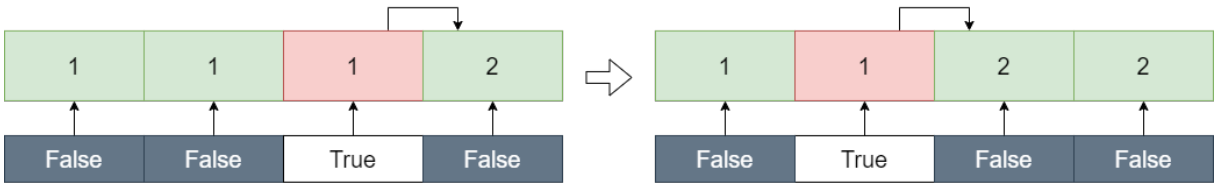


Figure 34: Evolution of Bidirectional Worlds with token from the Right

It is important to specify that nodes `0` and `n_nodes-1` should not be considered on these post-conditions, since their behavior is the opposite. For example, when the token is on node `0`, the token is being created from the node to the right (`has_token_R`), and the abstract world is going to pass the `True` bool to the right. But in this definition, tokens that are created from the right are going to pass the token to their left. Also, in these there is a possibility that the step does not change the respective abstract world. This happens when a step in the concrete world does not change which node has token. So here the transformation of the concrete into the abstract world before and after the step should be the same (`refn w = refn result`), since there is no change in the node with token.

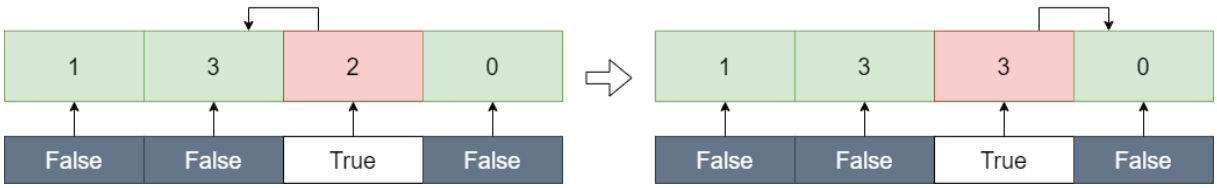


Figure 35: Evolution of Bidirectional Worlds when token does not move

Finally, there is a generic case. In this example this could be removed, since it is known that there are no more possible steps than the ones already mentioned.

```
1  ensures  { inv w ->
2       refn result = refn w  \/  OneToken.stepind (refn w) (refn result)}
```

With this it is possible to ensure that, no mater the type of token, after the step is taken, the respective abstract world is also going to change according to the changes made on the concrete world. All that is left is to define the actual step function.

```
1 inductive stepind world world =
2  | trans : forall w :world, n :node.
3      trans_enbld w n -> stepind w (trans w n )
4
5  let ghost predicate step (w1:world) (w2:world) = stepind w1 w2
```

Finally, now that the entirety of the system is modeled, it is possible to complete the refinement process by cloning the *refinement* module with the respective variables and predicates.

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

goal oneToken : forall w:world. reachableC w -> oneToken (refn w)
```

With these, *Why3* will create additional verification conditions to check if the respective relations made between the abstract and concrete world on the *refinement* module are still maintained when using the abstract world `OneToken` and the concrete world defined here.

Chapter 5

Conclusions and future work

In this chapter we discusse some conclusions drawn from the work developed in this thesis. It will be divided in two parts, one for the simple example of the array and the other for the refinement technique. Finally, We will also add some ideas of possible future work regarding this subject.

5.1 Conclusions

We think that the main objective of this thesis, to study the use of *Why3* as a tool when working regarding the Formal Verification of Distributed Systems, was reached. In particular, we have formalized a self-stabilizing array system and proved its closure property for stable configurations. Our works shows that such systems can effectively be verified with a high degree of automation using *Why3* and SMT solvers.

We also introduced the Refinement idea and implemented it using both examples shown previously. Refinement in the *Why3* context is something that has not been studied before, so the work developed in Chapter 4 will be useful, presenting a case-study that can be seen as an example for similar works. The workflow between *Why3* and solvers is maintained even when applied to the refinement component, so this is not an obstacle to create other proofs using it.

Besides, this work also brings to light the differences between the original formalization and the one based on refinement. Both took a similar amount of effort to implement and prove, so, when looking at their formal verification process it should be clear that the refinement has the advantage, in terms of specification, of allowing the user to implement multiple systems on top of the same abstract configuration. This does not harm the overall proving process, even allowing the properties to be implemented at their natural (abstract or concrete) level, like how token *lemmas* and properties about the states of nodes are specified in different modules.

Finally, Self-stabilizing Systems have never before been presented as refinements of an abstract specification has done in Chapter 4 of this thesis, so this is also an interesting case study.

We will now present some of the difficulties and problems that we encountered during the development of this work, separating the two main parts of this thesis, the first regarding the work done initially, without refinement, and the second with the refinement technique in mind.

Simple Implementation

Some properties about the Self-stabilizing Bidirectional Array are fairly easy to understand and notice, but it seems that some of them have not been identified before, like the idea that only some values can have token from the left or the right on stable configurations. These were easily translated into *lemmas* that allowed the solver to work more efficiently, even allowing the system to be proven without a limit on the total number of nodes, just like what happened with the Ring example.

When compared to the Self-Stabilizing Ring, the Self-Stabilizing Array needed more work done in order to be fully proved. This was because, just by not having a connection from the first to the last node, the transition rules of the system allowed a bigger variety of different steps to be made, and also because this system allows the state to be read from the previous and following node. The first example has a total of 2 types of behaviors for nodes with token (token on node 0 on any other), while the Array has 4 (token on the first, last and any other, with the latter one be divided in two case: token from left and right).

During early stages of this work we were having problems with the main structure *world*, even trying to change completely like so:

```
type state = Zero | One | Two | Three

  let function incr (x:state) : state
= match x with
  | Zero -> One
  | One -> Two
  | Two -> Three
  | Three -> Zero
  end
```

This way, it would bypass all the possible problems that might appear with *ints*, such as the `incr` functions or the `has_token`. After having that version working, it was easy to change it back to the version using the *int* theory.

When looking at the provers themselves and the actual proving process, we used *Z3* for most of the early process, since it could deal with most proofs, even before we added the additional *lemmas* that helped simplify the proofs. In conjunction with *CVC4*, these had the added benefit of being capable of

creating counter examples, which showed us how the system was working, and where the proof was failing, by presenting the variables that were created and where in the proof the error was found. The "power" of these counter-examples is completely on the side of the provers, and so, sometimes the examples would be incomprehensible. Depending on what we were testing, things like negative array indexes, negative numbers and other unexpected values would appear, some times only to one of the provers, while the other was working just fine. At the end of the work, most of the proofs could be made using *Alt-ergo* as well. This usually has some difficulties proving more complex properties, but would be even faster most of the time than the other two provers used. However, in some proofs we needed to apply some *Why3* transformation to simplify them, like simple *splits* of the proof in the various components, or using *induction_pr* to specify that the proof should be made using induction on some predicates, which happened with some recursive *lemmas* like the ones used to prove that properties were being maintained through multiple steps.

The invariant also had some changes through the work. In earlier stages, we started by using only the properties of "at least" and "at most one" token as invariant, but the at least one was causing problems. So we decided to start by proving it as a separate *lemma*, since just by having the extremity states correct there should always be at least one token present. When we started noticing other properties on the array we reached the other versions of the invariant explained previously. The latter also have the idea of `atMostOneToken` present, but besides checking the tokens they also limit the states of each node.

Regarding the extra properties, like the ones presented in table 1, they started as a way to identify which cases where causing problems in the proofs, but by the end we think they also show a different approach possible to explore with *Why3* where the propositions and *lemmas* are being directly applied to a *world* given by the programmer. These also directed us to possible mistakes on the system's code since if any part of it was not correct, like how the system checks for the token, at least one of these would detect an inconsistency.

Refinement Formalization

Most of the problems we encountered while working on the implementation with the refinement technique were present because parts of the main example were not correct. Since we started working on the refinement portion of the code while the first one was still in development it was clear that there would be inconsistencies.

Most of the code is the same when comparing both examples, besides the way the states are accessed. The main differences were the invariant and the `trans` function. The invariant, just like previously ex-

plained, had a lot of alternatives possible, and at the time we started working on it we still had the invariant of the simple version only as *atLeast* and *atMost*, which was later removed. The idea for the invariant used on both actually came from the refinement portion of the work, since it made sense that an invariant that is going to be tested on the concrete world should focus on the actual value of each node of the array, and not on the tokens as we were approaching it before.

Finally, the `trans` function caused some difficulties. Even with an invariant that looked correct, and had no problems on the simple version of the array without refinement, it would still find inconsistencies when using it for this function. The adopted strategy used was to divide it as much as possible while trying to find a solution: Splitting the post-conditions as much as possible, and dividing the possible cases (token on 0, `n_nodes-1` or any other). Even with this, the proof itself also had to be divided, since for each type of step, there were three possibilities of nodes to consider. Here *Why3* would actually present an inconsistency when testing. For example:

```
1    ensures  { inv w -> has_token_R w n ∧ n<>0 ->
2        refn w = refn result  ∨  refn result = OneToken.trans (refn w) n (n-1) }
```

Here, by splitting the proof of this post-condition, *Why3* would present three new ones, one for case 0, one case `n_nodes-1` and another for the else. The definition of `has_token_R` already says that the node cannot be `n_nodes-1`, so only the else proof is important. The problem was that *Why3* would be able to prove this for the three cases, but for the two that should never happen in the system, in this case n=0 or n=n_nodes-1, it would present an inconsistency. So once we managed to complete the proof for all the parts, simplifying and removing splits actually removed this problem. This probably happened because *Why3* was detecting that we were trying to prove something that was impossible to happen, so it would try to warn the user about the redundant proof.

Another problem that we noticed was that the idea of refinement would create conditions that were too complicated for the provers. So in order to simplify this work we used the `unfold` transformation that would substitute the function definition directly on the proof. This was mostly used with the function `refn`, which transforms the concrete world into an abstract one. By using the `unfold refn` transformation, the verification condition would change the function `refn` to its definition directly. This might create code that is not as easy to read, but it reduces the amount of work needed for the solver to test the properties.

5.2 Prospect for Future Work

As possible approaches for related work, the first thing that comes to mind is the proof of the Convergence property. As mentioned before, this would most likely be focused on a variant that would change its value depending on how close the array is to a stable configuration. As is known, there are no steps that increase the number of tokens, so all possible steps would lead to a stabler configuration. A variant just representing the number of tokens would not be enough, since there are steps that do not reduce this number, but still evolve the configuration into one closer to stabilization. The length of the array and the position of the tokens would also influence how close it is to the stable configuration. For example, only arrays with an even number of nodes can have token in all nodes.

It is also possible to expand to other systems on the refinement part of the work. The strategy of only allowing one token can be compared to some mutual exclusion algorithms, so it should be possible to also implement them using the same abstract model. Having multiple components share a connection to a resource where only one can access it at the time is very similar to having multiple nodes in an array where only one checks the property for having token. It is easy to see that `atLeastOne` property does not apply to mutual exclusion problems, since it is possible that no components is accessing the shared portion, but the `atMostOne` still applies, since there can only be one component accessing it at the same time.

Bibliography

Thesis github. https://github.com/Keeper17/Why3_thesis.

Ivy web page. http://microsoft.github.io/ivy/.

TLA+ web page. https://lamport.azurewebsites.net/tla/tla.html.

François Bobot Jean-Christophe Filliâtre Andrei Paskevich, Claude Marché. Why3: Shepherd your herd of provers. 1993.

Manos Kapritsos Jacob R. Lorch Bryan Parno Michael L. Roberts Srinath Setty Brian Zill Chris Hawblitzel, Jon Howell. Ironfleet: Proving practical distributed systems correct. URL https://www.microsoft.com/en-us/research/publication/ironfleet-proving-practical-distributed-systems-correct/.

Coq. The coq proof assistant. URL https://toccata.gitlabpages.inria.fr/toccata.

John Ousterhout Diego Ongaro. In search of an understandable consensus algorithm.

Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. 1974.

Jean-Christophe Filliâtre. Sum and maximum, in why3. URL http://toccata.lri.fr/gallery/vstte10_max_sum.en.html.

Sukumar Ghosh. *Distributed Systems An Algorithmic Approach*. COMPUTER and INFORMATION SCIENCE SERIES. CRC Press, 2014. ISBN 978-1-4665-5298-2.

Pavel Panchekha Zachary Tatlock Xi Wang Michael D. Ernst Thomas Anderson James R. Wilcox, Doug Woos. Verdi: A framework for implementing and formally verifying distributed systems. URL https://homes.cs.washington.edu/~ztatlock/pubs/verdi-wilcox-pldi15.pdf.

Cláudio Belo Lourenço Jorge Sousa Pinto. Why3-do: The way of harmonious distributed system proofs. URL https://repositorium.sdum.uminho.pt/handle/1822/78595.

Leslie Lamport Stephan Merz Kaustuv Chaudhuri, Damien Doligez. Verifying safety properties with the TLA+ proof system. 1993.

Leslie Lamport. Paxos made simple. 2001.

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

Jamilah Din Abdul Azim Abdul Ghani Muhammed Basheer Jasser, Mar Yah Said. A survey on refinement in formal methods and software engineering. URL https://www.warse.org/IJATCSE/static/pdf/file/ijatcse16814sl2019.pdf.

Marco Schneider. Self-stabilizing. 1993.

Ted Herman Sriram V. Pemmaraju Sukumar Ghosh, Arobinda Gupta. Fault-containing self-stabilizing algorithms. URL https://coq.inria.fr.

Tocatta. Tocatta - formally verified programs, certified tools and numerical computation. URL https://toccata.gitlabpages.inria.fr/toccata.

Why3. Why3 - where programs meet provers. URL http://why3.lri.fr/.