Distance-Vector Routing Protocols in mCRL2

A practical comparison of fixed-point operators to helper functions for model checking

Mert Yılmaz[1]
Supervisor(s): Benedikt Ahrens[1]**, Anna Lukina**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Mert Yılmaz
Final project course: CSE3000 Research Project
Thesis committee: Benedikt Ahrens, Anna Lukina, Tim Coopmans

Abstract

Model checking is a formal verification technique that can be used to ensure correctness of software systems. mCRL2 is a specification language with an accompanying toolset that can perform various operations including model checking. Modal μ-calculus is the property language used by the mCRL2 toolkit. The fixed-point operators included in modal μ-calculus allow it to express inductive properties. These operators give modal μ-calculus very high expressivity, allowing it to encompass other temporal logics such as LTL, CTL, and CTL*. However, this expressivity comes with some downsides as modal μ-calculus formulae can be hard to understand and take more time to verify compared to less expressive logics. In this study we compare the use of fixed-point operators to the use of helper functions for expressing three properties in a case study based on four distance-vector routing protocols. We find that the properties expressed using fixed-point operators were more performant in terms of verification time. We also found that fixed-point operators lead to larger logic expressions but smaller overall expressions when taking the size of the helper functions into account. Furthermore, we conclude that fixed-point operators can be better suited for people experienced with them, while helper functions can be better suited for the average programmer, at least for this type of task.

1 Introduction

Some software errors can be very costly, especially in safety-critical applications such as aerospace and healthcare [29]. While software testing can be a useful tool in detecting these errors, it by itself does not provide any correctness guarantees. Model checking is a formal verification technique that can verify that a specification of a system satisfies a given property.

In model checking, a system is verified. The system is usually a non-deterministic (finite) state machine that transitions between various states. To verify the correctness of a system with model checking, it first needs to be described in a formal language. This description is called a specification, and the language used for specification is called the specification language. Then, how the system is expected to behave needs to be described in a logical language. These expected behaviors are called properties, and the language they are expressed in is called the property language. Finally, both the specification and the properties are fed into a software program called a model checker. This program then checks whether the specification satisfies the property or not.

For example, assume somebody wants to verify the communication between two computers is correct. First, they would need to express that the first computer could send a message and the second computer could receive a message as a specification. Then, they would need to express that the communication is correct as a property. This could be done with "All messages sent by the first computer are eventually received by the second computer." expressed in a temporal logic that supports time based expressions such as "eventually". Finally, the model checker would be run on these inputs, and it would either verify that the communication is correct, or it would provide a counter-example in which the second computer never receives a message sent by the first computer.

There is a variety of model checkers such as CMBC [9], NUSMV [8], UPPAAL [3]. Each model checker has property language(s) they support with their own advantages and disadvantages. mCRL2 [17, 6] is a specification language with an accompanying toolset that includes such model checking capabilities. Unlike most other model checkers, mCRL2 uses a property language based on first-order modal μ-calculus. Compared to other commonly used property languages such as linear temporal logic (LTL) [24], computation tree logic (CTL) [10], and their common superset CTL* [16], modal μ-calculus is more expressive [13]. This means it can express and allow verification of a wider range of properties. This expressiveness comes from the so-called fixed-point operators, which allow definition of recursive predicates.

Theoretically, there are modal μ-calculus formulae that cannot be expressed in CTL*, such as asserting that a proposition p holds at every even numbered state [4, 23]. However, in practice, many such properties can be reexpressed with small modifications to the specification or with the help of additional functions. In addition, this expressiveness comes with some downsides, such as making model checking undecidable [22], or the use of fixed-point operators making formulae hard to understand. While there is a wide range of results about the theoretical expressiveness of modal μ-calculus, there are relatively few results about the benefits of these fixed-point operators and their expressiveness in practical use with mCRL2.

The goal of our research is to investigate the usefulness of the fixed-point operators in a more practical context. In particular, we try to answer the research question: "What are the advantages and disadvantages of using fixed-point operators to express a property compared to using helper functions when verifying distance-vector routing protocols in mCRL2?". Our subquestions are to investigate the advantages and disadvantages of using these operators in terms of their verification time and ease of understanding. The reason we decided to focus on distance-vector routing was that it provided a context in which interesting recursive properties can be defined, which are a main use case of fixed-point operators. Our main contribution is a case study of four distance-vector routing protocols. We express three different properties these protocols are expected to satisfy both using fixed-point operators and helper functions. We compare their performance as measured by model checking time and their ease of understanding as measured by the size of the expressions. Overall, we find that properties based on fixed-point operators are both smaller overall and verify in less time. However, they are also more likely to contain errors that cause invalid results from model checking.

Section 2 gives some relevant background information about mCRL2, modal μ-calculus, mCRL2 model checking, and related work. Section 3 explains the chosen problem, the protocols, and the properties that will be used for the case study. Our chosen metrics, observations, and benchmarking setup are explained in section 4. The results of the study are presented in section 5. Our interpretation of these results are given in section 6. Finally, section 7 presents our conclusions and potential future work that can build on top of this work.

2 Background

This section first explains the parts of mCRL2 and first-order modal μ-calculus that are most relevant to this paper. Then it explains how the mCRL2 toolkit can be used to do model checking. Finally, it presents an overview of the relevant literature and how they motivated the research question of this paper. Detailed descriptions of mCRL2, modal μ-calculus, and the mCRL2 toolkit can be found in the mCRL2 documentation [28].

2.1 mCRL2

mCRL2 is a specification language based on the algebra of communicating processes. In an mCRL2 specification, there are processes which can optionally execute in parallel. These processes can perform certain series of actions which are the behavior of the system being modeled. They can also synchronize by doing certain actions at the same time, which is called communication. As an example, consider the following mCRL2 specification that demonstrates the most important parts of the mCRL2 language:

```
1  act
2      send, receive, communicate, print : Nat;
3      wait;
4  proc
5      Proc1 = wait . Proc1 + sum (n : Nat) send(n) . Proc1
6      Proc2 = wait . Proc2 + sum (n : Nat) receive(n) . print(n) . Proc2
7  init
8      allow(
9          {communicate, print, wait},
10         comm(
11             { send|receive -> communicate },
12             Proc1 || Proc2
13     ));
```

The specification starts by declaring the actions that this system can perform. The send, receive, communicate, and print actions have `: Nat` at the end of their declaration, on line 2. This means these actions carry a natural number parameter. The wait action, defined on line 3, on the other hand has no parameters.

Then the next section has the definition of the two processes. These two definitions contain three different operators: `+`, `.` and, `sum`. The `.` operator represents sequential composition. When a process does an action containing this operator it first performs the left-hand side action and then performs the right-hand side action. The `+` operator represents alternative composition. A process can do either the left-hand side or the right-hand side of this operator. Finally, the `sum` operator represents an alternative choice across all possible values a variable of a certain type can have. For instance, `sum (n : Nat)` represents an action where `n` can be any natural number. Putting these definitions together the behavior of the two processes can be understood. The first process either waits and repeats itself or it sends an arbitrary natural number and repeats itself, as defined on line 5. The second process on the other hand either waits and repeats itself or it receives an arbitrary natural number, prints it, and then repeats itself, as defined on line 6.

Finally, the last section defines the initial process. It contains three new operations: `allow`, `comm`, and `||`. The `||` operator represents parallel composition of the two processes. When two processes are composed in parallel, their actions happen in arbitrary order, including interweaving and simultaneous actions. Hence, `Proc1 || Proc2` at line 12 means processes one and two are executing in parallel. The `comm` operator is the communication operator. It allows processes to communicate by doing actions with equal parameters simultaneously, resulting in a new action. For this specification, `{ send|receive -> communicate }` at line 11 means a communicate action is generated whenever the two processes do send and receive actions at the same time, with the same natural number parameter. Finally, the `allow` operator restricts the set of possible actions that can be performed by a process to the ones listed in it. For this example, it only disallows the send and receive actions from occurring on their own. This ensures all messages that are received were sent by the other process and vice versa.

Overall, this example specification represents a system where a process can send an arbitrary natural number for the other one to print in a loop. The processes can also wait instead of performing these actions.

2.2 Modal μ-calculus

The mCRL2 toolkit allows model checking of mCRL2 specifications with respect to first-order modal μ-calculus properties. The variant of modal μ-calculus used by mCRL2 is based on typed first-order logic. Most notably, it includes modal and fixed-point operators in addition to predicates, boolean operators, and typed quantifiers.

Conventional modal μ-calculus has two types of modal operators which are duals of each other. They are $[a]\phi$ and $\langle a\rangle\phi$ where a is an action and ϕ is a modal μ-calculus formula. Semantically, $[a]\phi$ means formula ϕ holds in all states reachable by taking action a. Similarly, $\langle a\rangle\phi$ means formula ϕ holds in a state reachable by taking action a. As these operators refer to future states, they allow the expression of temporal properties that depend on future states. In the variant of modal μ-calculus used by mCRL2, these operators are extended to take a regular formula instead of an action a. The formal grammar for these regular formulae is the following:

```
RegFrm ::=  ActFrm |
            RegFrm . RegFrm |
            RegFrm + RegFrm |
            RegFrm * |
            RegFrm +
```

These regular formulae are similar to regular expressions, including concatenation (.), choice operator (infix +), zero or more occurrences operator ($*$), and one or more occurrences operator (postfix +). These patterns generalize modal operators from considering the next action to considering a sequence of actions. Furthermore, instead of using actions as the building blocks in the patterns, it uses action formulae. The modal operators only consider states reachable by taking actions that satisfy the action formula. These formulae can contain predicates, boolean operators, and typed quantifiers. These modifications allow a modal operator to consider a wide variety of actions or action sequences.

As an example demonstrating the use of these operators, consider the property that the example system

given in the last section can print any number. This property can be expressed as follows:

$$\forall n : \mathbb{N}.\langle \text{true} * .\text{print}(n)\rangle true$$

Directly translated, this formula states that for all natural numbers n, there exists a path containing an arbitrary number of actions satisfying true and then the action print(n), such that in the final state true is satisfied. As true is always satisfied, the formula can be interpreted to say that for all natural numbers n, there is a path of actions that ends in print(n).

Similarly, modal μ-calculus includes two types of fixed-point operators which are also duals of each other. They are $\mu X(d_0 : D_0 = v_0, \ldots).\phi$ and $\nu X(d_0 : D_0 = v_0, \ldots).\phi$, the least and greatest fixed-point operators respectively. Both operators bind a predicate variable X, which optionally has a number of parameters d_i of types D_i with values v_i. This variable can then occur in formula ϕ, as long as it is under an even number of nots plus left-hand sides of the implication operator. This ensures the right-hand side is monotonic in the variable, which in turn guarantees the existence of least and greatest fixed points by the Knaster-Tarski theorem [27]. Semantically the right-hand side ϕ can be thought of a function that maps a predicate X to a formula. A predicate P is the fixed point of this function if it is mapped to itself. More precisely it is a fixed point if and only if $P \iff \phi[X := P]$, where $\phi[X := P]$ is ϕ with every instance of X replaced with P. With this, the least and greatest fixed point are the fixed points that are true for the smallest or the largest set of states respectively. These fixed-point operators allow the expression of recursive and inductive properties and greatly increase the expressiveness of modal μ-calculus.

As an example using fixed-point operators, consider the property that each print action comes after a communication action with the same parameter, with no print action between these actions. In other words, process two prints only values it receives, and it prints the values it receives only once. This property is expressed by the following formula:

$$\begin{aligned}
&\nu X(n_1 : \mathbb{N} = 0, \text{canPrint} : \mathbb{B} = \text{false}).\forall n_2 : \mathbb{N}.\\
&\quad [\text{print}(n_2)](n_1 == n_2 \wedge \text{canPrint} \wedge X(n_1, \text{false})) \wedge\\
&\quad [\text{communicate}(n_2)](\neg\text{canPrint} \wedge X(n_2, \text{true})) \wedge\\
&\quad [\text{wait}](X(n_1, \text{canPrint}))
\end{aligned}$$

Here the fixed point $X(n_1, \text{canPrint})$ represents the given property being satisfied, with the last communicated number being n_1 and canPrint being true if this number was not printed and false if it was. The main body of the formula consists of three sub-formulae in conjunction. The first sub-formula states that if a print action is taken with a value n_2, this value has to be equal to the last communicated value n_1 and canPrint has to be true. The value of canPrint is updated to be false as the number has been printed. The second sub-formula states that if a communication action is taken with a value n_2, the previous value should have been printed indicated by canPrint being false. It also updates the last communicated value to n_2 and sets canPrint to true as this value has not been printed yet. Finally, the last clause states that after a wait action, the two parameters do not change. While it is relatively easy to see that the desired property satisfies this fixed point, it is harder to understand why it is the greatest fixed-point. While it does not always work, a useful heuristic to consider is that the least fixed point is used for properties that hold after a finite number of steps while the greatest fixed point is used for properties that should hold throughout the path.

2.3 Model checking workflow in mCRL2

While the mCRL2 toolkit supports model checking, there is not one tool that does model checking. Instead, various tools need to be used in combination to verify that an mCRL2 specification satisfies a modal μ-calculus property. This subsection briefly explains the tools needed for model checking with mCRL2. Figure 1 contains a diagram representing how the tools used for model checking interact.

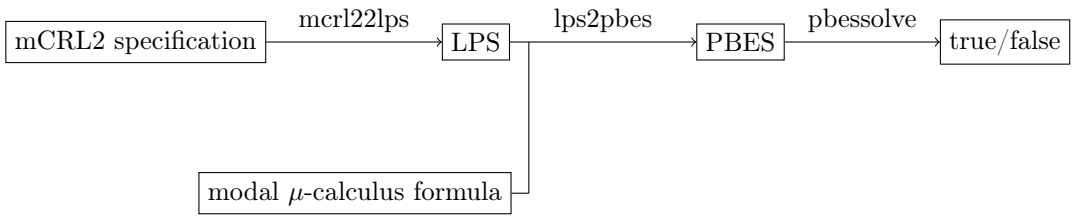

Figure 1: A diagram demonstrating the model checking workflow in mCRL2. First, the mcrl22lps command linearises the mCRL2 specification into an LPS. Then, the lps2pbes command combines this LPS with a modal μ-calculus formula to create a PBES. Finally, the pbessolve solves this PBES and returns whether the mCRL2 specification satisfies the given modal μ-calculus formula.

To verify an mCRL2 specification, it first needs to be converted to an alternative format called a linear process specification (LPS). In a linear process specification, all the parallelism has been removed to create a process that consists of condition-action-effect rules. As an LPS is simpler than a regular mCRL2 specification, it is easier to process by the other tools in the toolkit. The main tool used to linearize an mCRL2 specification is mcrl22lps, which can use various linearization methods with their own advantages and disadvantages.

The next step is to create a parameterized boolean equation system (PBES) encoding the verification problem. The lps2pbes tool can be used for this purpose, taking a linear process specification and a modal μ-calculus formula as input. There are other ways of creating a PBES, but they are out of the scope of this paper.

The final step is to solve the PBES to learn whether the specification satisfies the given property or not. There are multiple tools that can be used for this purpose. The main tool that is used for solving PBES is pbessolve, which is the tool we use in this paper. This tool has features like multithreading and a compiled rewriter to speed up the verification process.

2.4 Related Work

In this subsection, we discuss previous works that are related to this study. First, we investigate papers considering mistakes in specifications and how the difficult to understand nature of modal μ-calculus could exacerbate these problems. Then we go over papers about the importance of verification time in model checking and how it can be a disadvantage for modal μ-calculus. After that, we discuss papers that consider the high expressiveness of modal μ-calculus theoretically. Finally, we conclude this section with papers that contain uses of modal μ-calculus in a practical context.

The correctness of properties is important in model checking. Incorrectly stated properties can cause incorrect systems to appear to be correct defeating the purpose of model checking. Beer et al. [1] consider vacuous properties in model checking, which are properties that are true due to their preconditions being impossible. They state that formulae which are vacuously true usually indicate an error, which is a serious problem in day-to-day use. To address this they define a subset CTL they call w-ACTL in which these vacuous properties can be detected, with a non-trivial witness of the property. Chockler, Kupferman, and Vardi [7] and Kupferman [19] similarly highlight the importance of correct properties in model checking, and offer coverage metrics in addition to vacuousness for detecting errors in properties.

Modal μ-calculus and its fixed-point operators are famously hard to understand. This creates the question of whether this makes modal μ-calculus less suitable for model checking, as its complexity could increase the likelihood of errors in specifications. As Bradfield and Stirling [5] states, "Its defining feature is the addition of inductive definitions to modal logic; thereby it achieves a great increase in expressive power, and an equally great increase in difficulty of understanding.". Spronck, Luttik, and Willemse [26] similarly note the hard-to-understand nature of modal μ-calculus. This potential downside of using modal μ-calculus is a part of the motivation behind our subquestion about ease of understanding of properties.

While modal μ-calculus has disadvantages in terms of understandability, it has the advantage of being more expressive than comparable logics. As Cranen, Groote, and Reniers [13] demonstrate, it is possible to express any LTL, CTL, or CTL* property in first-order modal mu calculus with only linear growth in

formula size. This proves modal μ-calculus is at least as expressive as these logics. In fact, any CTL* formula can be represented in modal μ-calculus with a formula of alternation depth three. Since formulae of higher alternation depth are strictly more expressive [4], modal μ-calculus is capable of expressing more properties than CTL*. Okamoto [23] demonstrates one such property, namely that a proposition holds at every evenly numbered state. While it is true that CTL* cannot express this property directly, it still could be used to verify this property by modifying the original model. Indeed, if another parameter representing the parity of the state number is added to the model, this property can be expressed in CTL*. Similarly, using helper functions can make properties previously unexpressible, expressible. However, these additional modifications or code can introduce errors to the verification process, similar to errors in properties. This again raises the question of whether using modal μ-calculus is advantageous or disadvantageous in terms of overall complexity for a specific problem.

Another important consideration in model checking is verification time. Long verification time makes model checking of large models infeasible, and more performant model checking techniques can allow verification of larger models. Rapid increases in the size of the state space and accompanying longer verification times are one of the biggest obstacles for model checking [11]. The expressiveness becomes a downside in terms of performance, with verifying modal μ-calculus formulae taking exponential time in the size of the property in the worst case compared to the linear time of less expressive logics such as CTL [15]. This motivated our second subquestion comparing verification time for formulae using fixed-point operators with formulae using helper functions.

The mCRL2, its toolkit, and its variant of modal μ-calculus has been used in many practical applications. One such application has been the formal specification and verification of the Maeslant Storm Surge Barrier [2]. They showed model checking can be used for a problem of this scale. They successfully verified six properties, after making three of the properties more precise. None of these properties directly used any fixed-point operator, however. Another system that has been verified using mCRL2 is the startup phase of the FlexRay protocol [12]. In this paper, they considered four different scenarios where one of the nodes communicating on the FlexRay bus had an error. They specify two properties using modal μ-calculus and check if the system satisfies the properties under the malfunction scenarios. These properties make use of the fixed-point operators. In particular, the second property that states that "eventually all correctly functioning nodes will keep receiving each other's messages" uses multiple fixed-point operators to keep track of the current message and the set of nodes that has not received the message. While this study demonstrates a practical scenario where the expressiveness of the fixed-point operators are demonstrated, it does not consider any alternative approaches. In general, while there are both papers that directly use fixed-point operators and papers that do not directly use these operators, we could not find any paper comparing these two approaches. This gap in research has motivated us to conduct this study, to demonstrate the advantages and disadvantages of both approaches in a single practical example.

3 Problem description

In this paper, we investigate a case study based on distance-vector routing protocols. In this section, first the details of the problem being considered are explained. Then, the protocols that will be considered in this paper are specified. Next, the properties of the protocols that will be investigated are listed. Finally, our research questions in the context of this case study are stated.

3.1 Distance-Vector Routing

We will study the distance-vector routing problem in this paper. In this problem, a collection of computers are connected by unidirectional communication channels. As not all computers are necessarily connected to each other, some messages need to travel across multiple computers to reach their actual destinations. As each hop between computers adds latency and creates additional load, using the shortest path can be more efficient for these relayed messages. To this end, each computer computes a routing table by communicating with its neighbors. This routing table contains one entry for each potential target computer for a message and which neighbor the message should be forwarded to. The goal is to fill this table completely such that when followed messages take the shortest possible path.

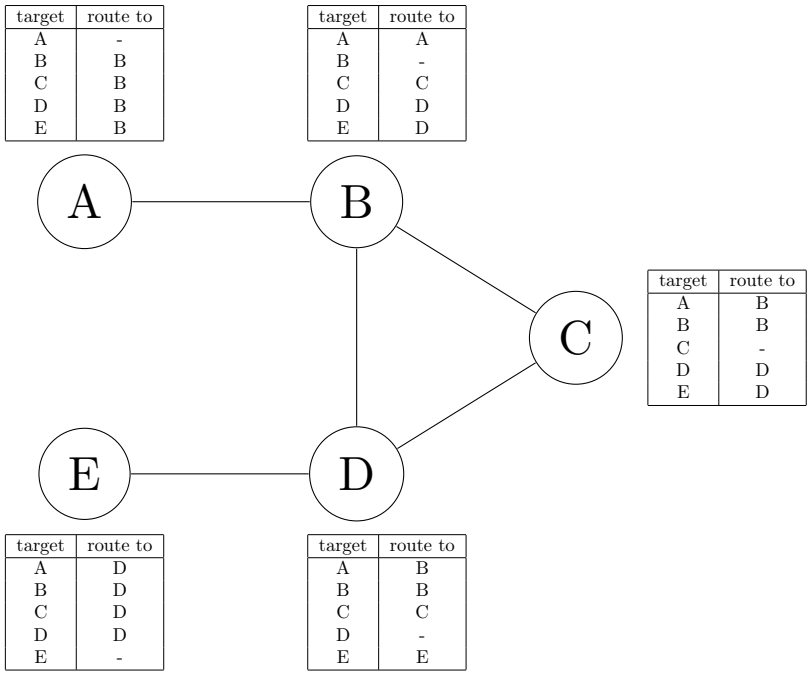

Figure 2: Final state of an example network after distance-vector routing was applied. Each node has a filled table listing where messages should be forwarded based on their target destination. It can be seen that following these tables gives the shortest path to the target node.

In figure 2, the final state of an example network after distance-vector routing was applied can be seen. Initially, each node started with an empty table. After the protocol terminates, each has a full table as this is a connected network. It can be seen that following the routing tables gives the shortest path. As an example assume a message is sent from node A to node E. Node A's routing table says messages going to E should be forwarded to B. Node B's routing table similarly routes the message to node D. Finally node D forwards the message to node E. This path of length three is indeed the shortest path between nodes A and E in this network.

In its full generality, the shortest path routing problem allows edges with different weights as well as connection and disconnection of computers. However, to keep the state space size more reasonable in this paper, we assume a static topology with uniform cost edges. The reason this problem was chosen was that it admits interesting uses of the fixed-point operators outside their regular usage for expressing temporal properties. Furthermore, as most distributed algorithms execute on a network of computers, the results of this case study should be at least partially applicable to other distributed, graph-based algorithms. Finally, these protocols also have practical relevance, as they are commonly used in local area networking [25].

3.2 Protocols

We consider four different protocols for calculating routing tables in this paper. The first two are protocols that correctly calculate the routing tables. Their main difference between them is the level of non-determinism in the protocols. The last two protocols have some errors in them. The first incorrect one calculates routing tables that always succeed in delivering the message, but not always using the shortest route. The second incorrect one calculates incomplete routing tables that can fail to route even if two nodes can reach each other. However, whenever the tables succeed in routing, they use the shortest path. We chose these protocols to see the performance of the two approaches in variety of scenarios such as low/high non-determinism and positive/negative result. The exact mCRL2 specifications for these protocols can be found in appendix A or our GitHub repository (`https://github.com/CARBONTREX/cse3000-mucalc`).

All four protocols are based on message exchanges. Each node starts with an empty routing table, except for the entry for itself. At any moment, a node can try to exchange its routing table with one of its neighbors. When two neighbors attempt an exchange with each other, the exchange occurs and each node gets their neighbor's table. The table they receive can then be used to update their own routing table. The nodes are assumed to be numbered by zero-indexed natural numbers, though any totally ordered unique identifier could be used. These identifiers are used as indexes and values in the routing tables.

The first protocol works in iterations. At the start of each iteration, every node creates a copy of its routing table. Then it tries to exchange this copy with each of its neighbours, in order of their index numbers. After each exchange, the routing table for the node is updated. This is done by filling any entry that is missing in its routing table but is filled in the neighbour's routing table with the index of the neighbour. While the main routing table is updated, the copy that is used for the exchanges in the current iteration is not updated, as this is required for the paths found to be the shortest. After a node finishes exchanging with all of its neighbours, it starts the next iteration, creating a new copy of the updated routing table. Each node does $N - 1$ iterations where N is the number of nodes. After the last iteration finishes, the protocol terminates and the node is expected to have a complete table for all the nodes it can reach.

The second protocol is very similar to the first protocol. The only difference is that instead of each node trying to exchange with its neighbours in a set order, it instead non-deterministically chooses any neighbour that was not exchanged with in the current iteration. While this does not change the result of the protocol, it introduces a large amount of non-determinism to the system as exchanges in an iteration can happen in any order.

The third protocol is based on the first protocol with a small error introduced. Instead of keeping two tables, it uses the same table for keeping updates and exchanging with. While this does not create any incorrect routings, under certain configurations the intermediate updates can cause longer paths to be found earlier than shorter paths. This means the paths the routing tables describe are not always the shortest path.

The fourth protocol is also based on the first protocol with a bug introduced. Instead of performing $N-1$ iterations, it just does two iterations. This results in only routes shorter than three hops to be found. However, the paths that are found are still the shortest.

3.3 Properties

We investigate three different properties the distance-vector routing protocols are expected to satisfy in this paper. Each of these properties is implemented in two different ways: one implementation using fixed-point operators, and one implementation using helper functions. This way, the relative ease of understanding and the verification time of these two approaches can be compared. The modal μ-calculus formulae for the first property are provided as examples, while the rest can be found in appendix B or our GitHub repository.

The first property the protocols are expected to satisfy is that the routing tables should always forward a message to its destination if the destination is reachable from the node. This property will be referred to as the reachability property in the rest of the paper. We expect this property to be satisfied by all protocols for sizes less than or equal to three, and to be satisfied by all protocols except protocol 4 for all sizes. This property can be expressed as follows using fixed-point operators:

$$\forall n_1 : \text{Nat}.n_1 \implies \forall n_2 : \text{Nat}.n_2 < N \implies \forall \text{graph} : \text{Graph}. \quad (1)$$
$$[\text{top\_comm}(\text{graph})] \quad (2)$$
$$((\mu X(n_r : \text{Nat} = n_1).n_r = n_2 \vee \exists n_3 : \text{Nat}.n_3 < N \wedge \text{edge}(\text{graph}, n_r, n_3) \wedge X(n_3)) \implies \quad (3)$$
$$\mu X(n_r : \text{Nat} = n_1). \quad (4)$$
$$\quad n_r = n_2 \vee \quad (5)$$
$$\quad [\forall \text{id} : \text{Nat}, \text{table} : \text{Table}.\neg \text{route}(\text{id}, \text{table})]X(n_r) \wedge \quad (6)$$
$$\quad (\forall n_3 : \text{Nat}.n_3 < N \implies \quad (7)$$
$$\quad\quad [\exists \text{table} : \text{Table}.\text{route}(n_r, \text{table}) \wedge \text{table}.n_2 = n_3]X(n_3)) \wedge \quad (8)$$
$$\quad\quad [\exists \text{table} : \text{Table}.\text{route}(n_r, \text{table}) \wedge \text{table}.n_2 < 0]\text{false}) \quad (9)$$

The formula starts with quantifiers referring to any two nodes and any graph (of a certain size N) at lines 1-2. The first fixed-point formula at line 3 is true if node n_2 is reachable from node n_r in the graph. This is true either when these two nodes are the same as all nodes can reach themselves, or if there is a node n_3 that can reach n_2 which in turn has an edge connecting it to n_r. If n_2 is reachable from n_1, the second fixed-point formula at lines 4-9 also needs to hold due to the implication. This formula is true if node n_2 is reachable from node n_r by following the routing tables. This is true if they are same node just like the previous case. Otherwise, for any action other than routing this property needs to keep holding, which is specified in line 6. Furthermore, for any routing action done by the node n_r the resultant node in routing table should also reach n_2, as specified in line 8. Finally, the table should always have a valid entry for the message, that is it should not fail to route the message at any point, this is specified in line 9.

The expression of this property using helper functions is as follows:

$$\forall n_1 : \text{Nat}.n_1 < N \implies \forall n_2 : \text{Nat}.n_2 < N \implies \forall \text{graph} : \text{Graph}. \quad (10)$$
$$[\text{top\_comm}(\text{graph}).\text{true}*] \quad (11)$$
$$\forall \text{tb}_0 : \text{Table}.[\text{route}(0, \text{tb}_0)] \quad (12)$$
$$\forall \text{tb}_1 : \text{Table}.[\text{route}(1, \text{tb}_1)] \quad (13)$$
$$\forall \text{tb}_2 : \text{Table}.[\text{route}(2, \text{tb}_2)] \quad (14)$$
$$\forall \text{tb}_3 : \text{Table}.[\text{route}(3, \text{tb}_3)] \quad (15)$$
$$\forall \text{tb}_4 : \text{Table}.[\text{route}(4, \text{tb}_4)] \quad (16)$$
$$(\text{reachable}(\text{graph}, n_1, n_2) \implies \text{reachable\_route}([\text{tb}_0, \text{tb}_1, \text{tb}_2, \text{tb}_3, \text{tb}_4], n_1, n_2)) \quad (17)$$

The first two lines are identical and serve a similar purpose. The quantifiers in lines 12-16 bind the final result of the routing process. The formula given is for $N = 5$, as this part of the formula has to be modified according to the value of N. Finally, the last line asserts that if node n_1 can reach n_2 in the graph as calculated by function reachable, then it should also be reachable following the tables as calculated by function reachable_route. The reachable function uses breath first search to check if the first node can reach the second one. The reachable_route uses recursion with a set of visited values. If it detects a cycle or an invalid routing it returns false. If the target destination is reached it returns true. The exact definition of these two functions can be found in appendix A or our GitHub repository.

The second property we defined is that any path defined by the routing table should be at most length $N - 1$. This will be called the distance property in the rest of the paper. We expect all protocols to satisfy this property. Finally, the third property we used was that any path defined by the routing tables should be the shortest possible path. We will call this the shortest path property in the remaining sections. We expect all but the third protocol to satisfy this property.

3.4 Research Question

Our main research question for this paper is "What are the advantages and disadvantages of using fixed-point operators to express a property compared to using helper functions when verifying distance-vector routing

protocols in mCRL2?". As subquestions we will evaluate the verification time and ease of understanding of the three properties we described. The performance of both types of properties will be compared across all four protocols.

4 Methodology

In this section, we explain the methodology we are following to answer the research question. We first explain the metrics and observations that will be used for measuring understandability. We finish by explaining the benchmark setup that will be used to measure verification times.

4.1 Understandability Measures

While ease of understanding is mostly subjective, we still believe there is value in considering an objective metric for it. There are many metrics that can be used for evaluating the complexity of code, and by proxy its difficulty of understanding, such as the number of statements, the number of branches, or cyclomatic complexity [21]. Similarly, there are many complexity metrics that can be used to measure complexity of logical expressions, such as symbol count, quantifier rank, and alternation depth [18, 4]. While there are many complexity metrics for both, there is little overlap that can be used to measure both. As we need to both measure the complexity of functions and logical expressions, we chose expression size as a complexity metric as it is applicable to both types of expressions. To eliminate size differences that are caused by things that do not significantly affect understanding like variable name length, we decided to use the number of nodes in the abstract syntax tree of the expressions as the measure of size. Furthermore, we discarded nodes that only have one child and have no terminal character associated with it. This approach was chosen because these types of nodes mostly happen due to nesting of different types of expressions, and have little effect on the complexity of the expression. As helper-function based properties need to be adapted to problem size, there is no singular size for these properties. As the part that changes is very repetitive and contributes little to complexity, we only counted these parts once while calculating size. The formal grammar for mCRL2 and modal μ-calculus that was used for these calculations can be found in the mCRL2 documentation [28].

While expression size is a useful quantitative metric, it is not enough to draw conclusions on ease of understanding by itself. To get a better understanding of this, we also noted some subjective qualitative observations during the development of these properties. In particular, we noted how understandable the two types of properties were to us with previous experience in programming but without any previous experience in model checking before this paper. In addition, we also noted what kinds of errors were likely during the creation of these properties. We noted if an error would be easily detected, for instance, by causing a crash or an infinite loop, or be harder to detect, for instance, by giving an incorrect result. Ideally, some other tool such as surveys would also be utilized to get less biased results that did not depend on our observations. Unfortunately, this was not possible in the time frame we had for this study, in addition to our other approaches.

4.2 Verification Time Benchmarks

To measure the verification time of both approaches, we use benchmarks. Only the PBES solving time was considered, as linearization of the mCRL2 specification is independent of the property, and PBES generation takes an insignificant amount of time. The "regular2" linearization method was used, as the "regular" method did not terminate for the mCRL2 specification of these protocols. The stack linearization method also succeeded in linearization. However, this method is not advised for any use other than state space generation, so we did not use it for model checking benchmarks [28]. For all tests, 8 threads and the compiled rewriter were used as we expect both multithreading and the compiled rewriter to be used in practice to speed up verification. For each protocol and property combination, we benchmarked increasing graph sizes until a timeout of one hour was hit. We run each benchmark three times to calculate average run times and standard deviations to ensure the data collected were not outliers. The exact benchmarking script used can be found in appendix C or our GitHub repository.

For the benchmarks, an HP Victus 16-s0000 laptop with 32 GiB of RAM and an AMD Ryzen 7 7840HS CPU running at 45W was used. As the number of verification threads, 8, is less than the number of CPU threads, 16, we expect the background OS tasks to have a minimal effect on the verification time. No program that was not needed for the benchmarks was run to ensure this effect would be minimal. As the CPU's cooling was enough to keep the power draw stable during the benchmarks, we expect little variability from the cooling system, power delivery, and ambient temperature on the results. The "7.0.12-xanmod1" Linux kernel was used for the benchmarks. For consistent results, we dropped all Linux kernel caches before each benchmark by writing `3` to `/proc/sys/vm/drop_caches`. To make getting the exact version of the mCRL2 toolkit used easier, we used the Nixpkgs [14] version of the mCRL2 toolkit, as the Nix package manager has strong reproducibility guarantees [20]. The "202507.0.1db00c84f6 (Release)" version of the mCRL2 toolkit from the "567a49d1913ce81ac6e9582e3553dd90a955875f" git revision of Nixpkgs was used for the benchmarks.

5 Results

In this section, first property and function expression sizes, and then the benchmark results are presented.

5.1 Property Expression Sizes

The syntax three sizes for all three properties implemented using fixed-point operators and helper functions can be found in table 1. The size of the abstract syntax tree for the reachability, distance, and shortest path properties were 129, 149, and 160 nodes when the properties are expressed using fixed-point operators. These properties did not require the use of any function that was not a part of the original model. The node counts were 96, 112, and 120 respectively when the properties were expressed using helper functions. However, unlike properties expressed using fixed-point operators, these properties depend on four helper functions, named reachable, reachable_route, distance, and distance_route in the code. These functions had syntax trees of sizes 117, 130, 144, and 165, with a total size of 556. Overall, the properties using helper functions were slightly smaller by themselves, but were overall much larger when considered with their dependents.

property variant	reachability	distance	shortest path
fixed-point based	129	149	160
helper function based (excluding function size)	96	112	120
helper function based (including function size)	343	407	559

Table 1: The syntax tree node size for all three properties. For properties implemented using helper functions, both a size excluding and including function size is provided.

5.2 Benchmark Results

All the protocols got positive results for properties that they were supposed to satisfy, and got negative results for properties that they were supposed to fail. The average run times for verification with the standard deviations across three runs can be found in table 2. The max graph size that could be verified within the timeout was five for protocols 1, 3, and 4, and it was four for protocol 2, for all properties and property implementations. The runtime for problem sizes smaller than these maximums were very similar across the board at around twenty seconds and no noticeable difference could be observed. However, for the maximum problem sizes that could be verified, there are significant differences between run times depending on the protocol, property, and property implementation style. Overall, the properties expressed using fixed-points verified in less time for all protocols at the maximum size, by factors ranging from 1.36 to 2.39. The run time difference was most prominent when verifying protocol 4, and least prominent when verifying protocol 2. The run time ratio was mostly stable between the three properties, showing similar performance differences across them.

protocol and model size	reachability (fixed-point)	reachability (function)	distance (fixed-point)	distance (function)	shortest path (fixed-point)	shortest path (function)
protocol 1, $N = 3$	19.44 ± 0.06	19.70 ± 0.02	19.41 ± 0.04	19.64 ± 0.03	19.61 ± 0.05	19.60 ± 0.07
protocol 2, $N = 3$	20.07 ± 0.02	20.31 ± 0.04	20.61 ± 1.05	20.19 ± 0.06	20.16 ± 0.05	20.23 ± 0.08
protocol 3, $N = 3$	19.46 ± 0.10	19.74 ± 0.06	19.41 ± 0.05	19.61 ± 0.09	20.20 ± 0.08	19.84 ± 0.07
protocol 4, $N = 3$	19.44 ± 0.07	19.70 ± 0.07	19.41 ± 0.06	19.61 ± 0.09	19.53 ± 0.06	19.62 ± 0.05
protocol 1, $N = 4$	20.83 ± 0.08	22.87 ± 0.06	20.92 ± 0.09	22.89 ± 0.06	21.05 ± 0.07	22.93 ± 0.08
protocol 2, $N = 4$	50.67 ± 0.10	72.45 ± 0.67	52.91 ± 0.48	72.20 ± 0.55	52.27 ± 0.30	73.33 ± 0.28
protocol 3, $N = 4$	20.62 ± 0.02	22.53 ± 0.06	20.79 ± 0.05	22.78 ± 0.16	20.82 ± 0.04	22.66 ± 0.04
protocol 4, $N = 4$	21.08 ± 0.10	22.15 ± 0.11	20.42 ± 0.06	22.06 ± 0.10	20.99 ± 0.08	22.69 ± 0.09
protocol 1, $N = 5$	285.54 ± 3.62	526.22 ± 3.35	303.43 ± 3.83	562.05 ± 1.13	298.18 ± 1.62	562.25 ± 4.77
protocol 2, $N = 5$	timeout	timeout	timeout	timeout	timeout	timeout
protocol 3, $N = 5$	252.82 ± 1.27	461.60 ± 2.93	266.65 ± 0.34	489.00 ± 2.19	263.21 ± 2.06	493.92 ± 1.93
protocol 4, $N = 5$	134.59 ± 0.22	315.81 ± 3.44	143.24 ± 0.39	345.10 ± 2.85	141.87 ± 0.30	339.95 ± 0.55

Table 2: The average verification times and standard deviations for various combinations of protocol, model size, and property. All values are given in seconds.

6 Discussion and Evaluation

In this section, we discuss the implications of our results and observations. We first discuss the results and observations relating to ease of understanding for both types of properties. We finish with a discussion of the results relating to performance for both types of properties.

6.1 Ease of Understanding

The logic expressions were simpler when using helper functions. They had smaller abstract syntax trees. Furthermore, most of the expression was boilerplate consisting of quantifiers, and the important part of the expression consisted of only a few logic operations. The expressions using fixed-point operators were more complex in comparison, both in terms of overall size and the type of operators used. While this is true, the properties using helper functions depend on additional code, which cannot be said for the fixed-point based properties. When accounting for the size of the function definitions, the helper function based properties had the higher overall size.

While the sizes of the expressions give some idea on their ease of understanding, they do not tell the whole story. Helper function based properties have larger overall sizes, however, most of the size and complexity comes from function definitions instead of logical expressions. Our observation, as people experienced with programming but not with fixed-point operators, was that the properties using helper functions were easier to understand and write correctly. This is inline with the literature we studied. It is, however, possible that somebody more experienced with fixed-point operators might prefer them, as smaller overall expressions could make verifying their correctness easier.

Another aspect that should be noted is how easy it is to detect errors in the properties during the verification process. During the development of this case study, we noted that errors in properties are usually hard to detect. As long as the formula parsed, it usually did not create any crashes or non-terminating verifications. It instead gave an incorrect result, which could give a false sense of security. Errors in functions on the other, while still capable of causing incorrect results, more commonly caused crashes and non-terminating verifications. For example, using the wrong type of fixed-point operator for the reachability precondition results in successful verification of incorrect protocols. On the other hand, forgetting to update the set of visited nodes in a function results in a stack overflow, which makes it obvious that there is a mistake. As helper function based properties have simpler logical expressions, mistakes in them are more likely to be spotted.

Overall, we found no clear-cut advantage in terms of ease of understanding for either approach. Which approach is better can depend on the experience of the user and, presumably, the type of problem. While this paper focused on distance-vector routing protocols, the results should be at least somewhat applicable to similar graph-based distributed algorithms. It should, however, be noted that these results are based on

limited metrics and subjective observation. It is possible that different problems and users could come to different conclusions.

6.2 Verification Time

We found that for small problem sizes, the runtime was mostly constant at twenty seconds. This was most likely because a constant overhead dominated the runtime for such small problem sizes. The runtime stopped being constant at size four for protocol 2 and at size five for the other protocols. The runtime scaling more rapidly for protocol 2 can be explained by its high non-determinism, increasing the number of states the system had exponentially more compared to other protocols. For these larger problem sizes, fixed-point operator based properties verified in less time across the board.

All three properties performed similarly, with the reachability property usually verifying in slightly less time. There was not a significant slowdown for fixed-point properties with the distance and shortest path properties, despite the larger number of parameters for the fixed-point operators. The three properties continued to have similar verification times, regardless of whether the result of verification was positive or negative. The only significant trend was between the four protocols, with protocol 2 taking the longest, protocol 1 taking less than protocol 2, protocol 3 taking less than protocol 1, and protocol 4 taking less than protocol 3. This order is the same as the order of the state space sizes of the protocols, which suggests state space size is the dominating factor determining runtime.

Overall, we can say that fixed-point based properties perform significantly better than helper function based properties for this case study. A reason for this could be that the mCRL2 toolkit is designed around these operators, making it optimized for this use case and less optimized for the other. This performance difference is higher than what could be explained by variance alone. While significant, the performance difference is not big enough to make model checking tractable for problem sizes where model checking is intractable with the slower approach. However, it should be noted that we could only measure the performance difference for a single problem size. Due to the rapid scaling of the number of graphs of a certain size, one problem size fell between the constant regime and the timeout. It is possible that the two approaches scale differently for larger problem sizes.

7 Conclusions and Future Work

The aim of this paper was to figure out the advantages and disadvantages of fixed-point operators compared to helper functions when expressing properties in mCRL2. The scope of the question was limited to a case study based on distance-vector routing protocols. Two sub-questions focusing on verification time and expression understandability were investigated. Overall, it was found that the verification times of the fixed-point based properties were significantly lower. While significant, the lower verification time was not different enough to make the slower approach intractable for any problem size we tested. The results concerning expression understandability were more varied. Fix-point operator based properties had an overall smaller size, but involved operators that are harder to understand, especially for people inexperienced with them. It also had errors that were less likely to be noticed during verification, compared to helper function based properties. Overall, we conclude that the use of fixed-point operators could be beneficial for people who are experienced with them, while the use of helper functions is better suited for people more experienced with programming, at least for problems similar to this case study.

There are various avenues for future work to build on top of this study. The first avenue of improvement could be to investigate more case studies. Considering more case studies could allow the results to be applied in more general contexts. A less rapidly scaling problem could also allow a comparison of how the verification time scales between the two approaches. Another approach could be to use methods such as surveys to gather opinions of other people, making the results more objective and reliable. Finally, a case study containing formulae that make more complicated use of fixed-point operators could be conducted, such as nesting different types of fixed-point operators. These more complicated uses might be harder to replicate with helper functions or could be unnecessarily complicated, potentially leading to different results than the types of problems considered in this study.

8 Responsible Research and Generative AI Usage

While conducting this study, we made sure to abide by ethical standards. Model checking in theory could be used to verify software that will be used for unethical purposes. However, this is hardly unique to model checking, as any tool or technology can be used unethically in the wrong hands. With that said, we do not foresee any particular unethical usage of this research. The research is conducted as an artificial case study without any personal or private data that could have ethical implications.

Throughout this project, we made sure to make our study reproducible. The exact specifications, properties, and benchmark script used in this study can be found both in the appendix and our linked GitHub repository. The version of mCRL2 used for the benchmarks is listed to aid with reproducibility. We also used the Nix package manager to aid with getting the exact version of the toolkit, including its dependencies. During the benchmarks, we took special care to avoid sources of variability such as additional background OS processes or kernel caching. Finally, the exact hardware that was used for the benchmarks is listed to allow the benchmark setup to be replicated as closely as possible.

The use of generative AI was limited to only aiding with grammar checking in this project. For this purpose, the textLSP language server was used with a locally running gemma3 model. As no content was directly generated by the LLM, there should be no risk of incorrect information coming from the AI model. All grammar recommendations by the LLM were checked to ensure it did not change the meaning of the text. The study did not contain any sensitive information, and as the LLM was running locally, no information left the computer used for writing this paper. Overall, we believe our usage of generative AI poses very little to no risk to the correctness of our paper.

References

[1] Ilan Beer et al. "Efficient detection of vacuity in ACTL formulas". In: *International Conference on Computer Aided Verification*. Springer. 1997, pp. 279–290.

[2] Adrian Beers et al. "A Complete Formal Specification and Verification of the BESW Software Control System of the Maeslant Storm Surge Barrier". In: *Formal Methods for Industrial Critical Systems*. Ed. by Anne Remke and Bernhard Steffen. Cham: Springer Nature Switzerland, 2026, pp. 225–240. ISBN: 978-3-032-00942-5.

[3] Johan Bengtsson et al. "UPPAAL-a tool suite for automatic verification of real-time systems". In: *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III : Verification and Control: Verification and Control*. New Brunswick, NeW Jersey, USA: Springer-Verlag, 1996, pp. 232–243. ISBN: 354061155X.

[4] J.C. Bradfield. "The modal mu-calculus alternation hierarchy is strict". In: *Theoretical Computer Science* 195.2 (1998). Concurrency Theory, pp. 133–153. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(97)00217-X. URL: https://www.sciencedirect.com/science/article/pii/S030439759700217X.

[5] Julian Bradfield and Colin Stirling. "12 Modal mu-calculi". In: *Handbook of Modal Logic*. Ed. by Patrick Blackburn, Johan Van Benthem, and Frank Wolter. Vol. 3. Studies in Logic and Practical Reasoning. Elsevier, 2007, pp. 721–756. DOI: https://doi.org/10.1016/S1570-2464(07)80015-2. URL: https://www.sciencedirect.com/science/article/pii/S1570246407800152.

[6] Olav Bunte et al. "The mCRL2 Toolset for Analysing Concurrent Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tomáš Vojnar and Lijun Zhang. Cham: Springer International Publishing, 2019, pp. 21–39. ISBN: 978-3-030-17465-1.

[7] Hana Chockler, Orna Kupferman, and Moshe Y Vardi. "Coverage metrics for formal verification". In: *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer. 2003, pp. 111–125.

[8] Alessandro Cimatti et al. "NUSMV: A New Symbolic Model Verifier". In: *Proceedings of the 11th International Conference on Computer Aided Verification*. CAV '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 495–499. ISBN: 3540662022.

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. ISBN: 3-540-21299-X.

[10] Edmund M Clarke and E Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Workshop on logic of programs*. Springer. 1981, pp. 52–71.

[11] Edmund M Clarke et al. "Model checking and the state explosion problem". In: *LASER Summer School on Software Engineering*. Springer, 2011, pp. 1–30.

[12] Sjoerd Cranen. "Model Checking the FlexRay Startup Phase". In: *Formal Methods for Industrial Critical Systems*. Ed. by Mariëlle Stoelinga and Ralf Pinger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 131–145. ISBN: 978-3-642-32469-7.

[13] Sjoerd Cranen, Jan Friso Groote, and Michel Reniers. "A linear translation from CTL* to the first-order modal mu-calculus". In: *Theoretical Computer Science* 412.28 (2011). Festschrift in Honour of Jan Bergstra, pp. 3129–3139. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2011.02.034. URL: https://www.sciencedirect.com/science/article/pii/S0304397511001587.

[14] Eelco Dolstra and the Nixpkgs/NixOS contributors. *nixpkgs*. 2003. URL: https://github.com/NixOS/nixpkgs (visited on 06/20/2026).

[15] E Allen Emerson. "Model Checking and the Mu-calculus." In: *Descriptive Complexity and Finite Models* 31 (1996), pp. 185–214.

[16] E. Allen Emerson and Joseph Y. Halpern. ""Sometimes" and "not never" revisited: on branching versus linear time (preliminary report)". In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '83. Austin, Texas: Association for Computing Machinery, 1983, pp. 127–140. ISBN: 0897910907. DOI: 10.1145/567067.567081. URL: https://doi.org/10.1145/567067.567081.

[17] Jan Friso Groote and M Mousavi. *Modelling and analysis of communicating systems*. Technische Universiteit Eindhoven, 2013.

[18] Lauri Hella and Jouko Väänänen. *The Size of a Formula as a Measure of Complexity*. 2015.

[19] Orna Kupferman. "Sanity checks in formal verification". In: *International Conference on Concurrency Theory*. Springer. 2006, pp. 37–51.

[20] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. "Reproducibility of Build Environments through Space and Time". In: *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER'24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 97–101. ISBN: 9798400705007. DOI: 10.1145/3639476.3639767. URL: https://doi.org/10.1145/3639476.3639767.

[21] Thomas J McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[22] Thomas Neele, Tim A. C. Willemse, and Jan Friso Groote. "Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting". In: *Formal Aspects of Component Software*. Ed. by Kyungmin Bae and Peter Csaba Ölveczky. Cham: Springer International Publishing, 2018, pp. 216–236. ISBN: 978-3-030-02146-7.

[23] Keishi Okamoto. "Comparing expressiveness of first-order modal μ-calculus and first-order CTL*". In: *RIMS Kokyuroku* 1708 (2010), pp. 1–14.

[24] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[25] *Routing Information Protocol*. RFC 1058. June 1988. DOI: 10.17487/RFC1058. URL: https://www.rfc-editor.org/info/rfc1058.

[26] Myrthe Spronck, Bas Luttik, and Tim Willemse. "Progress, Justness and Fairness in Modal mu-Calculus Formulae". In: *arXiv preprint arXiv:2407.08060* (2024).

[27] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: (1955).

[28] Technische Universiteit Eindhoven. *mCRL2 202507.0 documentation*. 2011. URL: https://mcrl2.org/web/index.html (visited on 06/20/2026).

[29] Michael Zhivich and Robert K. Cunningham. "The Real Cost of Software Errors". In: *IEEE Security and Privacy* 7.2 (2009), pp. 87–90. DOI: 10.1109/MSP.2009.56.

A mCRL2 Specification for the Protocols

A.1 Protocol 1

```
sort
    Nbs = List(Nat);
    Table = List(Int);
    Graph = List(List(Bool));
map
    edge: Graph # Nat # Nat -> Bool;

    wellformed: Graph # Nat -> Bool;
    noloops: Graph # Nat -> Bool;
    bidirectional: Graph # Nat -> Bool;

    valid: Graph # Nat -> Bool;

    neighbours: Graph # Nat -> Nbs;
    neighbours_rec: List(Bool) # Nat -> Nbs;

    inittable: Nat # Nat -> Table;
    inittable_rec: Nat # Nat # Nat -> Table;
    updatetable: Table # Table # Int -> Table;

    reachable: Graph # Nat # Nat -> Bool;
    reachable_rec: Graph # FSet(Nat) # List(Nat) # List(Nat) # Nat -> Bool;
    distance: Graph # Nat # Nat -> Nat;
    distance_rec: Graph # FSet(Nat) # List(Nat) # List(Nat) # Nat # Nat -> Nat;

    reachable_route: List(Table) # Nat # Nat -> Bool;
    reachable_route_rec: List(Table) # FSet(Nat) # Nat # Nat -> Bool;
    distance_route: List(Table) # Nat # Nat -> Nat;
    distance_route_rec: List(Table) # FSet(Nat) # Nat # Nat # Nat -> Nat;

    N : Nat;
var
    graph: Graph;
    size, n1, n2: Nat;
eqn
    edge(graph, n1, n2) = graph . n1 . n2;

    wellformed(graph, size) = #graph == size && (forall ind: Nat . ind < size => #(graph
    ↪  . ind) == size);
    noloops(graph, size) = (forall n: Nat . n < size => !edge(graph, n, n));
    bidirectional(graph, size) = forall n1: Nat . n1 < size => (forall n2: Nat . n2 <
    ↪  size => edge(graph, n1, n2) == edge(graph, n2, n1));
```

```
    valid(graph, size) = wellformed(graph, size) && noloops(graph, size) &&
    ↪  bidirectional(graph, size);
var
    graph: Graph;
    n, size, ind: Nat;

    h: Bool;
    t: List(Bool);

    h1, h2, nb: Int;
    t1, t2: List(Int);
eqn
    neighbours(graph, n) = neighbours_rec(graph . n, 0);
    neighbours_rec([], ind) = [];
    !h -> neighbours_rec(h |> t, ind) = neighbours_rec(t, ind + 1);
    h -> neighbours_rec(h |> t, ind) = ind |> neighbours_rec(t, ind + 1);

    inittable(n, size) = inittable_rec(n, size, 0);
    ind == size -> inittable_rec(n, size, ind) = [];
    ind == n -> inittable_rec(n, size, ind) = n |> inittable_rec(n, size, ind + 1);
    ind < size && ind != n -> inittable_rec(n, size, ind) = -1 |> inittable_rec(n, size,
    ↪  ind + 1);

    updatetable([], [], nb) = [];
    h1 >= 0 || h2 < 0 -> updatetable(h1 |> t1, h2 |> t2, nb) = h1 |> updatetable(t1, t2,
    ↪  nb);
    h1 < 0 && h2 >= 0 -> updatetable(h1 |> t1, h2 |> t2, nb) = nb |> updatetable(t1, t2,
    ↪  nb);
var
    graph: Graph;
    s, e, nd, d: Nat;
    visited: FSet(Nat);
    queue, discover: List(Nat);

    tables: List(Table);
eqn
    reachable(graph, s, e) = reachable_rec(graph, {}, [s], [], e);
    reachable_rec(graph, visited, [], [], e) = false;
    discover != [] -> reachable_rec(graph, visited, [], discover, e) =
    ↪  reachable_rec(graph, visited, discover, [], e);
    nd == e -> reachable_rec(graph, visited, nd |> queue, discover, e) = true;
    nd != e && nd in visited -> reachable_rec(graph, visited, nd |> queue, discover, e)
    ↪  = reachable_rec(graph, visited, queue, discover, e);
    nd != e && !(nd in visited) -> reachable_rec(graph, visited, nd |> queue, discover,
    ↪  e) = reachable_rec(graph, visited + {nd}, queue, neighbours(graph, nd) ++
    ↪  discover, e);

    distance(graph, s, e) = distance_rec(graph, {}, [s], [], e, 0);
    distance_rec(graph, visited, [], [], e, d) = d;
    discover != [] -> distance_rec(graph, visited, [], discover, e, d) =
    ↪  distance_rec(graph, visited, discover, [], e, d + 1);
    nd == e -> distance_rec(graph, visited, nd |> queue, discover, e, d) = d;
    nd != e && nd in visited -> distance_rec(graph, visited, nd |> queue, discover, e,
    ↪  d) = distance_rec(graph, visited, queue, discover, e, d);
```

```
    nd != e && !(nd in visited) -> distance_rec(graph, visited, nd |> queue, discover,
    ↪  e, d) = distance_rec(graph, visited + {nd}, queue, neighbours(graph, nd) ++
    ↪  discover, e, d);

    reachable_route(tables, s, e) = reachable_route_rec(tables, {}, s, e);
    s in visited -> reachable_route_rec(tables, visited, s, e) = false;
    !(s in visited) && s == e -> reachable_route_rec(tables, visited, s, e) = true;
    !(s in visited) && s != e && tables . s . e < 0 -> reachable_route_rec(tables,
    ↪  visited, s, e) = false;
    !(s in visited) && s != e && tables . s . e >= 0 -> reachable_route_rec(tables,
    ↪  visited, s, e) = reachable_route_rec(tables, visited + {s}, Int2Nat(tables . s .
    ↪  e), e);

    distance_route(tables, s, e) = distance_route_rec(tables, {}, s, e, 0);
    s in visited -> distance_route_rec(tables, visited, s, e, d) = d;
    !(s in visited) && s == e -> distance_route_rec(tables, visited, s, e, d) = d;
    !(s in visited) && s != e && tables . s . e < 0 -> distance_route_rec(tables,
    ↪  visited, s, e, d) = d;
    !(s in visited) && s != e && tables . s . e >= 0 -> distance_route_rec(tables,
    ↪  visited, s, e, d) = distance_route_rec(tables, visited + {s}, Int2Nat(tables . s
    ↪  . e), e, d + 1);
eqn
    N = 5;
act
    top_send, top_recieve0, top_recieve1, top_recieve2, top_recieve3, top_recieve4,
    ↪  top_comm : Graph;
    exchange, transfer: Nat # Table # Nat # Table;
    route: Nat # Table;
proc
    Network = sum graph: Graph . valid(graph, N) -> top_send(graph);
    Init0 = sum graph: Graph . top_recieve0(graph) . Node(graph, 0);
    Init1 = sum graph: Graph . top_recieve1(graph) . Node(graph, 1);
    Init2 = sum graph: Graph . top_recieve2(graph) . Node(graph, 2);
    Init3 = sum graph: Graph . top_recieve3(graph) . Node(graph, 3);
    Init4 = sum graph: Graph . top_recieve4(graph) . Node(graph, 4);
    Node(graph: Graph, id: Nat) = Calculate(id, neighbours(graph, id), 0, 1,
    ↪  inittable(id, N), inittable(id, N));
    Calculate(id: Nat, nbs: Nbs, next: Nat, itr: Nat, table: Table, tableAcc: Table) =
        (#nbs == 0) -> Route(id, table) <>
        (itr >= N) -> Route(id, table) <>
        sum othertable: Table . ((id < nbs . next) -> exchange(id, table, nbs . next,
        ↪  othertable) <> exchange(nbs . next, othertable, id, table)) .
        ((next + 1 < #nbs) -> Calculate(id, nbs, next + 1, itr, table,
        ↪  updatetable(tableAcc, othertable, nbs . next)) <> Calculate(id, nbs, 0, itr
        ↪  + 1, updatetable(tableAcc, othertable, nbs . next), updatetable(tableAcc,
        ↪  othertable, nbs . next)));
    Route(id : Nat, table: Table) = route(id, table) . Route(id, table);
init
    allow(
        { top_comm, route, transfer },
    comm(
        {
            top_send|top_recieve0|top_recieve1|top_recieve2|top_recieve3|top_recieve4 ->
            ↪  top_comm,
```

```
            exchange | exchange -> transfer
        },
        Network || Init0 || Init1 || Init2 || Init3 || Init4
    ) ) ;
```

A.2 Protocol 2

```
sort
    Nbs = List(Nat);
    Table = List(Int);
    Graph = List(List(Bool));
map
    edge: Graph # Nat # Nat -> Bool;

    wellformed: Graph # Nat -> Bool;
    noloops: Graph # Nat -> Bool;
    bidirectional: Graph # Nat -> Bool;

    valid: Graph # Nat -> Bool;

    neighbours: Graph # Nat -> Nbs;
    neighbours_rec: List(Bool) # Nat -> Nbs;

    inittable: Nat # Nat -> Table;
    inittable_rec: Nat # Nat # Nat -> Table;
    updatetable: Table # Table # Int -> Table;

    reachable: Graph # Nat # Nat -> Bool;
    reachable_rec: Graph # FSet(Nat) # List(Nat) # List(Nat) # Nat -> Bool;
    distance: Graph # Nat # Nat -> Nat;
    distance_rec: Graph # FSet(Nat) # List(Nat) # List(Nat) # Nat # Nat -> Nat;

    reachable_route: List(Table) # Nat # Nat -> Bool;
    reachable_route_rec: List(Table) # FSet(Nat) # Nat # Nat -> Bool;
    distance_route: List(Table) # Nat # Nat -> Nat;
    distance_route_rec: List(Table) # FSet(Nat) # Nat # Nat # Nat -> Nat;

    N : Nat;
var
    graph: Graph;
    size, n1, n2: Nat;
eqn
    edge(graph, n1, n2) = graph . n1 . n2;

    wellformed(graph, size) = #graph == size && (forall ind: Nat . ind < size => #(graph
    ↪  . ind) == size);
    noloops(graph, size) = (forall n: Nat . n < size => !edge(graph, n, n));
    bidirectional(graph, size) = forall n1: Nat . n1 < size => (forall n2: Nat . n2 <
    ↪  size => edge(graph, n1, n2) == edge(graph, n2, n1));

    valid(graph, size) = wellformed(graph, size) && noloops(graph, size) &&
    ↪  bidirectional(graph, size);
var
    graph: Graph;
    n, size, ind: Nat;
```

```
    h: Bool;
    t: List(Bool);

    h1, h2, nb: Int;
    t1, t2: List(Int);
eqn
    neighbours(graph, n) = neighbours_rec(graph . n, 0);
    neighbours_rec([], ind) = [];
    !h -> neighbours_rec(h |> t, ind) = neighbours_rec(t, ind + 1);
    h -> neighbours_rec(h |> t, ind) = ind |> neighbours_rec(t, ind + 1);

    inittable(n, size) = inittable_rec(n, size, 0);
    ind == size -> inittable_rec(n, size, ind) = [];
    ind == n -> inittable_rec(n, size, ind) = n |> inittable_rec(n, size, ind + 1);
    ind < size && ind != n -> inittable_rec(n, size, ind) = -1 |> inittable_rec(n, size,
    ↪  ind + 1);

    updatetable([], [], nb) = [];
    h1 >= 0 || h2 < 0 -> updatetable(h1 |> t1, h2 |> t2, nb) = h1 |> updatetable(t1, t2,
    ↪  nb);
    h1 < 0 && h2 >= 0 -> updatetable(h1 |> t1, h2 |> t2, nb) = nb |> updatetable(t1, t2,
    ↪  nb);
var
    graph: Graph;
    s, e, nd, d: Nat;
    visited: FSet(Nat);
    queue, discover: List(Nat);

    tables: List(Table);
eqn
    reachable(graph, s, e) = reachable_rec(graph, {}, [s], [], e);
    reachable_rec(graph, visited, [], [], e) = false;
    discover != [] -> reachable_rec(graph, visited, [], discover, e) =
    ↪  reachable_rec(graph, visited, discover, [], e);
    nd == e -> reachable_rec(graph, visited, nd |> queue, discover, e) = true;
    nd != e && nd in visited -> reachable_rec(graph, visited, nd |> queue, discover, e)
    ↪  = reachable_rec(graph, visited, queue, discover, e);
    nd != e && !(nd in visited) -> reachable_rec(graph, visited, nd |> queue, discover,
    ↪  e) = reachable_rec(graph, visited + {nd}, queue, neighbours(graph, nd) ++
    ↪  discover, e);

    distance(graph, s, e) = distance_rec(graph, {}, [s], [], e, 0);
    distance_rec(graph, visited, [], [], e, d) = d;
    discover != [] -> distance_rec(graph, visited, [], discover, e, d) =
    ↪  distance_rec(graph, visited, discover, [], e, d + 1);
    nd == e -> distance_rec(graph, visited, nd |> queue, discover, e, d) = d;
    nd != e && nd in visited -> distance_rec(graph, visited, nd |> queue, discover, e,
    ↪  d) = distance_rec(graph, visited, queue, discover, e, d);
    nd != e && !(nd in visited) -> distance_rec(graph, visited, nd |> queue, discover,
    ↪  e, d) = distance_rec(graph, visited + {nd}, queue, neighbours(graph, nd) ++
    ↪  discover, e, d);

    reachable_route(tables, s, e) = reachable_route_rec(tables, {}, s, e);
```

```
    s in visited -> reachable_route_rec(tables, visited, s, e) = false;
    !(s in visited) && s == e -> reachable_route_rec(tables, visited, s, e) = true;
    !(s in visited) && s != e && tables . s . e < 0 -> reachable_route_rec(tables,
    ↪  visited, s, e) = false;
    !(s in visited) && s != e && tables . s . e >= 0 -> reachable_route_rec(tables,
    ↪  visited, s, e) = reachable_route_rec(tables, visited + {s}, Int2Nat(tables . s .
    ↪  e), e);

    distance_route(tables, s, e) = distance_route_rec(tables, {}, s, e, 0);
    s in visited -> distance_route_rec(tables, visited, s, e, d) = d;
    !(s in visited) && s == e -> distance_route_rec(tables, visited, s, e, d) = d;
    !(s in visited) && s != e && tables . s . e < 0 -> distance_route_rec(tables,
    ↪  visited, s, e, d) = d;
    !(s in visited) && s != e && tables . s . e >= 0 -> distance_route_rec(tables,
    ↪  visited, s, e, d) = distance_route_rec(tables, visited + {s}, Int2Nat(tables . s
    ↪  . e), e, d + 1);
eqn
    N = 5;
act
    top_send, top_recieve0, top_recieve1, top_recieve2, top_recieve3, top_recieve4,
    ↪  top_comm : Graph;
    exchange, transfer: Nat # Table # Nat # Table;
    route: Nat # Table;
proc
    Network = sum graph: Graph . valid(graph, N) -> top_send(graph);
    Init0 = sum graph: Graph . top_recieve0(graph) . Node(graph, 0);
    Init1 = sum graph: Graph . top_recieve1(graph) . Node(graph, 1);
    Init2 = sum graph: Graph . top_recieve2(graph) . Node(graph, 2);
    Init3 = sum graph: Graph . top_recieve3(graph) . Node(graph, 3);
    Init4 = sum graph: Graph . top_recieve4(graph) . Node(graph, 4);
    Node(graph: Graph, id: Nat) = Calculate(id, neighbours(graph, id), {}, 1,
    ↪  inittable(id, N), inittable(id, N));
    Calculate(id: Nat, nbs: Nbs, messaged: FSet(Nat), itr: Nat, table: Table, tableAcc:
    ↪  Table) =
        (#nbs == 0) -> Route(id, table) <>
        (itr >= N) -> Route(id, table) <>
        sum other: Nat . (other in nbs && !(other in messaged)) -> (sum othertable:
        ↪  Table . ((id < other) -> exchange(id, table, other, othertable) <>
        ↪  exchange(other, othertable, id, table)) .
        ((#messaged + 1 < #nbs) -> Calculate(id, nbs, messaged + {other}, itr, table,
        ↪  updatetable(tableAcc, othertable, other)) <> Calculate(id, nbs, {}, itr + 1,
        ↪  updatetable(tableAcc, othertable, other), updatetable(tableAcc, othertable,
        ↪  other))));
    Route(id : Nat, table: Table) = route(id, table) . Route(id, table);
init
    allow(
        { top_comm, route, transfer },
    comm(
        {
            top_send|top_recieve0|top_recieve1|top_recieve2|top_recieve3|top_recieve4 ->
            ↪  top_comm,
            exchange | exchange -> transfer
        },
        Network || Init0 || Init1 || Init2 || Init3 || Init4
```

```
    ) ) ;
```

A.3 Protocol 3

```
sort
    Nbs = List(Nat);
    Table = List(Int);
    Graph = List(List(Bool));
map
    edge: Graph # Nat # Nat -> Bool;

    wellformed: Graph # Nat -> Bool;
    noloops: Graph # Nat -> Bool;
    bidirectional: Graph # Nat -> Bool;

    valid: Graph # Nat -> Bool;

    neighbours: Graph # Nat -> Nbs;
    neighbours_rec: List(Bool) # Nat -> Nbs;

    inittable: Nat # Nat -> Table;
    inittable_rec: Nat # Nat # Nat -> Table;
    updatetable: Table # Table # Int -> Table;

    reachable: Graph # Nat # Nat -> Bool;
    reachable_rec: Graph # FSet(Nat) # List(Nat) # List(Nat) # Nat -> Bool;
    distance: Graph # Nat # Nat -> Nat;
    distance_rec: Graph # FSet(Nat) # List(Nat) # List(Nat) # Nat # Nat -> Nat;

    reachable_route: List(Table) # Nat # Nat -> Bool;
    reachable_route_rec: List(Table) # FSet(Nat) # Nat # Nat -> Bool;
    distance_route: List(Table) # Nat # Nat -> Nat;
    distance_route_rec: List(Table) # FSet(Nat) # Nat # Nat # Nat -> Nat;

    N : Nat;
var
    graph: Graph;
    size, n1, n2: Nat;
eqn
    edge(graph, n1, n2) = graph . n1 . n2;

    wellformed(graph, size) = #graph == size && (forall ind: Nat . ind < size => #(graph
    ↪  . ind) == size);
    noloops(graph, size) = (forall n: Nat . n < size => !edge(graph, n, n));
    bidirectional(graph, size) = forall n1: Nat . n1 < size => (forall n2: Nat . n2 <
    ↪  size => edge(graph, n1, n2) == edge(graph, n2, n1));

    valid(graph, size) = wellformed(graph, size) && noloops(graph, size) &&
    ↪  bidirectional(graph, size);
var
    graph: Graph;
    n, size, ind: Nat;

    h: Bool;
    t: List(Bool);
```

```
    h1, h2, nb: Int;
    t1, t2: List(Int);
eqn
    neighbours(graph, n) = neighbours_rec(graph . n, 0);
    neighbours_rec([], ind) = [];
    !h -> neighbours_rec(h |> t, ind) = neighbours_rec(t, ind + 1);
    h -> neighbours_rec(h |> t, ind) = ind |> neighbours_rec(t, ind + 1);

    inittable(n, size) = inittable_rec(n, size, 0);
    ind == size -> inittable_rec(n, size, ind) = [];
    ind == n -> inittable_rec(n, size, ind) = n |> inittable_rec(n, size, ind + 1);
    ind < size && ind != n -> inittable_rec(n, size, ind) = -1 |> inittable_rec(n, size,
    ↪  ind + 1);

    updatetable([], [], nb) = [];
    h1 >= 0 || h2 < 0 -> updatetable(h1 |> t1, h2 |> t2, nb) = h1 |> updatetable(t1, t2,
    ↪  nb);
    h1 < 0 && h2 >= 0 -> updatetable(h1 |> t1, h2 |> t2, nb) = nb |> updatetable(t1, t2,
    ↪  nb);
var
    graph: Graph;
    s, e, nd, d: Nat;
    visited: FSet(Nat);
    queue, discover: List(Nat);

    tables: List(Table);
eqn
    reachable(graph, s, e) = reachable_rec(graph, {}, [s], [], e);
    reachable_rec(graph, visited, [], [], e) = false;
    discover != [] -> reachable_rec(graph, visited, [], discover, e) =
    ↪  reachable_rec(graph, visited, discover, [], e);
    nd == e -> reachable_rec(graph, visited, nd |> queue, discover, e) = true;
    nd != e && nd in visited -> reachable_rec(graph, visited, nd |> queue, discover, e)
    ↪  = reachable_rec(graph, visited, queue, discover, e);
    nd != e && !(nd in visited) -> reachable_rec(graph, visited, nd |> queue, discover,
    ↪  e) = reachable_rec(graph, visited + {nd}, queue, neighbours(graph, nd) ++
    ↪  discover, e);

    distance(graph, s, e) = distance_rec(graph, {}, [s], [], e, 0);
    distance_rec(graph, visited, [], [], e, d) = d;
    discover != [] -> distance_rec(graph, visited, [], discover, e, d) =
    ↪  distance_rec(graph, visited, discover, [], e, d + 1);
    nd == e -> distance_rec(graph, visited, nd |> queue, discover, e, d) = d;
    nd != e && nd in visited -> distance_rec(graph, visited, nd |> queue, discover, e,
    ↪  d) = distance_rec(graph, visited, queue, discover, e, d);
    nd != e && !(nd in visited) -> distance_rec(graph, visited, nd |> queue, discover,
    ↪  e, d) = distance_rec(graph, visited + {nd}, queue, neighbours(graph, nd) ++
    ↪  discover, e, d);

    reachable_route(tables, s, e) = reachable_route_rec(tables, {}, s, e);
    s in visited -> reachable_route_rec(tables, visited, s, e) = false;
    !(s in visited) && s == e -> reachable_route_rec(tables, visited, s, e) = true;
```

```
    !(s in visited) && s != e && tables . s . e < 0 -> reachable_route_rec(tables,
    ↪  visited, s, e) = false;
    !(s in visited) && s != e && tables . s . e >= 0 -> reachable_route_rec(tables,
    ↪  visited, s, e) = reachable_route_rec(tables, visited + {s}, Int2Nat(tables . s .
    ↪  e), e);

    distance_route(tables, s, e) = distance_route_rec(tables, {}, s, e, 0);
    s in visited -> distance_route_rec(tables, visited, s, e, d) = d;
    !(s in visited) && s == e -> distance_route_rec(tables, visited, s, e, d) = d;
    !(s in visited) && s != e && tables . s . e < 0 -> distance_route_rec(tables,
    ↪  visited, s, e, d) = d;
    !(s in visited) && s != e && tables . s . e >= 0 -> distance_route_rec(tables,
    ↪  visited, s, e, d) = distance_route_rec(tables, visited + {s}, Int2Nat(tables . s
    ↪  . e), e, d + 1);
eqn
    N = 5;
act
    top_send, top_recieve0, top_recieve1, top_recieve2, top_recieve3, top_recieve4,
    ↪  top_comm : Graph;
    exchange, transfer: Nat # Table # Nat # Table;
    route: Nat # Table;
proc
    Network = sum graph: Graph . valid(graph, N) -> top_send(graph);
    Init0 = sum graph: Graph . top_recieve0(graph) . Node(graph, 0);
    Init1 = sum graph: Graph . top_recieve1(graph) . Node(graph, 1);
    Init2 = sum graph: Graph . top_recieve2(graph) . Node(graph, 2);
    Init3 = sum graph: Graph . top_recieve3(graph) . Node(graph, 3);
    Init4 = sum graph: Graph . top_recieve4(graph) . Node(graph, 4);
    Node(graph: Graph, id: Nat) = Calculate(id, neighbours(graph, id), 0, 1,
    ↪  inittable(id, N));
    Calculate(id: Nat, nbs: Nbs, next: Nat, itr: Nat, table: Table) =
        (#nbs == 0) -> Route(id, table) <>
        (itr >= N) -> Route(id, table) <>
        sum othertable: Table . ((id < nbs . next) -> exchange(id, table, nbs . next,
        ↪  othertable) <> exchange(nbs . next, othertable, id, table)) .
        ((next + 1 < #nbs) -> Calculate(id, nbs, next + 1, itr, updatetable(table,
        ↪  othertable, nbs . next)) <> Calculate(id, nbs, 0, itr + 1,
        ↪  updatetable(table, othertable, nbs . next)));
    Route(id : Nat, table: Table) = route(id, table) . Route(id, table);
init
    allow(
        { top_comm, route, transfer },
    comm(
        {
            top_send|top_recieve0|top_recieve1|top_recieve2|top_recieve3|top_recieve4 ->
            ↪  top_comm,
            exchange | exchange -> transfer
        },
        Network || Init0 || Init1 || Init2 || Init3 || Init4
    ) ) ;
```

A.4 Protocol 4

```
sort
```

```
    Nbs = List(Nat);
    Table = List(Int);
    Graph = List(List(Bool));
map
    edge: Graph # Nat # Nat -> Bool;

    wellformed: Graph # Nat -> Bool;
    noloops: Graph # Nat -> Bool;
    bidirectional: Graph # Nat -> Bool;

    valid: Graph # Nat -> Bool;

    neighbours: Graph # Nat -> Nbs;
    neighbours_rec: List(Bool) # Nat -> Nbs;

    inittable: Nat # Nat -> Table;
    inittable_rec: Nat # Nat # Nat -> Table;
    updatetable: Table # Table # Int -> Table;

    reachable: Graph # Nat # Nat -> Bool;
    reachable_rec: Graph # FSet(Nat) # List(Nat) # List(Nat) # Nat -> Bool;
    distance: Graph # Nat # Nat -> Nat;
    distance_rec: Graph # FSet(Nat) # List(Nat) # List(Nat) # Nat # Nat -> Nat;

    reachable_route: List(Table) # Nat # Nat -> Bool;
    reachable_route_rec: List(Table) # FSet(Nat) # Nat # Nat -> Bool;
    distance_route: List(Table) # Nat # Nat -> Nat;
    distance_route_rec: List(Table) # FSet(Nat) # Nat # Nat # Nat -> Nat;

    N : Nat;
var
    graph: Graph;
    size, n1, n2: Nat;
eqn
    edge(graph, n1, n2) = graph . n1 . n2;

    wellformed(graph, size) = #graph == size && (forall ind: Nat . ind < size => #(graph
    ↪  . ind) == size);
    noloops(graph, size) = (forall n: Nat . n < size => !edge(graph, n, n));
    bidirectional(graph, size) = forall n1: Nat . n1 < size => (forall n2: Nat . n2 <
    ↪  size => edge(graph, n1, n2) == edge(graph, n2, n1));

    valid(graph, size) = wellformed(graph, size) && noloops(graph, size) &&
    ↪  bidirectional(graph, size);
var
    graph: Graph;
    n, size, ind: Nat;

    h: Bool;
    t: List(Bool);

    h1, h2, nb: Int;
    t1, t2: List(Int);
eqn
```

```
    neighbours(graph, n) = neighbours_rec(graph . n, 0);
    neighbours_rec([], ind) = [];
    !h -> neighbours_rec(h |> t, ind) = neighbours_rec(t, ind + 1);
    h -> neighbours_rec(h |> t, ind) = ind |> neighbours_rec(t, ind + 1);

    inittable(n, size) = inittable_rec(n, size, 0);
    ind == size -> inittable_rec(n, size, ind) = [];
    ind == n -> inittable_rec(n, size, ind) = n |> inittable_rec(n, size, ind + 1);
    ind < size && ind != n -> inittable_rec(n, size, ind) = -1 |> inittable_rec(n, size,
    ↪  ind + 1);

    updatetable([], [], nb) = [];
    h1 >= 0 || h2 < 0 -> updatetable(h1 |> t1, h2 |> t2, nb) = h1 |> updatetable(t1, t2,
    ↪  nb);
    h1 < 0 && h2 >= 0 -> updatetable(h1 |> t1, h2 |> t2, nb) = nb |> updatetable(t1, t2,
    ↪  nb);
var
    graph: Graph;
    s, e, nd, d: Nat;
    visited: FSet(Nat);
    queue, discover: List(Nat);

    tables: List(Table);
eqn
    reachable(graph, s, e) = reachable_rec(graph, {}, [s], [], e);
    reachable_rec(graph, visited, [], [], e) = false;
    discover != [] -> reachable_rec(graph, visited, [], discover, e) =
    ↪  reachable_rec(graph, visited, discover, [], e);
    nd == e -> reachable_rec(graph, visited, nd |> queue, discover, e) = true;
    nd != e && nd in visited -> reachable_rec(graph, visited, nd |> queue, discover, e)
    ↪  = reachable_rec(graph, visited, queue, discover, e);
    nd != e && !(nd in visited) -> reachable_rec(graph, visited, nd |> queue, discover,
    ↪  e) = reachable_rec(graph, visited + {nd}, queue, neighbours(graph, nd) ++
    ↪  discover, e);

    distance(graph, s, e) = distance_rec(graph, {}, [s], [], e, 0);
    distance_rec(graph, visited, [], [], e, d) = d;
    discover != [] -> distance_rec(graph, visited, [], discover, e, d) =
    ↪  distance_rec(graph, visited, discover, [], e, d + 1);
    nd == e -> distance_rec(graph, visited, nd |> queue, discover, e, d) = d;
    nd != e && nd in visited -> distance_rec(graph, visited, nd |> queue, discover, e,
    ↪  d) = distance_rec(graph, visited, queue, discover, e, d);
    nd != e && !(nd in visited) -> distance_rec(graph, visited, nd |> queue, discover,
    ↪  e, d) = distance_rec(graph, visited + {nd}, queue, neighbours(graph, nd) ++
    ↪  discover, e, d);

    reachable_route(tables, s, e) = reachable_route_rec(tables, {}, s, e);
    s in visited -> reachable_route_rec(tables, visited, s, e) = false;
    !(s in visited) && s == e -> reachable_route_rec(tables, visited, s, e) = true;
    !(s in visited) && s != e && tables . s . e < 0 -> reachable_route_rec(tables,
    ↪  visited, s, e) = false;
    !(s in visited) && s != e && tables . s . e >= 0 -> reachable_route_rec(tables,
    ↪  visited, s, e) = reachable_route_rec(tables, visited + {s}, Int2Nat(tables . s .
    ↪  e), e);
```

```
    distance_route(tables, s, e) = distance_route_rec(tables, {}, s, e, 0);
    s in visited -> distance_route_rec(tables, visited, s, e, d) = d;
    !(s in visited) && s == e -> distance_route_rec(tables, visited, s, e, d) = d;
    !(s in visited) && s != e && tables . s . e < 0 -> distance_route_rec(tables,
    ↪  visited, s, e, d) = d;
    !(s in visited) && s != e && tables . s . e >= 0 -> distance_route_rec(tables,
    ↪  visited, s, e, d) = distance_route_rec(tables, visited + {s}, Int2Nat(tables . s
    ↪  . e), e, d + 1);
eqn
    N = 5;
act
    top_send, top_recieve0, top_recieve1, top_recieve2, top_recieve3, top_recieve4,
    ↪  top_comm : Graph;
    exchange, transfer: Nat # Table # Nat # Table;
    route: Nat # Table;
proc
    Network = sum graph: Graph . valid(graph, N) -> top_send(graph);
    Init0 = sum graph: Graph . top_recieve0(graph) . Node(graph, 0);
    Init1 = sum graph: Graph . top_recieve1(graph) . Node(graph, 1);
    Init2 = sum graph: Graph . top_recieve2(graph) . Node(graph, 2);
    Init3 = sum graph: Graph . top_recieve3(graph) . Node(graph, 3);
    Init4 = sum graph: Graph . top_recieve4(graph) . Node(graph, 4);
    Node(graph: Graph, id: Nat) = Calculate(id, neighbours(graph, id), 0, 1,
    ↪  inittable(id, N), inittable(id, N));
    Calculate(id: Nat, nbs: Nbs, next: Nat, itr: Nat, table: Table, tableAcc: Table) =
        (#nbs == 0) -> Route(id, table) <>
        (itr >= 3) -> Route(id, table) <>
        sum othertable: Table . ((id < nbs . next) -> exchange(id, table, nbs . next,
        ↪  othertable) <> exchange(nbs . next, othertable, id, table)) .
        ((next + 1 < #nbs) -> Calculate(id, nbs, next + 1, itr, table,
        ↪  updatetable(tableAcc, othertable, nbs . next)) <> Calculate(id, nbs, 0, itr
        ↪  + 1, updatetable(tableAcc, othertable, nbs . next), updatetable(tableAcc,
        ↪  othertable, nbs . next)));
    Route(id : Nat, table: Table) = route(id, table) . Route(id, table);
init
    allow(
        { top_comm, route, transfer },
    comm(
        {
            top_send|top_recieve0|top_recieve1|top_recieve2|top_recieve3|top_recieve4 ->
            ↪  top_comm,
            exchange | exchange -> transfer
        },
        Network || Init0 || Init1 || Init2 || Init3 || Init4
    ) ) ;
```

B Modal μ-Calculus Expressions for the Properties

B.1 Reachability Property (Fixed-Point)

```
forall n1: Nat . val(n1 < N) => forall n2: Nat . val(n2 < N) => forall graph: Graph .
[top_comm(graph)]
```

```
((mu X(nr : Nat = n1) . val(nr == n2) || exists n3: Nat . val(n3 < N && edge(graph, nr,
↪  n3)) && X(n3)) =>
mu X(nr: Nat = n1) .
    val(nr == n2) ||
    [forall id: Nat, table: Table . !route(id, table)]X(nr) &&
    (forall n3: Nat . val(n3 < N) =>
        [exists table: Table . route(nr, table) && val(table . n2 == Nat2Int(n3))]X(n3))
        ↪  &&
        [exists table: Table . route(nr, table) && val(table . n2 < 0)]false)
```

B.2 Reachability Property (Helper function)

```
forall n1: Nat . val(n1 < N) => forall n2: Nat . val(n2 < N) => forall graph: Graph .
[top_comm(graph) . true*]
forall tb0: Table . [route(0, tb0)]
forall tb1: Table . [route(1, tb1)]
forall tb2: Table . [route(2, tb2)]
forall tb3: Table . [route(3, tb3)]
forall tb4: Table . [route(4, tb4)]
(val(reachable(graph, n1, n2) => reachable_route([tb0, tb1, tb2, tb3, tb4], n1, n2)))
```

B.3 Distance Property (Fixed-Point)

```
forall n1: Nat . val(n1 < N) => forall n2: Nat . val(n2 < N) => forall graph: Graph .
[top_comm(graph)]
((mu X(nr : Nat = n1) . val(nr == n2) || exists n3: Nat . val(n3 < N && edge(graph, nr,
↪  n3)) && X(n3)) =>
mu X(nr: Nat = n1, d: Nat = 0) .
    val(nr == n2 || nr == -1) ||
    val(d < Int2Nat(N - 1)) &&
    [forall id: Nat, table: Table . !route(id, table)]X(nr, d) &&
    (forall n3: Nat . val(n3 < N) =>
        [exists table: Table . route(nr, table) && val(table . n2 == Nat2Int(n3))]X(n3,
        ↪  d + 1)))
```

B.4 Distance Property (Helper function)

```
forall n1: Nat . val(n1 < N) => forall n2: Nat . val(n2 < N) => forall graph: Graph .
[top_comm(graph) . true*]
forall tb0: Table . [route(0, tb0)]
forall tb1: Table . [route(1, tb1)]
forall tb2: Table . [route(2, tb2)]
forall tb3: Table . [route(3, tb3)]
forall tb4: Table . [route(4, tb4)]
(val(reachable_route([tb0, tb1, tb2, tb3, tb4], n1, n2) => distance_route([tb0, tb1,
↪  tb2, tb3, tb4], n1, n2) < N))
```

B.5 Shortest Path Property (Fixed-Point)

```
forall n1: Nat . val(n1 < N) => forall n2: Nat. val(n2 < N) => forall graph: Graph .
[top_comm(graph)]
((mu X(nr : Nat = n1) . val(nr == n2) || exists n3: Nat . val(n3 < N && edge(graph, nr,
↪  n3)) && X(n3)) =>
(mu X(nr: Nat = n1, no: Nat = n1) .
    val(nr == n2 || nr == -1) ||
```

```
    val(no != n2) &&
    [forall id: Nat, table: Table . !route(id, table)]X(nr, no) &&
    (forall n3: Nat . val(n3 < N) => forall n4: Nat . val(n4 < N && edge(graph, no, n4))
    ↪  =>
        [exists table: Table . route(nr, table) && val(table . n2 == Nat2Int(n3))]X(n3,
        ↪  n4))))
```

B.6 Shortest Path Property (Helper function)

```
forall n1: Nat . val(n1 < N) => forall n2: Nat . val(n2 < N) => forall graph: Graph .
[top_comm(graph) . true*]
forall tb0: Table . [route(0, tb0)]
forall tb1: Table . [route(1, tb1)]
forall tb2: Table . [route(2, tb2)]
forall tb3: Table . [route(3, tb3)]
forall tb4: Table . [route(4, tb4)]
(val(reachable_route([tb0, tb1, tb2, tb3, tb4], n1, n2) => distance_route([tb0, tb1,
↪  tb2, tb3, tb4], n1, n2) <= distance(graph, n1, n2)))
```

C The benchmarking script

```
#!/usr/bin/env zsh

TIMEFMT=$'%J %MKB memory %U user %S system %P cpu %E total\a'

for size in 3 4 5; do
    for mcrl2 in mcrl2/*.mcrl2; do
        mcrl22lps =(sed "s/N = 5/N = $size/" "$mcrl2" | sed "s/|top_recieve[$size-5]//g"
        ↪  | sed "s/|| Init[$size-5]//g") tmp.lps
        for prop in mu/*.mu; do
            lps2pbes -f =(sed "s/, tb[$size-5]//g" "$prop" | sed "/tb[$size-5]/d")
            ↪  tmp.lps tmp.pbes
            echo "$size" "$mcrl2" "$prop"
            repeat 3; do
                echo 3 | sudo tee /proc/sys/vm/drop_caches
                time timeout 1h pbessolve --threads=8 -rjittyc tmp.pbes
            done
        done
    done
done
```