MODELLING AND REASONING ABOUT DYNAMIC NETWORKS AS CONCURRENT SYSTEMS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2014

By
Yanti Rusmawati
School of Computer Science

Contents

Approximate number of words in this thesis: 53,391

List of Tables

List of Figures

Abstract

Highly dynamic and complex computing systems are increasingly needed and are relied upon in daily life. One such system is the dynamic network, particularly in communication, in which it has widespread applications, such as: Internet, peer-to-peer networks, mobile networks and wireless networks. Dynamic networks consist of nodes and edges whose operating status may change over time; the edges may be unreliable and operate intermittently. Message-passing in such networks is inherently difficult and reasoning about the behaviour of message-passing algorithms is also difficult and hard to analyse. Their behaviour and correctness are hard to formulate and establish. To undertake formal reasoning about such systems, abstract models are essential in order to separate the general reasoning about message routing and the updating of routing tables from the details of how these are implemented in particular networks.

This thesis proposes a new approach to modelling and reasoning about dynamic networks as follows. It develops a series of abstract models which makes it possible to focus on the correctness of routing methods. It models the dynamic network as a "demonic" process which runs concurrently with routing updates and message-passing, to express dynamic networks as concurrent systems. This allows the use of temporal logic and fairness constraints to reason about dynamic networks.

To do so, it introduces a modal logic and formulates concepts of fairness which capture network properties. The correctness of dynamic networks means that under certain conditions, all messages will eventually be delivered. Formulating networks as concurrent systems means can establish the correctness for networks that never cease to change. Modelling at that one level of abstraction means being able to prove the properties of networks independently of the mechanisms in actual networks. Therefore, it provides "a factorisation" of proofs of correctness for actual dynamic networks. The models are implemented as multi-threaded programs, and then adopted an experimental runtime verification tool called RULER to test whether model instances satisfy the modal correctness for message delivery.

Declaration

Copyright

Acknowledgements

First of all, I am grateful to God for the opportunity and inspiration that lead me to complete and present this thesis.

I really would like to thank Dr David Rydeheard as my supervisor, Prof Howard Barringer as my co-supervisor and Prof Ulrike Sattler as my advisor for their support and helpful guidance throughout all stages of this research.

Thanks to my husband Darjana for his great understanding, ultimate support, love and encouragement, and to my kids Vinci, Gagas, and Illio who always cheer me up. Thanks to all my friends and family in UK and Indonesia.

Last but not least, thanks to my country Indonesia and especially The Ministry of Communication and Informatics for supporting and financially aiding my studies throughout this year.

Chapter 1

Introduction

This chapter presents a brief introduction to dynamic network systems. It discusses where they occur, why they need to be reasoned about, and why such reasoning is hard. There are also a summary of achievements, a research hypothesis and a set of research questions, as well as presentation of the structure of this dissertation.

1.1 Problems in dynamic networks

These computing systems are complex and highly dynamic. Such systems are complex regarding their structure and components. They are dynamic regarding their response to change. The latter is required for such systems for being long-life systems. They should have an ability to evolve in response to changes from outside, as well as to changes from inside, without being interrupted. One such type of system is the dynamic network. In communication, dynamic networks are widespread; these include: the Internet, peer-to-peer networks, mobile networks and wireless networks. Such networks may have the links down, the nodes may move, or there might be routing instability due to the changing of networks. Chapter 2 presents more technical explanations.

Communication networks can be modelled as a collection of n nodes connecting to one other through links. Each node communicates by exchanging messages directly or through intermediate nodes. We usually use a graph, which consists of nodes and edges, to model such networks at a high level of abstraction. Each edge may have a cost and the network's routing may use this as a decision mechanism. Networks that are more complex are typically modelled as a hierarchical graph.

There are *static network systems* and *dynamic network systems*. In a *static network* *system*, the number of available nodes and edges (links) are fixed over time, as is their connectivity. Routing in this type of system, such as a private network with reliable connections, can be computed in advance and is predictable [59]. By contrast, the number of nodes and edges in a *dynamic network system* may vary over time, as can the edge cost. For example, a wireless network, which depends heavily on efficient energy consumption and priority, may lose the connection between nodes over time. A node in a mobile network may change its position and its neighbour frequently.

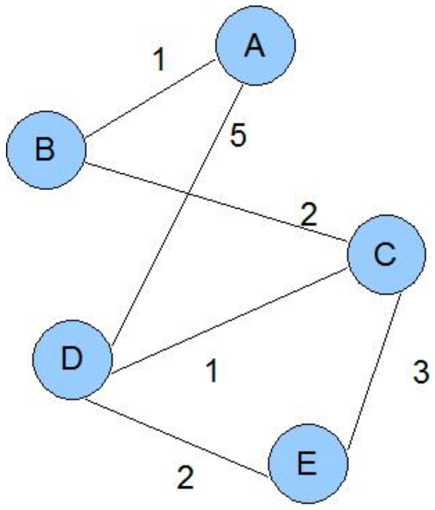


Figure 1.1: An example of a simple network with cost on each link

What is a dynamic network? Suppose we have a computing systems network represented as a graph consisting of five nodes and six edges as links, as shown in Figure 1.1. Each node (labelled A, B, C, D, and E) represents a computing system. Each link has a cost and represents a communication link between nodes. In a static network, each node has the same neighbour nodes over time. There is no change in the communication link between nodes. However, in a dynamic network, each node may have different neighbour nodes over time. Moreover, the communication link between nodes can be *connected* and *disconnected* interchangeably. This can occur due to node movement which changes the distance between nodes (for example in wireless network or mobile ad hoc network); alternatively there may be a disruption over a communication link, such as noise or medium disruption. These changes affect message delivery from one node to another node, especially if the network needs time to recover from the changes.

Message delivery in this type of network uses routing that is unstable over time. Ideally, nodes within the network should know immediately that there is a change of message routing from source to destination due to changes of links or the existence of a node. This is to ensure that a message to be delivered from source to destination uses a *correct routing information* so that message delivery can be guaranteed.

Figure 1.1 shows that each node can exchange a message with another node using a direct path or through the intermediate nodes. Each link has a link cost and can be used as a metric for a routing decision. For example, node A can send a message to D through three alternative paths. These paths are:

1. A, D (directly)
2. A, B, C, D
3. A, B, C, E, D

Node A will use the most optimal or efficient path. The optimality measurement can be based on various metrics, such as distance, number of hops, or the weight of the link. To obtain the best path for sending a message, the system needs a routing algorithm.

Network routing is the capability of a computing network to send a unit of information from one point to another point by determining a path through the network [45]. The determination of an efficient path depends on a number of factors. A routing algorithm determines an efficient path, which results in a form of a route. Routing algorithms in dynamic networks are extended from some basic routing algorithms (such as the Dijkstra routing algorithm [45, pp. 38-42] and the Bellman-Ford routing algorithm [45, pp. 33-38]) according to various types of dynamic networks.

Assume that there are some conditions for the network in Figure 1.1 as follows:

- link A to D going down, or
- link B to C going down when A has sent a message to D via B, or
- link C to D going down when A has sent a message to D via B, or
- link A to D going up again when A has sent a message to D via C, or
- link cost of B and C changes to 4, or
- node D leaves the network when A has sent a message to D via B.

If any of these conditions applies, then how does the network system deal with these conditions, especially network routing? How do we obtain correct routing information in these circumstances?

The concept of correctness in sequential programming [8] has two definitions: *total correctness* and *partial correctness*. Total correctness requires termination of the program to obtain the correct answer to a problem. Partial correctness does not require termination. If the program terminates then the condition holds after termination. Termination in infinite traces, such as in dynamic networks, means that, after some period of time, it will eventually terminate. Afterwards, the connections within the network may continue to change again.

For dynamic networks which have no termination, correctness in concurrent programming is more appropriate. Instead of defining correctness about computing a functional result, correctness is defined about the properties of computations. These correctness properties are: *safety properties* and *liveness properties*. Safety means that the property must always be true. Liveness means that the property must eventually become true. In terms of dynamic networks, each node should always know the correct routing information and a message should eventually reach its destination. However, it is not trivial to achieve this correctness in real dynamic networks.

Why it is not trivial to ensure the correctness of routing information in dynamic networks? Figure 1.2 describes the changes in a dynamic network over time. At time

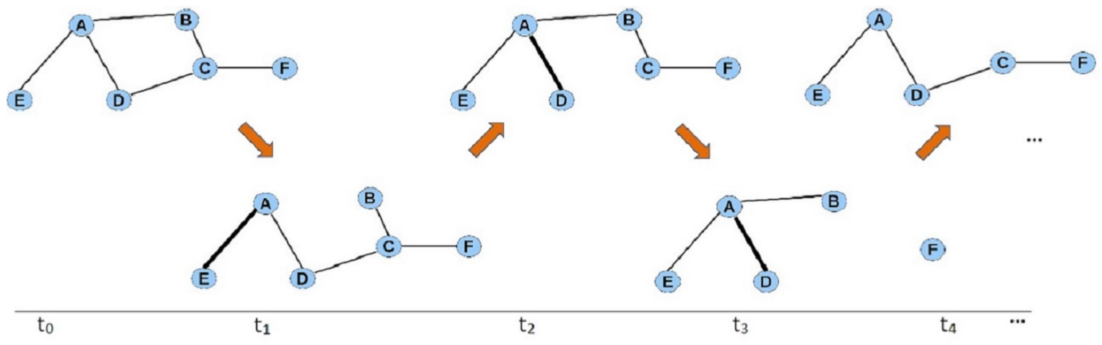


Figure 1.2: A dynamic network changes with respect to time

t_0, a link between node A and node B still exists. Suppose we want to send a message from node E to node F through nodes A, B and C. At time t_1, the message has reached node A; meanwhile the link between node A and B has disappeared. As a response to this change, the message then being delivered to node D at time t_2 is based on knowledge about there existing a route from node A to node F through D and C. However, at the same time, the link between D and C has disappeared. Therefore, the message needs to be delivered using an alternative route. At time t_3, node C has disappeared and the link between A and B has been recovered. As can be observed from Figure 1.2, the network topology at time t_4 changes again and this will change the route information as well. So how do we ensure the correctness of such networks?

Notice that although sometimes there is no path but messages can get through; alternatively there might be a path but messages are unable to get through. As in Figure 1.3, message can possibly be delivered from node A to node C even though there is no path between them at time t_1 and time t_2.

There is a mechanism in a network which keep recalculating route information.

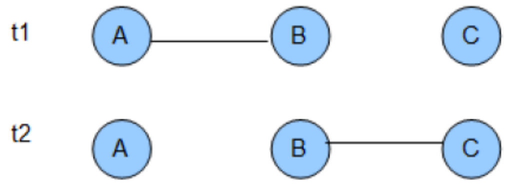


Figure 1.3: An example of message might be delivered with no path

This information is about how to reach one node from another node, and can be kept in a routing table. This recalculation process will work based on the changes that occur in the networks. Therefore, we can figure out that in dynamic networks, the recalculation process works frequently to obtain the routing information as accurately as possible. On the assumption that the network has a correct routing table or information, the message should eventually get delivered. However, this can only happen if the network itself is sufficiently connected for sufficiently often. We cannot place much reliance on the possibility given in Figure 1.3.

Another problem is if there is always a path but the message cannot get through. Consider the following (Figure 1.4). Suppose a message is delivered from node E to node F, and message has been at node A. This message may never get through if one of the paths between node A and node C is broken or disrupted, and if the message does not know that there is an alternative path. The circumstance can occur if a message is overlooked by the message dispatcher or a system in node A.

Therefore, there are two separate aspects of message delivery in dynamic networks that need to be considered: routing and message passing. Furthermore, there can be a *livelock*. Ashcroft [2] introduces this term as the oscillation which continues forever when proving the correctness of parallel program. Ho et al. [28] identify that in the past, the computing literature has used the term “livelock” inconsistently to mean starvation, infinite execution, or simply the failure to maintain liveness. Livelock is the condition when a system is at a standstill since no forward progress is being made.

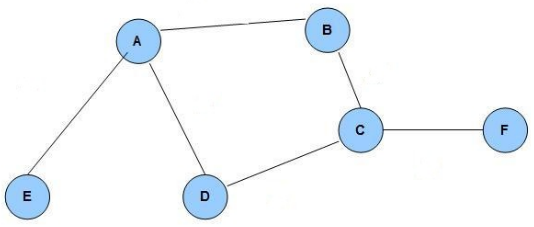


Figure 1.4: An example of a simple network

Forward progress occurs when the progress performs useful computation towards termination or a goal. An infinite program execution exhibits non-progress behaviour if there is a thread which is continually performing some computation, i.e., it is not blocked, but it is making any progress. Livelocks may be seen as a special case of non-progress behaviour.

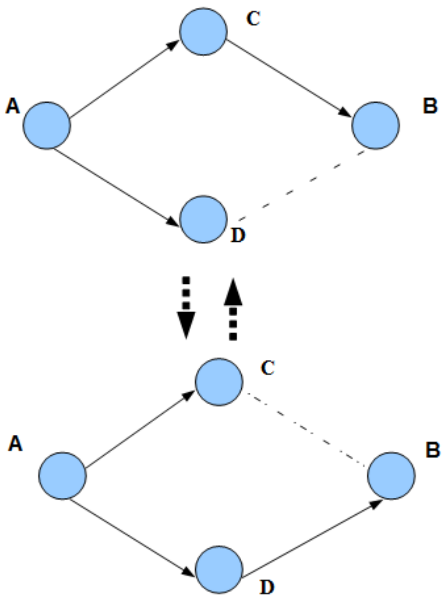


Figure 1.5: An example of livelock for message passing

An example of livelock for message passing is given in Figure 1.5. If a message is to be delivered from node A to node B, this message has the path ADB, which has a broken link between D and B. By the next chance of delivery, the message will be allocated the path ACB; meanwhile, in fact, the link between C and B has been disrupted. This condition may re-occur at the next chance of delivery. Therefore it seems there is a progress of allocating a path for the message to go through, although in fact the message cannot get through to its destination.

There are some significant works that have been published on this area and there are many approaches to describe and reason about dynamic networks. Works on Internet routing instability ultimately focuses on route oscillation and convergence problems, as well as on duplicate announcement. The two problems of message-passing in high-level models [34, 48, 11] and self-stabilising systems [10] in dynamic networks have been widely studied [35]. Numerous models and algorithms have been proposed but proofs of correctness (especially liveness) have tended to require the assumption that changes in networks eventually cease, i.e., they are no longer dynamic.

These works leave some questions unanswered. How can we **describe** the correctness criteria for re-routing algorithms in dynamic network systems? How does one **model and reason** about dynamic network systems in an appropriate formal and modelling framework, with an assumption that the graph is always connected, but as weakly as possible? The formal framework should be useful for obtaining the desired correctness criteria for dynamic network systems. How does one obtain **realistic** implementations, proof of correctness, and models of actual systems?

Here, we can establish the correctness of dynamic networks without the termination requirement. The correctness of dynamic networks is developed ensuring that: even when routing tables do not reflect the actual network connections, the routing information is correct sufficiently often; messages eventually get delivered; the network is sufficiently connected for sufficiently often; and there is no persistent livelock.

Furthermore, there are some ways of updating routing tables. Some mechanisms of routing-update algorithms are: broadcasting, using path to communicate, and using communication between processes. Some research into modelling dynamic networks based on particular networks such as mobile ad hoc networks or wireless networks, and maintains a focus on lower mechanisms such as broadcast or consensus. In this research, the model uses an abstraction which apart from the particular routing mechanism, and factorisation of analysis. It focuses on the message passing mechanism and looks for an abstract approach (ultimately correctness criteria for routing tables, networks and messages).

Implementation of dynamic networks in a program can be in a skeleton form with usable and unusable links, or based on the appearance and disappearance of nodes and links. This research uses a skeleton form to focus on understanding the behaviour of dynamic network models.

1.2 Summary of achievements

Some challenges in this dynamic networks research are: message-passing mechanisms in unreliable networks; and computations attempting to reach a consensus or compute global properties. A systematic approach to describing and reasoning about dynamic network systems can answer these challenges. That approach involves developing a formal framework for dynamic network systems and reasoning about concurrent systems by modelling dynamic networks using multiple processes.

The research hypothesis is as follows: we can reason about correctness of dynamic networks systematically based upon the techniques application for reasoning about concurrent systems. This research uses techniques from the proof properties of concurrent systems for the analysis of message-passing in dynamic networks. It also uses modal logic and fairness, and using a "demonic" process to model network changes. This process disrupts the connection between two nodes over time.

A new approach is introduced to proof techniques for dynamic networks; this uses ideas from concurrent systems [49] to analyse message-passing. Linear Temporal Logic is used and concepts of fairness are formulated which capture network properties. To express dynamic networks as concurrent systems [42], consider the dynamic changes as the result of a "demonic" process which runs concurrently with routing updates and message-passing.

By the correctness of dynamic networks, means under certain conditions, all messages will eventually be delivered. By formulating networks as concurrent systems, we can establish correctness for networks that never cease to change. By modelling at this level of abstraction, we are able to prove the properties of networks independently of the mechanisms in actual networks and therefore provide "a factorisation" of proofs of correctness for actual dynamic networks. The two abstract models are implemented as concurrent systems and then adopted the runtime verification system RULER [5], to analyse execution traces and test whether model instances satisfy the modal correctness for message delivery.

The main contributions of this research are:

- *abstraction of a dynamic network model using concurrent systems approach*;
- *concurrency proof techniques for dynamic networks*, including *factorisation of proof*; and
- *runtime verification on implementation of dynamic network models*.

Following sections discuss these achievements in modelling (abstraction hierarchy of models), reasoning and devising a correct algorithm.

1.2.1 Abstraction hierarchy of models

This research into dynamic networks attempts to move forward by combining two long-established research domains: concurrent systems and dynamic networks. So far, the other approaches that are used in dynamic network researches require termination to obtain liveness as one of the correctness properties. In a real world, dynamic networks do not terminate regarding the change. By considering dynamic networks as concurrent systems, it is a novelty to use the standard technique, which *requires no termination*.

Here, description of dynamic network model behaviour uses several logics. Description of predicates on actions are in first order logic, and actions between two states are presented in Hoare Triple logic. Discrete-time Linear Temporal Logic (LTL) [15] is used to describe the properties of the execution traces of these multi-process systems. Semantic in propositional logic derives properties of states.

We begin by developing an abstraction of a dynamic network model separately from mechanism and routing algorithms. Through such a separation, the level of this model is between the general abstraction level of the dynamic network model and the dynamic network model with particular mechanisms (such as broadcast, point-to-point, and consensus). So this dynamic network model is at a high level of abstraction, but reasoning can still be done even though without low-level involvement. Furthermore, this abstraction can then be extended to include lower level abstraction.

The hierarchical model approach in modelling a dynamic network exploits a separation concern. As an initial point, there are three general properties of dynamic networks being formulated: graph properties, message properties, and routing properties. From this level of abstraction, they can be extended further to create a more specific model. For instance, the relation definition between graph properties and message properties in a dynamic network two-process model will slightly differ from the three-process model, which also involves routing properties. Some of the key properties which enable us to reason about network correctness are expressed as *fairness* constraints [36] in concurrent process models. Fairness enables us to express, such as, the relative frequency of network change to message motion and of routing table updates to network change. Further discussion about this separation concern can be found in Section 4.2.

The dynamic network is modelled as a concurrent system. A concurrent system is a collection of asynchronous processes which possibly runs on different physical processors in a distributed environment [36]. This approach introduces two models. Firstly, *Model A* is a two-process model with instantaneous updates in which the routing tables are always correct. The two processes are a "demonic" *Disrupter* (which disrupts the connectivity of dynamic networks) and an *Organizer* (which attempts to deliver messages). *Model B* introduces a more realistic routing table update, adding a third process called an *Updater*. Here, the routing tables may not be correct at any time but routing is still possible. The *Disrupter* process can disrupt the edge connection (so it can switch between "on" and "off"). The *Updater* runs concurrently, re-calculating the routing update information to obtain available paths. If there is an available path, the *Organizer* process which runs concurrently can send a message to the next node along the path. Chapter 4 discusses more explanation of the models.

1.2.2 Concurrency proof techniques for dynamic networks

After we define the dynamic network model formally, the next stage involves developing proofs, proof structure and approaches to verification. Proving the correctness of message-passing in the dynamic network model involves three steps. Here, the correctness means that, under certain conditions, all messages will eventually be delivered.

Firstly, description of dynamic network systems behaviour *uses linear temporal logic*. Decomposition of the behaviour yields a connectivity requirement and fairness conditions. These are graph, message and routing properties, as well as no persistent livelock definition in temporal logic.

Secondly, we use *concurrency proof methods* to reason about dynamic network systems. The *colouring according to status* method is inspired by Dijkstra [12] and Gries [21]. The correctness definition of dynamic networks, in terms of network condition, is as follows: "if a network is sufficiently well-behaved, all the messages eventually reach their destination".

The third step is *Factorisation of proof.* The dynamic network model is built using the hierarchical approach, so it has levels of correctness according to this type of abstraction. *Factorisation of proof* means that the proof of correctness of an actual network (e.g. the internet) is split into proofs at various levels, which deal with separation concerns. Therefore, a proof of an actual network will require us to prove whether a temporal logic description is satisfied by an actual condition of the network. For example, we will be required to prove that the temporal logic description of connectivity is satisfied by an actual connectivity condition as met in network algorithms. The same technique is applied similarly for the other temporal logic conditions and levels of models.

Approaches to verification are carried out by the factorisation of analysis, such as: trace-oriented analysis and process-oriented analysis. Trace-oriented analysis shows changes of states over time, while process-oriented analysis shows changes of states over time by some actions which belong to a process. The relation between these two traces shows the connection between the temporal logic behaviour of the model and the Hoare logic of actions of each process.

Example: Figure 1.6 shows that the trace indicates the changing state of a message. Initially the colour is green; it becomes red and could become green again until moving to black. Figure 1.7 shows the changes of state due to an action belong to a process. Action GR(m) of the Organizer process cause the changing state of a message, from green to red. The message colour changes from green to black due to action GB(m). Another example is that action CTD(e1) cause the change of status of a link going down. This action belongs to the Disrupter process.

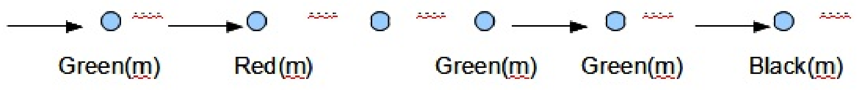


Figure 1.6: An example of trace properties

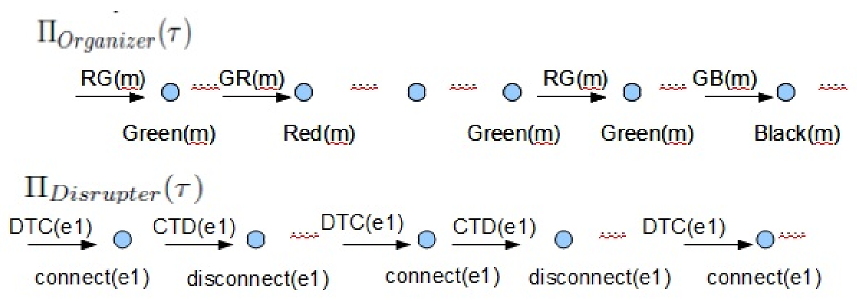


Figure 1.7: An example of process-oriented trace

1.2.3 Runtime verification of the implementation of dynamic network models

We could prove the implementation manually or we could use a verification technique such as model checking. Here a new approach is introduced based on runtime verification (RV) [39]. Whether or not a system satisfies the properties required for message delivery depends on inter-process interaction and the parameters involved in this. Runtime verification is particularly suitable here as it is the relationship of these parameters to the execution traces that determines the correctness of the dynamically allocated inter-process interaction.

The two abstract models, *Model A* and *Model B* (mentioned in Section 1.2.1), have been implemented as multi-threaded programs so that they can act as simulated concurrent systems and then adapted the runtime verification systems RULER [5]. These allow us to analyse execution traces to test whether model instances satisfy the modal correctness for message delivery. RULER is a rule-based runtime verification with dynamic rules, as opposed to automata-based systems. This is an experimental use of RV on concurrent models.

This runtime verification is doing two things: 1) validating models, which means satisfying the assumptions; and 2) providing an experimental platform for proof (using the particular assumption, to explore whether it satisfies). Notice that the validation of the dynamic network model implementation uses RULER, which is *an experimental system*. This work represents an additional example of RULER, particularly for a different system. It is not only about defining a RULER specification, but also about to deal with the concurrent systems and fairness conditions.

Using runtime monitoring on a concurrent system means that we should aware of any issues in concurrent systems as well. Furthermore, dealing with the fairness means dealing with the infinite traces. Some trace properties required are properties properly of infinite traces. On the contrary, runtime monitoring deals with finite traces. This contradiction brings many possibilities and interesting experiments. We use runtime verification to examine finite traces and relate this to the overall network behaviour.

1.3 Dissertation structure

Following this introductory chapter, Chapter 2 describes foundation of this research, such as logics, research domain for dynamic networks, and tools for runtime verification. The Hoare logic is used to describe the actions of each process. Properties of traces are presented in temporal logic. This chapter also presents a brief survey of various existing dynamic network models, together with discussion of their abstraction and proof. To complete the understanding of this research, there is a brief explanation of the frame problem, as well as a short description of RULER as the experimental verification tool.

Chapter 3 presents the research methodology, which uses preliminary studies of "on-the-fly" sorting. The material in this chapter is intended to establish the nature of change in dynamic networks starting with a simple example. It also provides an understanding of how a system can be considered as a concurrent system. "On-the-fly" sorting is an example of an attempt to sort *n* elements of array in ascending order while the value of items keeps changing over time. This sorting system involves two processes: a "bad" process (called the *feeder*, which keeps changing the value of the items) and a "good" one (called the *sorter*). Furthermore, this chapter describes how to reason about such systems through an example of proof (safety and liveness properties).

Chapter 4 is one of the core chapters of this dissertation. This chapter presents how dynamic networks are explored and how the model is developed. It discusses why and how to model a dynamic network as concurrent systems using a hierarchical approach. The description starts by exploring various possible models of dynamic networks as concurrent systems, followed by a formal dynamic network model, together with the general properties of the model. Afterwards, the dynamic network is modelled as a two concurrent process, the *disrupter* and the *organizer*; and then one other process is added in a three concurrent process model. These abstractions are followed by an explanation of trace analysis as the result of processes execution.

As a continuation from the previous chapter, Chapter 5 provides proof techniques for the dynamic network model and discussion about the frame problem in the following proof. The logic proofs are Temporal Logic proofs (trace property-oriented).

After Chapter 4 defines the dynamic network model, then Chapter 6 presents its implementation in multi-threaded programs. This simulates dynamic network models as a bridge from abstraction into automated verification. There is also discussion about colouring decisions.

The previous chapters present the abstraction of a dynamic network model, the proofs, and how it is implemented in a multi-threaded program. Chapter 7 focuses on verifying the correctness of dynamic network model through utilises RULER as the runtime verification tool in that implementation. Some results of experiments are presented in this chapter, including: sparse network vs. dense network, percentage of messages reaching their destination, and proof of the properties of the dynamic network model.

The final chapter, Chapter 8 gives an overview of what has been done in this dynamic network research. Moreover, it also sets out the strength of this work and how it can be extended or explored in future studies.

Chapter 2

Background

This chapter focuses on some foundational idea referred to this research. There are discussions of several logics related to this dynamic network research. We can use these logics to describe and reason about the dynamic network model. Communication mechanisms and routing in dynamic networks are presented after a refresh material of issues and techniques in concurrent systems. Subsequently, there is an overview provided the formal analysis of network systems so far and a brief survey of existing dynamic network models. One of the interesting aspects of those dynamic network models is about abstraction and proof. Some other materials are presented, such as: constant domain model, the frame problem, and runtime verification; followed by a summary.

2.1 Linear Temporal logic (LTL)

Logic is needed to express and verify properties of dynamic system such as safety and liveness (see Section 2.3 and Section 2.6.3 for further explanation about these two properties). Modal logic [15] is developed to study different modes of truth, where an assertion may be true depending on the given world. Temporal logic is a modal logic where truth values of assertions vary with time. This logic is an extension of classical logic, specifically adding operators relating to time [20], with some of the formal foundations provided by modal logic [15].

The modality operators are:

- sometimes φ is true if φ holds at some future moment
- always φ is true if φ holds at all future moments

Temporal logic is often used to specify and verify reactive systems, i.e. systems which continuously interact with the environment.

Temporal logic is the logic of time. There are some different ways of modelling time, such as: linear time vs. branching time; discrete vs. continuous time; points vs. intervals; past vs. future; and finite vs. infinite into the future. This work uses LTL, which reveal about computation as sequences of states.

Here, it is considered discrete linear time. A linear time structure is a three tuple $M = (S, x, L)$, depending on a set AP of atomic propositions P, Q, P', Q',..., of:

- a set S of states
- a timeline $x : \mathcal{N} \rightarrow S$
- a labeling $L : S \rightarrow \gamma\,(AP)$ of states

A timeline x is denoted as s_0s_1.... Let $x = s_0s_1s_2$...., then $x(j)$ for s_j, and x^j for s_js_{j+1}... Some LTL operators being used are as follows:

- $\Box\varphi$ means φ will always hold in every state
- $\Diamond\varphi$ means φ will eventually hold in some state
- $\varphi\ \mathcal{U}\ \psi$ means ψ will eventually hold, and until that point φ will hold
- $\varphi\ \mathcal{W}\ \psi$ means φ will hold until ψ holds

Propositional LTL is the least set of formulae generated by the following conditions:

1. each atomic proposition P is a formula
2. if φ is an LTL formulae, so is $\neg\varphi$
3. if φ and ψ are formulae then $\varphi \wedge \psi$ are formulae
4. if φ and ψ are formulae then $\varphi\ \mathcal{U}\ \psi$ and $\bigcirc\varphi$ are formulae

Other temporal modalities are defined as abbreviations: $\Diamond\varphi \equiv$ true $\mathcal{U}\ \varphi$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$. Also, the infinitary modalities: $\Diamond_\infty\varphi \equiv \Box\Diamond\varphi$ and $\Box_\infty\varphi \equiv \Diamond\Box\varphi$.

Propositional LTL semantics are defined with respect to a time structure and a time line. $M = (S, x, L)$. $M, x \models \varphi$ means that "in structure M and time line x formula φ is true".

- $x \models P$ iff $P \in L(x(0))$, for $P \in AP$

- $x \models \varphi \wedge \psi$ iff $x \models \varphi$ and $x \models \psi$
- $x \models \neg\varphi$ if not $x \models \varphi$
- $x \models (\varphi\ \mathcal{U}\ \psi)$, iff $\exists_j x^j \models \psi$ and $\forall_{k<j} x^k \models \varphi$
- $x \models \bigcirc\ \varphi$ iff $x^1 \models \varphi$

The validity of a temporal logic expression on a trace is as follows.

$$\begin{gathered}
\tau, i \models \Box\varphi \text{ iff } \forall j > i, \tau_j > i \models \varphi \\
\tau, i \models \Diamond\varphi \text{ iff } \exists j > i, \tau_j > i \models \varphi \\
\tau, i \models \varphi\ \mathcal{U}\ \psi \text{ iff } \exists j > i, \tau_j > i \models \psi \ \textit{and}\ \forall i, 0 < i < j, \tau_i < j \models \varphi \\
\tau, i \models \varphi\ \mathcal{W}\ \psi \text{ iff } \forall i, 0 < i \leq j, \tau_i \leq j \models \varphi \ \textit{and} \\
\exists j > i, \tau_j > i \models \psi \ \textit{or}\ \forall j > i, \tau_j > i \models \varphi
\end{gathered}$$

A propositional LTL formula φ is **satisfiable** if *there exists* a linear time structure M = (S, x, L) such that M, $x \models \varphi$. A propositional LTL formula φ is **valid**, notation $\models \varphi$, if *for all* linear time structures M = (S, x, L) then M, $x \models \varphi$.

2.2 Hoare logic

Hoare logic is the first logical formalism defined for reasoning about programs in the form of a composition [30]. This logic considers a program as an implementation of some functionality, which expressed formally in terms of input and output. Using Hoare logic, a program is decomposed into smaller programs; in consequence, so this logic has the program explicit in the language.

There are two types of correctness of programs. A program is called **totally correct** if and only if it is guaranteed to terminate. The other type of correctness does not mainly focus on the termination of the program. A **partially correct** program means that when it terminates then the output specification is guaranteed.

The syntax of Hoare logic (called Hoare triples) is as follows:

$$\{\varphi\}P\{\psi\}$$

where P is a program built from a set of actions, and φ and ψ are formulas. φ (called *pre-condition*) specifies what input to the program P should be, and ψ is called the *post-condition*. This triple is read as " if φ is true before execution of P, then ψ is true after execution of P". Hoare triples will be used to describe the properties of actions.

$$S_1 \xrightarrow{a} S_2 \models \{\varphi_1\}\alpha\{\varphi_2\}$$
$$\text{iff } a = \alpha \; \textit{and} \text{ if } S_1 \models \varphi_1 \text{ then } S_2 \models \varphi_2$$

S_1 and S_2 are state 1 and state 2, where a is an action which changes the state of the system from S_1 into S_2. This triple is read as " if φ_1 at state 1 is true before execution of α of action a, then φ_2 at state 2 is true after execution of action α".

2.3 Concurrent systems: issues and techniques

This research considers dynamic networks as concurrent systems. There are some issues in concurrent systems that always to be carefully considered. The issues are deadlock, safety, liveness, livelock, fairness, and interference.

Deadlock occurs when the system is prevented from taking any action (no transitions are possible since all enabled conditions are false). This means no further progress. There are four necessary and sufficient conditions for deadlock [42]. Firstly, serially reusable resources: the processes involve shared resources, which used under mutual exclusion. Secondly, incremental acquisition: processes held on to the resources already allocated to them while waiting to acquire additional resources. Thirdly, no pre-emption: once acquired by a process, resources cannot be forcibly withdrawn, but are only released voluntarily. And fourth, a wait-for cycle: a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

There are properties that should true for every possible execution. A *safety* property asserts that nothing bad happens. In terms of the model, this means that there is no reachable error state. Safety property is specified by stating directly what is required. A *liveness* property asserts that something good eventually happens. In terms of the model, liveness correspond to progress. A progress property asserts that it is always the case that an action is eventually executed. This property requires a fair choice, in which if a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often.

Another issue, called "livelock", can still occur (see Section 1.1 for the definition). Livelock happens when system component is prevented from taking any action or a particular action. Livelock can have a connotation to *starvation* (systems with a non-zero service cost and unbounded input rate may experience starvation), *infinite execution* (the individual processes of an application may run successfully, but the application as a whole may be stuck in a loop), or *breach of safety properties* [28]. There is a standstill detection through generating the state predicates and temporal rules, called *Liveness signature* (i.e. a partial specification of application states that are significant in determining whether a program is making forward progress, and the valid transitions between those states), then recovery and repair. There is one definition of livelock and no progress behaviour [32] involve common concurrency errors. This is divided into: *various errors in safety*, i.e., errors that cause something bad to happen (data races, atomicity violation, order violation, deadlocks, and missed signals), and *various errors in liveness*, i.e., errors that prevent something good from happening, as well as errors mixing liveness and safety (starvation, livelocks, non-progress behaviour, and blocked threads).

Fairness is the general term for fair choice in concurrency and the hardest one to formulate. There is *weak fairness* and *strong fairness*. Weak fairness means that any transition that is continuously enabled eventually happens. Strong fairness means that any transition that is enabled infinitely often will eventually occur (see also Section 2.6.2).

Consider the fairness notion which describes conditions for every fairness F_i on valid traces. System components are defined to determine the granularity level of fairness as a *concurrent process*. There is a definition of the condition under which the component will proceed ("sufficiently often") to determine the strength of fairness. Whether a system component is possible depends on the type of system components. The concurrent process becomes possible if some action of the process is enabled. For *concurrency fairness*, fairness is defined as a property that states that no concurrent process should be delayed indefinitely. This "enabled" is defined in terms of traces and automata.

In concurrent systems, the execution of the instruction, which from a set of threads, could be interleaved arbitrarily. The result of concurrent process can be incorrect due to this interleaving, particularly incorrect state update of the shared variable/object. *Interference* [42] is a destructive update, caused by the arbitrary interleaving of actions from concurrent process. A common solution for this problem is using mutually exclusive access to a shared object. In Java programming, we can use keyword "synchronize" to make a method mutually exclusive.

Reasoning about concurrent systems requires properties and reasoning techniques which different from sequential programs. Owicki and Gries [49] extend Hoare logic to be able to prove the concurrent program. Gries [21] proves safety properties using proof of interference freedom. Dijkstra [12] presents on-the-fly garbage collection (using a mutator and a collector) and consensus algorithms. Gries [21] and Dijkstra [12] utilise colouring notification according to status as a means of reasoning about concurrent systems. Owicki and Lamport [50] propose formal proof of liveness, combining of Lamport's proof lattices with Pnueli's temporal logic. Manna and Pnueli [43] prove eventuality properties using the approach called "proof by eventuality chains". Hoare et al. [31] provide process algebra as the study of the behaviour of parallel or distributed systems by algebraic means and reason about such systems using equational reasoning. Ramamoorthy and Ho [52] use an extended timed-Petri net to model the synchronization involved in real-time asynchronous concurrent systems.

2.4 Communication mechanisms and routing

Before discussing the formal analysis of network systems, there is a question about what type of relation between message passing and routing is involved in dynamic networks? Here is how the two of them interrelated.

Message passing is a mechanism for sending and receiving messages between processes, usually within a network. The type of communication for this mechanism can be synchronous or asynchronous. In synchronous communication, it uses a synchronisation point and requires no buffering. By contrast, asynchronous communication requires buffering; the sender and receiver overlap the computation, and this may be blocked, resulting in deadlock and therefore unreliable. Different mechanisms for a message being passed are point-to-point, broadcast, and consensus. Some concerns about message passing are the reliability of message transfer and whether the message delivered in the right order (if message is divided into some data packet).

Dynamic routing has been implemented in various network technologies and used in many kinds of routing protocols. For example, routing in ad hoc networks is divided into *proactive* and *reactive* routing algorithms. A proactive routing algorithm (called "table driven routing algorithm") has a full network view but is not feasible for large networks. The topology update in this type of algorithm will broadcast immediately. The routing protocol being used by this type of routing algorithm is DSDV (Destination Sequenced Distance Vector routing), which guarantees no loops in the routing table. The routing table consists of a minimum number of hops, the next hop, and the sequence number. The routing update may be a full route or an incremental update; the content of the routing information includes the address of the destination, the number of necessary hop, and a sequence number.

However, a reactive or on-demand routing algorithm does not send topology updates. It floods a route request from one node to other nodes within the network. The sender node will receive a response from the destination or an intermediate node. The routing protocol being used by this type of algorithm is AODV (Ad hoc On-demand Distance Vector routing). This protocol uses a route request (RREQ), which consists of source address, destination address, and sequence number; and a route reply (RREP), which created by an intermediate node or the destination. Instead of using a routing table, the algorithm uses a routing cache, which reduces the packet overhead. The movement of route request is downstream, then the RREP is sent an infinite next hop metric upstream.

Routing in dynamic network model leads to actual path vs. potential path, the propagating routing information, and the reliability of network and information. Ramanathan et al. [51] show an example of the actual path vs. potential path in their challenged networks. A network may never be connected and never has an end-to-end path, but there is a temporally ordered link activation sequence that forms a path over time for every pair of nodes.

Propagating routing information within dynamic networks is distinctive due to its dynamic changes. For the wireless ad hoc networks which consist of some subset of the network, it is suitable to use dominating-set-based routing. Wu and Li [64] propose a simple distributed algorithm that can quickly determine a connected dominating set. Two sets of the network are connected through a gateway node. With this algorithm, only individual mobile hosts update their gateway status. Moreover, the entire network recalculate gateway or non-gateway status. The challenge is when and how each node should update or recalculate gateway information.

Baldoni et al. [4] provide an example of network and information reliability. They use a one-time query problem as a benchmark. Their model uses failure detection to delete faulty processes from the neighbourhood set in the case of the worst scenario (in which the intermediate link crash during query interval). Then we can assume that the values to include in the query computation are: at least coming from nodes that belong to the graph G (when the query starts); and remain connected to the querying process through a subgraph of G (during the whole query interval).

In summary, the relation between message passing and routing in dynamic networks should consider what type of interface between them. This interface can be represented in terms of logical properties to describe the actual and potential paths. Propagating routing information should consider whether there is a process communication or the use of shared memory concurrency. That should give a reliable network and information for all nodes within the dynamic networks.

2.5 Formal analysis of network systems

We now consider approaches to the formal analysis of network behaviour. First we need to explore how far the works on formal analysis of networks. The formal analysis of network systems starts from problems in inter-domain routing. There are the route oscillation and the convergence problem [6, 61, 22, 23, 24, 25, 55], link failures [38, 66], and duplicate announcements [14]. Furthermore, Labovitz et al. [37] shows that the instability which reflects real topological changes can lead to: increased packet loss; delay in network convergence, and more memory or CPU overhead on routers.

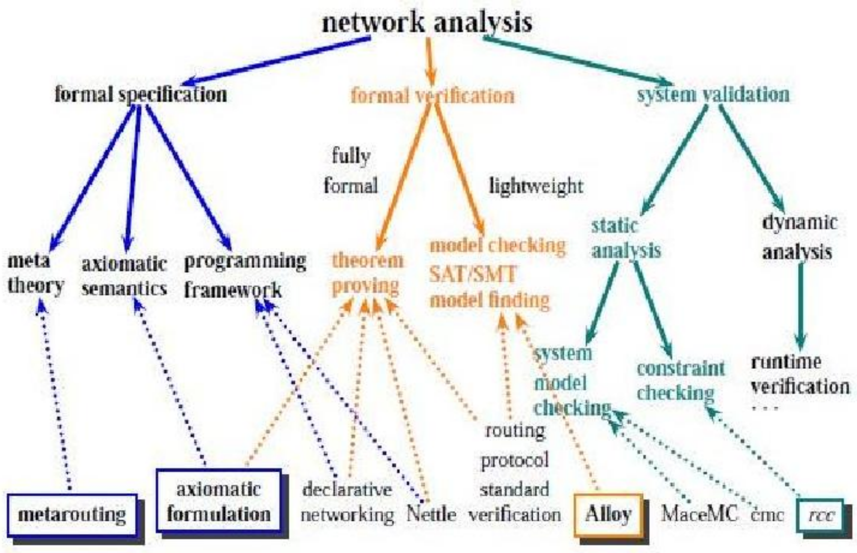


Figure 2.1: Taxonomy of formal network analysis (source: Wang [63])

Wang [63] classified internet challenges into routing, forwarding, and addressing. Figure 2.1 shows how these challenges are considered in the formal network analysis, such as formal specification, formal verification and system validation. Some researches in routing are: *metarouting* [26] (which is classified in the formal specification) and *rcc* [17] (which is an example of system validation). *Metarouting* is an algebraic meta-model for routing policy, which uses a routing algebra [57] with the added property of a convergence guarantee. Based on an assumption of the correctness of the BGP (Border Gateway Protocol) mechanisms, metarouting focuses on routing policy. Meanwhile, *rcc* is a router configuration checker for real BGP systems which identifies two types of high-level correctness properties: path visibility and route validity [18]. Feamster et al. [19] use a routing logic.

Figure 2.1 also shows that there is still not much work done in system validation using runtime verification. This field will be one of interest in this work of dynamic networks. The dynamic networks as concurrent systems will fill the place with the usage of runtime verification RULER for network analysis, particularly dynamic network behaviour.

In terms of forwarding challenge, one of the research is *axiomatic formulation* [33], with the assumption that all transmissions are error-free, infinite table and unaware to time. The formulation uses abstract switching element as an object that participates in network communication and relays messages. Otherwise, *alloy* [65] focuses on addressing. This is a lightweight integrated tool for object-oriented style formal specification and automatic analysis. An abstract model for network notions, such as connection and inter-operation, are constructed in Alloy specification logic.

There is some related research such as a formal framework of network systems using the separation concern of functionality [29], routing logic [57], the correctness properties of internet routing [18] and proof of inter-domain policy [62]. There are also some different formal approaches which are beyond the scope of this thesis, such as: multi-agent systems, the evolutionary framework of network services using genetic algorithm [47], and game theory [16].

2.6 A brief survey of dynamic network models

After the discussion about formal analysis of network systems, the following section presents related work in dynamic networks research. Dynamic networks in literature have different names with different intentions. The names are “challenged-networks”, “unreliable network”, “time-varying graph”, and “delay tolerant networks”. Although the names and purpose sometimes differ, they are based on the frequent changes that can occur within networks.

There are various reasons why networks can change over time, such as the reliability of links connectivity, and the creation of new nodes. The objective of this section is to obtain an understanding of the significant properties within the models, from the point of view of building a formal framework for dynamic networks. This survey aims at setting out a comprehensive overview of the significant behaviours of existing dynamic network models, and to understand the work in this area so far.

Nine papers are explored as representatives of this dynamic network models area; they are chosen based on the citation index as well as on other considerations. Those considerations include papers that are more recent, foundational papers, related papers, well-known journal papers, and an overview of a recent survey. The comparative analysis considers system structures changing over time, as well as the attributes of computation. This survey presents a comprehensive modelling dimension, regarding the network infrastructure, the dynamic changes of nodes and links, as well as the model attributes.

Another survey of dynamic network models includes work from Casteigts et al. [9], which classifies the dynamic network models based on the time-varying graph concept. This concept focuses on system structure, which varies across time. The temporal properties include the underlying graph, point of views, the temporal path, and reachability time. The classification ranges from a class with general assumptions to assumptions that are more specific. Meanwhile, Kuhn et al. [34] present discussion of works on dynamic network models in terms of topology change assumptions, protocols, interval connectivity, and the computational model.

A description of the type of model shown in a dynamic network model is given in Section 2.6.1. To provide the parameters for comparing the existing models, there is a list of the modelling dimensions of dynamic network models in Section 2.6.2, and details of the comparison in Section 2.6.3. The comparative analysis results are used to determine the important elements in a general formal framework of dynamic networks.

2.6.1 Dynamic network models

This section discusses, to some extent, a model that existing dynamic network models try to describe. The dynamic network models being explored focus on the following characteristics. Firstly, *dynamic features*, such as "what set of assumptions" the models have. A dynamic network model can have assumptions about the underlying graph and its sub-graph over time, the rate of change, as well as requirements and conditions for the computational algorithms. These assumptions can differ from one model to another on how restricted they are or not.

Another dynamic feature is “what has been changed” in the network systems over time. Nodes, edges, edge cost, or topology can change over time. This dynamic aspect may have different combinations, such as fixed-nodes with fixed-edges, fixed-nodes with varied-edges, varied-nodes with varied-edges, and increment or decrement of link cost; and node position may change as well.

Another consideration is about the type of impacts on the models with respect to the changes. Suppose there is a message sent from a sender node to the receiver node within the network. If a change occurs on a node or an edge at a point of time then how does the node in a dynamic network know the size of the actual network to deliver the message? Furthermore, if an intermediate node between sender and receiver disappears or the edge down, then how does a dynamic network model deal with this situation? It is also interesting to find the stable property that models use, such as: networks eventually stabilising vs. do not eventually stop changing. Some models have an assumption that a network eventually stabilises to simplify the problem. Other models may assume that the network continually changes, which is more realistic but difficult one as well; we have to maintain all the state changes over time.

Secondly, *feasible computational features*. Each node stores knowledge about the network, such as the diameter of the network and the neighbourhood nodes, to enable it to do the computations. Dynamic features limit this knowledge and affect the constraints of algorithms, such as in the flooding algorithm or routing algorithm.

The third is about *the complexity*, regarding how long the information spreads within the network with respect to time and space constraints. Finally, it is about *model applicability*. The existing dynamic network models describe various network technologies. Some of them describe a particular network technology; with some other it is claimed that they describe any network technology, ranging from traditional communication networks to mobile ad hoc networks.

2.6.2 Dimensions of modelling

Section 2.6.1 discussed the characteristics of dynamic network modelling. This section lists dimensions of dynamic network models as the model attributes.

1. **Time-related issues**, such as: static vs. dynamic, the rate of change, the speed of information spread, and discrete-time vs. continuous-time. In *static vs. dynamic*, a static network model is considered as a model with the slow changing of nodes

or links. The availability of nodes and links is nearly the same over a period of time. If the rate of change is denoted by r, then this model has r equal to zero. Meanwhile, a dynamic network model is considered as a model with the fast changing of nodes, or links, over time. The *rate of change* means the change can occur at a point of time or even infinitely often. *The speed of information spread* means that the model should consider how long the information reaches all nodes within the dynamic networks. *Discrete-time vs. continuous-time*: models that use discrete-time will describe the time in a difference equation, while models that use continuous-time will describe the time in a linear differential equation.

2. **Stochastic vs. non stochastic**. In a stochastic model, the connection or the existence of an edge in a random graph G is estimated using the probability number p.

3. **Fairness constraints**. Fairness is defined intuitively in [36] as a property of computations in which no component of the system that becomes possible sufficiently often should be delayed indefinitely. According to granularity level, fairness consists of process fairness, event fairness, transition fairness, and state fairness. Based on the strength of these, fairness is classified into *unconditional*, *justice* (weak fairness), and *fairness* (strong fairness). In *unconditional*, every process proceeds infinitely often, while in *justice*, the process may become disabled; if process is enabled continuously from some point onwards then it eventually proceeds. In *fairness* (strong fairness), if a process is enabled infinitely often then it proceeds infinitely often. Therefore, fairness is a collection of mostly independent properties depend on granularity level and the strength required. There are some notions of fairness concerned with dependent properties such as *equifairness* and *probabilistic fairness*. Equifairness is defined in [36] as follows: if a group of guards is enabled infinitely often, then there exist infinitely many time instants where all members of the group have been chosen the same number of times. Probabilistic fairness is applied to a computational model based on a state transition system with probabilities.

4. **Algorithmic constraints and behaviour**. The dynamic network models concern the type of computation that can run within the model. The algorithmic constraints represent this concern through the limit on message size, upper bound and lower bound, and the type of algorithms, such as flooding and routing. These attributes (including the finite or infinite behaviours, consensus problems, and

logical descriptions of the model) drive the type of correctness and the termination of the algorithms.

5. **Inter process communication mechanisms**. Examples of these include: *shared memory* and *message passing*. The *shared memory* mechanism uses a shared variable for communicating between processes. In case of the *message passing* mechanism, there is an exchange of message between processes through a send and receive action. This type of mechanism can be an exchanging message from one process to another, as in *point-to-point*, or from one process to the rest of the processes within the network systems, as in *broadcast*.

6. **Message passing networks vs. consensus networks**. A message passing networks is concerned with finding an optimal routing and to act as a non-terminating network. Meanwhile, a consensus network is concerned with using a constant to reach decisions within the network and to act as a terminating network.

7. **Timing model**. Examples of these include: *synchronous*, *asynchronous*, and *partially synchronous*. The synchronous model describes the execution of processes simultaneously, while the execution of process in an asynchronous model runs arbitrarily. Meanwhile, a partially synchronous model (a timing model which lies between a synchronous and asynchronous model), runs with the assumption that processors have some knowledge of time, such as message delivery times, and of approximate real time.

8. **Knowledge nodes of network topology**. This relates to local vs. global. Most dynamic networks rely on local knowledge about the network systems (neighbourhoods). The wired networks have more information about the network topology (as the global knowledge for a node).

2.6.3 Abstraction and proof algorithm correctness for dynamic networks

This section discusses the abstraction and proof of the existing dynamic network models to understand the related works of dynamic network research. Firstly, the purpose of each model is described, then Table 2.3 and Table 2.4 present the models relative to the dimensions of modelling as mentioned in Section 2.6.2. To clarify the models, the categorisation of the models is grouped by changes in the system structure over time and the attributes of computation. This review provides an understanding of how to model dynamic networks to date.

Model	**Type of changes**
Zhang	Fixed-Node Fixed-Edge
O'Dell	Fixed-Node Varied-Edge
Clementi	Fixed-Node Varied-Edge
Rupert	Fixed-Node Varied-Edge
Kuhn	Fixed-Node Varied-Edge
McElliece	Varied-Node Varied-Edge
Ramanathan	Varied-Node Varied-Edge
Baldoni	Varied-Node Varied-Edge
Grindrod	Fixed-Node Varied-Edge

Table 2.1: Type of changes in dynamic network models

Table 2.1 shows the existing dynamic network models with respect to the dynamical aspects as follows: *fixed-node fixed-edge*, which means that the set of nodes and edges are the same over time; *fixed-node and varied-edge*, which means that the set of nodes remains the same but a member of the set of edges can change over time; *varied-node and varied-edge*, which means that both the set of nodes and the set of edges can change over time.

Model	Internet	Wireless Ad Hoc Network	MANET/DTN	Wireless Sensor Network
Zhang	▨			
O'Dell			▨	
Clementi				▨
Rupert			▨	
Kuhn		▨	▨	
McElliece			▨	
Ramanathan			▨	
Baldoni	▨	▨	▨	▨

Table 2.2: Model applicability

Table 2.1 shows that most of the existing dynamic network models take into account the fixed-node vary-edge changes. This selection of the dynamic aspects regarding the simplified model and computation. Furthermore, the applicability of each model is described in Table 2.2. A model with a particular technology focuses on model attributes more deeply than a model with broad applicability.

Some of the many models that have been proposed for describing dynamic networks are explained as follows. Zhang et al. [66] define a network model (here called the “Zhang model”) to establish the impacts of link failure location on routing dynamics at the inter-domain level on the Internet. They use two kinds of rate to differentiate between the network growth at the core (network of service providers) and at the border (networks which rely on the core networks for global reachability). The growth rate of the border node is exponential while the growth rate of core node is linear; based on the Internet growth pattern where the growth of border networks faster than the core networks. Their simulation model, which relies on probability to determine link failure, differs from the case of Labovitz et al. [37] who use data measurements that are difficult to interpret.

O’Dell and Wattenhofer [48] define a dynamic network model (here called the “O‘Dell model”) regarding network mobility and limitation knowledge of the network. They investigate to what extent the flooding and routing algorithm is possible with this model.

Clementi and Pasquale [11] describe a dynamic network model (here called the “Clementi model”) which focuses on time complexity in terms of the speed of information spread. Their model relies on the random Markovian process. Meanwhile, Ruppert and Aspnes [54] define a population protocol model (here called the “Ruppert model”) which can be used to model mobile ad hoc networks of tiny devices. Their model is a collection of programmed finite state machines and has pairwise interactions between them. The infinite execution of the protocol is an infinite sequence of configurations.

Kuhn et al. [34] define a dynamic graph model (here called the “Kuhn model”) for dynamic network using certain assumptions, such as: the network does not eventually stop changing; that there is fixed set of nodes that operate in synchronised rounds and that communicate by broadcast; that there is no restriction on the mobility of the nodes; and that there is no assumption that geographical information or neighbour discovery are available.

McEliece and Soedarmadji [58] define a dynamic network model (here called the “McEliece model”) inspired by the O’Dell model. They relax the assumption of the model in terms of the dynamical aspects (they assume varied-node and varied-edge) and algorithmic constraints (they remove the requirement that the network stays connected at all times).

Ramanathan et al. [51] define the formal model of three kinds of dynamic networks and their routing mechanism (here called the “Ramanathan model”) is based on connectivity as the main concern. They studied the solvability (in terms of packet

Model	**Time-related issues**	**Stochastic**	**Fairness constraint**	**Algorithm constraints and behaviour**
Zhang	static, probabilistic link failure, $\geq$ 3 hops, discrete-time	Yes	probabilistic fairness	slow convergence, finite
O'Dell	dynamic, evolve slower than the message transfer time, $< 2\mathrm{T}_n$, discrete-time	No	weak fairness	require correctness and termination under conditions of storage space and number of nodes, finite
Clementi	dynamic, probabilistic death rate and birth rate, flooding time e-quals the diameter of the graph, discrete-time	Yes	probabilistic fairness	flooding time with upper bound and lower bound, finite and infinite
Ruppert	dynamic, the interaction between two agents, total number of steps to convergence, discrete-time	No	strong fairness	correctness: all agents produce correct output from some point of times onward, finite state, infinite execution
Kuhn	dynamic, T-interval connectivity, $O(n^2)$ rounds, discrete-time	No	weak fairness	consensus of k-agreement problem, req. correctness and termination, infinite
McElliece	dynamic, finite amount of time, within a time bounded function (the maximum message transit time between adjacent nodes, the maximum number of nodes in the network), discrete-time	No	weak fairness	require correctness and termination (arrival of message m depends on how far the nodes are from origin, how rapidly edge E and node V change, and how fast m travels), finite
Ramanathan	dynamic; one or more times for connectivity, routability, trans-portability; at time T (when a path from source to destination becomes available); discrete-time	No	weak fairness	require solvability (packet can be delivered), infinite
Baldoni	dynamic, subgraph G remain connected during the whole que-ry interval, a function of bounded diameter of the graph, discrete-time	No	weak fairness	use one-time query problem, termination and validity properties, finite and infinite
Grindrod	dynamic, probabilistic range-dependent death and birth mechanism, message range/longest range edge successive time steps, discrete-time	Yes	probabilistic fairness	use the equilibrium property, finite

Table 2.3: The result of dynamic network models comparison

Model	IPC mechanisms	Message - passing networks vs. Consensus networks	Timing model	Knowledge
Zhang	broadcast and point-to-point	Message passing	synchr.	Local & Global
O'Dell	broadcast	Consensus	asynchr.	Local
Clementi	broadcast	Consensus	synchr.	Local
Ruppert	point-to-point (between two nodes)	Consensus	synchr.	Local (pairwise interaction)
Kuhn	broadcast	Consensus	synchr.	Local
McElliece	broadcast	Consensus	asynchr.	Local
Ramanathan	point-to-point	Message-passing	synchr.	Local
Baldoni	broadcast	Consensus	synchr./asynchr. with failure detection	Local (reliable)
Grindrod	broadcast	Consensus	synchr.	Local

Table 2.4: The result of dynamic network models comparison (cont.)

delivery) using this model and various classes of algorithms. Baldoni et al. [4] define a dynamic distributed system (here called the "Baldoni model") with emphasises two orthogonal dimensions: the number of process/entities/nodes and diameter of the graph (in terms of geography/neighbourhood). They use one-time query problem as a benchmark problem (a simple data aggregation problem). Their proof shows that it is impossible to solve the dynamic one-time query problem if number of process and diameter of the graph are infinite. Grindrod and Higham [27] describe an evolving network that deals with dynamic links. Their model (here called the "Grindrod model") uses range dependency to evolve and simulate propagation within an evolving graph.

Table 2.3 and Table 2.4 shows various models with respect to the model attributes. Based on the comparison using the modelling dimension at Section 2.6.2, the discussion of models is grouped by the system structure changing over time and the attributes of computation. The models related to varying network topology over time are classified by two subgroups: *non-stochastic dynamic network model* and *dynamic network model using stochastic analysis*. Firstly, the *non-stochastic dynamic network model*. The underlying graph and subgraph connectivity are the assumptions which dominate in the Kuhn model [34] and O'Dell model [48]. Both of the models assume that the underlying graph is always connected over time. In the Ramanathan model [51], the dynamic network models are defined into three kinds of graph connectivity: *eventually connected* (the network is connected at some particular times), *eventually route-able* (the network may never be connected but there is an available end-to-end path between every pair of nodes at some points in time), and *eventually transportable* (the network may never be connected and never have an end-to-end path, but there is a temporally ordered link activation sequence that forms a path for every pair of nodes).

Meanwhile, Baldoni et al. [4] model the dynamic sizes in terms of the number of entities and geography, relaxing the fully connected assumption into a more realistic one. Several models can be defined that differ in terms of the assumptions made about the number of processes that can concurrently be part of the system at any time. Each process has only a partial view of the system, called its neighbourhood.

Secondly, *dynamic network model using stochastic analysis*. The Zhang model [66] is viewed as a dynamic network with slow rate of change (static), compared with other models. Their model uses a probabilistic approach to simulate the growth of the network. Furthermore, a random graph can be used to represent the dynamic network. This type of graph can describe the probability of edge existence between nodes in a graph with the same node set. Some related work has used a random graph to model the dynamic network, where the existence of edges in a graph varies over time. Clementi et al. [11] have presented the dynamic networks using random graph and the availability of edge determined by probabilistic number. The model uses probabilistic number p to denote the *birth rate* of an edge and q to denote the *death rate* of an edge. If edge E_i is available at the time t, then at the time $t + 1$ the edge E_i has probability q of being unavailable. By contrast, if edge E_i is unavailable at the time t, then at the time $t + 1$ the edge E_i has probability p of being available. The Grindrod model [27] uses range dependency for birth rate and death rate, in which a transitive connectivity between nodes is determined by the short range or long range. This type of dynamic network models usually has probabilistic fairness.

Now, the models are grouped regarding their *attributes of computation*. Firstly, *synchronous dynamic network*. The Kuhn model [34] uses a stability property called *T-interval connectivity*, which assume that for every T consecutive round there exists a stable connected spanning subgraph. The model can be used to model various dynamic networks, ranging from mobile networks, static or dynamic wireless networks, to traditional communication networks. Secondly, *asynchronous dynamic network models with weak fairness constraints*. O'Dell et al. [48] assume that the change of a graph G is allowed after each message transmission. This mechanism is used to ensure that an already inserted node can always receive the message. The correctness of the routing algorithm in this model is based on the assumption that the destination exists and must be reached. Termination means that eventually no node will transmit any more messages. Their work is interested in how high mobility and algorithmic constraints affect *what can be computed* in an ad hoc network. The graph G is connected for all times t. Changes of the network can occur within time T for all nodes. T is maximum message transmission time. This model uses broadcast medium, negligible local processing time, and asynchronous message transmission (therefore lost messages are possible). Transmission at a node will reach some node after, at most, $2T$ time.

Thirdly, *dynamic network models focus on the type of computation*. Basic communication task, such as flooding and routing, are discussed in the Eliece model [58], the Kuhn model [34], the O'Dell model [48], and the Ramanathan model [51]. Their works give the algorithmic constraints for correctness, termination, or solvability. Fourth, *dynamic network models focus on speed of information spread*. Flooding time has been explored in the Clementi model [11], the Kuhn model [34], and the O'Dell model [48], using the upper bound and lower bound. Fifth, *dynamic network models focus on message passing networks vs. consensus networks*. The Ruppert model [54] is one of the models of consensus networks, for which all nodes as agents should produce the correct output. The Kuhn model [34] is another instance of consensus networks, in which the model uses consensus on the k-agreement problem. By contrast, the Ramanathan model [51] describes the message-passing networks, which are concerned with the routing mechanism to deliver a packet from sender to receiver.

From all of the existing dynamic network models, some significant features with respect to the research challenges in dynamic network model are: *propagating routing information, reliability (of information and network)*, and *actual path vs. potential path*. The dynamic network model defined by O'Dell et al. [48] captures the asynchronous dynamic network model which emphasises appropriately the flooding and routing algorithm. However, their model relies on a broadcast mechanism and has a restrict assumption of graph connectivity over time. The Kuhn model in [34] is appropriate for describing a synchronous dynamic network model, although it focuses on how to obtain the knowledge of network size using the counting algorithm and the consensus problem of k-token dissemination (eventually all nodes will propagate the k token).

The stochastic analysis model by Clementi et al. [11] represents the uncertainty in a dynamic network. Their model still focuses on time complexity rather than feasibility of computation, such as routing or message passing which will be the subject of their future work. The most appropriate model at some degree of routing and message passing is defined by Baldoni et al. [4] and Ramanathan et al. [51]. Baldoni model uses the failure detection to obtain a reliable neighbourhood and considers the possibility of the message being lost. The Ramanathan model does not account for the delay in packet delivery but considers the actual path vs. potential path and the point-to-point mechanism.

The Grindrod model [27] uses range dependency for birth rate and death rate, in which a transitive connectivity between nodes is determined by the short range or long range. This type of dynamic network models usually has probabilistic fairness. Probabilistic fairness applies to a computational model based on a state transition system with probabilities.

Chen and Welch [10] propose a *self-stabilizing mutual exclusion algorithm (ssME)* which can tolerate restricted topology changes after it has converged, using tokens for mobile ad hoc networks. The algorithm is based on dynamic virtual rings formed by circulating tokens. It is inefficient to apply a self-stabilizing algorithm for static networks directly in a mobile ad hoc network because: (1) in self-stabilizing algorithms, a topology change is generally considered as a type of transient fault; (2) the algorithms usually require the topology to be static for converging to and staying in legitimate configurations. A mobile ad hoc network, which typically experiences frequent topology changes, would try to converge every time a topology change occurs and may never reach a legitimate configuration.

Chen and Welch [10] focus on the mutual exclusion problem and the idea is based on Dijkstra's self-stabilizing systems [13]. A mutual exclusion algorithm must satisfy two properties: *safety* (there is no more than one processor in the critical section) and *liveness* (every processor enters the critical section infinitely often). The network is required to be static while *ssME* is converging. After it has converged, under the given mobility constraints, safety and liveness are guaranteed. However, under the arbitrary mobility pattern, safety is guaranteed, while liveness is not guaranteed. The algorithm can be used in a mobile ad hoc network in which there are infinitely many static time intervals with lengths longer than the stabilisation time.

Lynch [41] considers the problem (in a two-node network) of implementing reliable FIFO communication using "less reliable channels". This includes channels that exhibit failures, such as the loss and duplication of messages, and channels that reorder messages. It also considers process crashes that lose process state information. In this reliable FIFO communication problem, processes can crash, losing information, and later recover. Lynch uses two well-known algorithms: Stenning's protocol and Alternating Bit protocol. In Stenning's protocol, the process at the sending end attaches (unbounded) integer tags to the message submitted by the user. This protocol tolerates loss, duplication, and reordering of messages on the channels. The Alternating Bit protocol use only bounded tags and tolerates loss and duplication, but not reordering.

The reliable FIFO problem is considered in an asynchronous send/receive network with underlying graph G consisting of two nodes, 1 and 2, connected by a single undirected edge. Liveness property as limitations on message loss is formulated as follows. *If infinitely many messages are sent, then infinitely many of them are delivered.* This property is formalised using two ways: strong loss limitation (SLL) and weak loss limitation (WLL).

> *In SLL, "if there are infinitely many send(m) events in* β *(for any particular m), then there are infinitely many send(m) events in the range of the cause function"* [41, p. 461].
> *In WLL, "if there are infinitely many send events in* β*, then the range of the cause function is infinite"* [41, p. 462].

This condition of restricting message loss does not mention any particular message *m*; it simply says that infinitely many sends cause receives of infinitely many messages. The difference between those two that the SLL condition specifies that the channel is fair to each particular type of message.

Feamster et al. [18] define correctness properties for internet routing, which include: *route validity*, *path visibility* and *safety*. *Route validity* means that if a router has a route to a destination, then a usable path corresponding to that route exists in the underlying topology. This route validity definition cannot be applied to dynamic networks because the underlying topology may change frequently. The route is valid with respect to the particular period of time in dynamic networks. *Path visibility* means that if there is a usable path between two nodes then the routing protocol will propagate information about that path. *Safety* means that the routing protocol converges to a stable route assignment (there are no persistent route oscillations).

Meanwhile, in dynamic networks, the routing information propagation must consider changes of neighbourhood over time. Therefore, correctness properties for dynamic network routing can be considered as follows. Firstly, *update visibility*, means that information about each link change or neighbour change is eventually received by all nodes. Secondly, *route validity* applies with respect to a period of time (snapshot). Thirdly, *safety* means that there is no persistent livelock.

2.7 Constant domain model

First-order modal logic [46] concerns with the logical interaction of the modal operators ($\Box$ and $\Diamond$) and the first-order quantifier ($\forall$ and $\exists$). In possible world semantics, this means the interaction of two types of quantifiers, one that ranges over possible worlds, and another that ranges over the objects in those worlds. A modal model contains many possible worlds. Therefore, the domain of the model can be fixed for the whole of a modal model, or can be varied from world to world.

A constant domain model has the domain to be fixed for the whole model. The domain of each possible world is equivalent each other. For example, in classical logic, the following formula is valid.

$$(\forall x)\varphi(x) \;\Rightarrow\; \varphi(x)$$

However, the validity of this formula in a modal model depends on which particular possible world semantics are used. In a modal model with a constant domain model, the formula holds. Moreover, the following formula also holds.

$$(\forall x)\varphi(x) \;\Rightarrow\; \varphi(y)$$

This is because in constant domain semantics, what exist in one world exists in all world.

To talk about things that do not exist but could, one solution from Melvin and Mendelsohn [46] is to open up the domain of the actual world to all possible objects, and keep the classical quantifier rules intact (this is called the *possibilist quantification*). The possibilist quantification corresponds to constant domain and it is evaluated for every element of the model $\mathcal{M}$. According to Melvin and Mendelsohn [46], constant domain semantics models the intuitions about modality if the domain consists of possible existence, not just actual ones. Otherwise, it would be required to treat every existent as a necessary existent. Some definitions related to constant domain models are as follows.

> *A structure $\langle \mathcal{G}, \mathcal{R}, \mathcal{D} \rangle$ is a constant domain augmented frame if $\langle \mathcal{G}, \mathcal{R} \rangle$ is a frame and $\mathcal{D}$ is a non-empty set, called the domain of the frame* [46, p. 95].

> *ℓ is an interpretation in a constant domain augmented frame $\langle \mathcal{G}, \mathcal{R}, \mathcal{D} \rangle$*

> *if ℓ assigns to each* n*-place relation symbol* R *and to each possible world* $\Gamma \in \mathcal{G}$*, some* n*-place relation on the domain $\mathcal{D}$ of the frame* [46, p. 95].
>
> *A constant domain first-order model is a structure $\mathcal{M} = \langle \mathcal{G}, \mathcal{R}, \mathcal{D}, \ell \rangle$ where $\langle \mathcal{G}, \mathcal{R}, \mathcal{D} \rangle$ is a constant domain augmented frame and ℓ is an interpretation in it* [46, p. 96].
>
> *A valuation in the model $\mathcal{M}$ is a mapping* v *that assigns to each free variable* x *some member* v(x) *of the domain $\mathcal{D}$ of the model* [46, p. 97].
>
> *Let* v *and* w *be two valuations.* w *is an* x-variant *of* v *if* v *and* w *agree on all variables except possibly the variable* x [46, p. 97].

This work of dynamic network models uses the constant domain model.

2.8 Frame problem

> *If we had a number of actions to be performed in sequence we would have quite a number of conditions to write down that certain actions do not change the values of certain fluents [properties and relationships]. In fact with* n *actions and* m *fluents, we might have to write down* mn *such conditions* [44, p. 31].

Afterwards, to guarantee an action only change some particular state while the other states remain the same as before, models utilise the frame problem.

However, some research then explores the possibilities to avoid the frame problem, particularly to avoid writing *mn* conditions, by recognising when there is no need of the frame problem. Instead, using a type of axioms as a replacement. In 1987, Andrew Haas introduced axioms governing the actions required to produce given types of changes. He named his axioms as “domain-specific frame axioms.” Schubert [56] proposed *explanation-closure axioms* in 1989 as the axioms provide complete sets of possible explanations for the given type of change. They use a model consisting of agents in a room: a human, a robot, and a cat in a box. The action of the human is *walking* from one location to a new location in the room. The actions of the robot are *holding, putdown, drop*, and *walk*. The cat has no action inside the box; it is moved only by the robot's action. Thus, if a, R, x, and s denote an action, the robot, the cat and situation respectively, the following axiom presented in [56]:

$$(\forall x, s, s')[[holding(R,x,s) \land \neg holding(R,x,s') \land s' = Result(a,s)] \rightarrow a = putdown(R,x)]$$

means that if the robot stops holding an object between situation s and s', and situation s' is a result of situation s by action a, then a must have been the action *putdown*. Moreover, the colour of the cat will not change when the robot is holding, then not holding the box. This is because the only action required here is the action *putdown*. There is no action involving painting the cat.

2.9 Runtime verification and RULER

A software failure can be viewed as a deviation between the observed behaviour and the required behaviour of the software system. A *fault* is defined as a deviation between the current behaviour and the expected behaviour, which is typically identified by a deviation of the current and the expected state of the system. A fault might lead to a failure, but not necessarily. An *error* is a mistake made by a human that results in a fault and possibly in a failure.

A typical approach for automatic verification of temporal properties of finite-state systems is as follows. Generate the state space of the systems and then apply a model checking algorithm for it to decide whether the system satisfies given temporal logic formulas.

Verification consists of all techniques suitable for showing that a system satisfies its specification. There are three traditional verification techniques [39]: theorem proving, model checking, and testing. Theorem proving allows to show correctness of programs, just as a proof in mathematics shows the correctness of a theorem. Model checking, which is an automatic verification technique, is mainly applicable to *finite-state* systems.

> *Model checking is a formal verification technique which allows for desired behavioural properties of a given system to be verified on the basis of a suitable model of the system through a systematic inspection of all states of the model* [3].

Testing covers a wide field of methods for showing correctness or for finding bugs. Each of these techniques has its trade-offs. Model checking requires a formal model.

Theorem proving over testing give stronger confidence.

Runtime verification is a lightweight verification technique complementing verification techniques such as model checking and testing and establishes another trade-off point between these forces. One of the main distinguishing features of runtime verification is that it is performed at runtime, which opens up the possibility of acting when incorrect behaviour of a software system is detected. Checking whether an execution meets a correctness property is typically performed using a monitor. A monitor decides whether the current execution satisfies a given correctness property by giving an output: either yes or true or no or false.

Leucker and Schallhart [39, p. 294] define runtime verification and monitor as follows:

> *Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property.*
> *A monitor is a device that reads a finite trace and yield a certain verdict.*

Runtime verification deals only with observed executions as they are generated by the real systems. Thus, runtime verification is applicable to black box systems for which no system model is at hand. However, when should runtime verification be used? Runtime verification can be used to easily check the actual execution of the system and make sure that the implementation meets its correctness properties. Moreover, sometimes the information is available only at runtime or is conveniently checked at runtime. In consequence, runtime verification may act as a complementary technique to theorem proving and model checking.

This research uses RULER, which is an experimental system. RULER, which has been developed by Barringer et al. [5], is a system that started as a low-level rule system into which one can compile different temporal specification logics used for runtime verification. The core of a RULER rule system is a collection of named rules. A rule consists of a condition (antecedent) and a body (consequent). A rule gets activated for the next evaluation step, then gets used and is automatically deactivated. This is called the *state* view in which a system remains in a particular state until some transition moves the system into another state. A trace of input observations can be checked for conformance against the rule system. RULER has three persistence attributes for rules: state persistence, always persistence and single shot persistence. Further explanation about the use of RULER can be found in Section 7.2.

2.10 Summary

Research in formal analysis of networks has been done over a wide field and a range of applications. However, only a few studies attempt to describe and reason about dynamic networks in high-level abstractions, or use the separation concern. This research is one attempt to build such a formal framework.

To describe the model, we use particular logics, such as temporal and Hoare logic, because they give certain advantages. Afterwards, we will formally define the model regarding the aim of the correctness, the type of domain model, and the frame problem. Moreover, this research describes dynamic networks as concurrent systems. Therefore, the model should tackle some of the issues in concurrent systems.

To reason about the model, and because the dynamic network model in this research involves concurrent systems, we will focus on some techniques for concurrent systems as well. Furthermore, the use of runtime verification, especially the experimental tool RULER, may complete this research of dynamic networks. The following chapter will set out how to describe and reason about dynamic network systems by using an introductory example.

To express the action of each concurrent process, we will use Hoare logic. To develop an algorithm using LTL properties. For further research, using Hoare logic to link the particular routing algorithms.

Chapter 3

Methodology and preliminary study

Before modelling the dynamic networks as concurrent systems, it is useful to understand how to model such systems through an example. This chapter sets out the methodology being adopted for obtaining an understanding of how to describe and reason about dynamic networks. We will start with the analogy of dynamic networks to gain a brief understanding of dynamic behaviours, before going through the preliminary study. The preliminary study involves on-the-fly sorting, which is a first step towards a view of how to reason about dynamic network systems.

3.1 Methodology

This methodology uses the analogy between a dynamic network system and on-the-fly sorting, as an early step towards reasoning about dynamic network system. The analogy is based on the principle that there are two processes run simultaneously in the system: one process as an *organiser* and the other process as a *disrupter*. In the next stage of this research, this study can be extended for use an on-the-fly-sorting algorithm with a set of values, or use a different mechanism until it is similar to an actual dynamic network system. Moreover, we will directly point out the applicability to a preliminary study with dynamic network systems and then attempt to describe and reason about an actual dynamic network system.

There is a discussion about how to prove an on-the-fly sorting algorithm which uses shared variables. Data resides in a single shared address space, communicated through shared read and write, and uses explicit synchronisation. The correctness properties that are being considered are safety and liveness.

3.2 The dynamic networks analogy

A dynamic network system can be viewed as a system with two main processes. One process has a role in organising the system in case there is a disruption to it, such as disconnection of a link or a routing update. Another process has a role in disrupting the connection between nodes of network systems, which has an impact on routing. We will consider these conditions as if something always attempts to cut the connection between any nodes.

There is an *organiser* and *disrupter* relationship as well in on-the-fly nature as in on-the-fly garbage collection [12, 21, 7] problem. This research uses on-the-fly sorting, where numbers in an array should always be ascending order, while a new number continuously overwrites the old number. This can be considered as the network connection that has been cut or interrupted infinitely often randomly and network routing should have its correctness at some point.

3.3 Introductory example: On-the-fly (OTF) sorting

We can use the On-the-fly (OTF) sorting case as a preliminary example of reasoning about a dynamic network system.

3.3.1 An OTF sorting problem

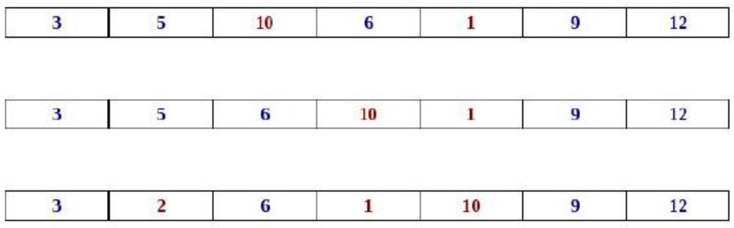


Figure 3.1: An illustration of on-the-fly sorting.

Suppose there is an array of integers. An element of the array can have the colour blue or red. *Blue colour* means that the integer is in order position (in this case, ascending order). *Red colour* means that the integer is not in the appropriate position, or is a new input number from the array to be sorted. Figure 3.1 shows three conditions of the array. The first array shows the initial condition, in which there are two red elements in position three and five within the array, with values 10 and 1, respectively. The other elements are blue and their values are given in ascending order. The second array shows that when "10" swaps position with "6" to establish the ascending order, "10" still needs to swap with "1". The third array shows "10" swapping position with "1" to establish ascending order. However, the second element of the array has changed its value to "2", which means that it is no longer in ascending order. Input for this sorting problem is an array *A[0..n-1]* of integers, with length *N*. The problem is how to prove that this array will be kept sorted while there are new input integers to be placed in the array.

This example uses two processors as in OTF garbage collection [12, 21, 7], which uses a *mutator* and a *collector*. The *mutator* changes the reachability of a node through deleting or changing the outgoing edge and adding a node from empty list, while the *collector* collects unreachable nodes and puts them in the heap. Dijkstra et al. [12] use these two processors, in which the *collector* has two phases: the marking phase and the collecting phase. The marking phase uses three colours: white (to denote a free node), black (to denote an accessible node from root), and grey (to denote a transition node, either white or black). The only synchronisation is that the *mutator* has to wait until there are two nodes in the empty list.

OTF sorting uses the terminology a *feeder* (which attempts to feed new integers into an array) and a *sorter* (which attempts to place the integer in an ascending order position). The activities of *feeder* and *sorter* are repeated executions in parallel. For the proof of safety no fairness condition on these two processes are required. A *feeder* gives a new integer repeatedly by replacing a random element in the array. The *sorter* attempts to rearrange the elements in ascending order and when it is confident that an element is in the right place (in order), the *sorter* will colour it blue. Our aim is to give an algorithm for the *sorter* and to prove that it is correct, using both safety and liveness properties. The elements of array have two fields:

- A[i].value as the current value of the element *i*
- A[i].colour as the current colour of the element *i*

The ***feeder*** is described below (let *j* be an index of array **A** and **num** is any positive integer number > 0) :

```
while (true)
  j := generate_random(a, c, n);
  num := generate_random(a, c, 100);
  A[j].value = num;
  A[j].colour = 'Red';
```

The ***sorter*** treats elements based on their position as follows, with $0 \leq i \leq N-1$.

```
while (true)
  for (0 <= i < numArrayItem)
  /* or use random i to pick element randomly */
    if (A[i].colour = Red)
      lock A[i];
      if A[i] = N-1
        process of the last element in array;
      else
        if A[i] = 0
              process of the first element in array;
        else
              process of other elements in array;
       release A[i];
```

Processing of the last element in array involves swapping and the three conditions *rb1*, *rb2* and *rb8*.

```
        if A[i].value < A[i-1].value
          SwapXY action;
          if A[i-2].value < A[i-1].value
          and ((A[i-2].colour = Blue) or (A[i].colour = Blue))
            rb1 condition;
        else
          if A[i-2].value < A[i-1].value
          ((A[i-2].colour = Blue) or (A[i-1].colour = Blue))
             rb2 condition;
          else if all j != i are red, then
             rb8 condition;
```

Processing of the first element in the array involves swapping and the two conditions *rb3* and *rb4*.

```
    if A[i].value > A[i+1].value
      SwapXY action;
      if A[i].value < A[i+1].value < A[i+2].value
      and ((A[i].colour = Blue) or (A[i+2].colour = Blue))
        rb3 condition;
```

```
else
  if A[i].value < A[i+1].value < A[i+2].value
  and ((A[i+1].colour = Blue) or (A[i+2].colour = Blue))
      rb4 condition;
```

Processing of the other elements in the array involves swapping and the three conditions *rb5, rb6* and *rb7.*

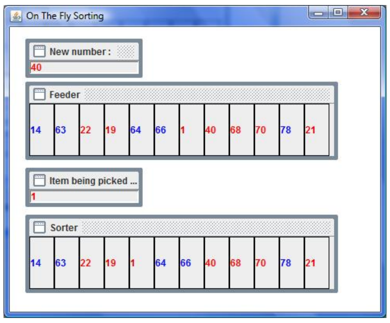
```
if A[i-1].value <= A[i].value <= A[i+1].value
and (A[i+1].colour = Blue and A[i-1].colour = Blue)
  rb5 condition;
else
  if A[i-1].value > A[i].value < A[i+1].value
  and A[i-1].colour = Blue and A[i+1].colour = Blue
    SwapXY action;
    if A[i-2].value < A[i-1].value < A[i].value
    and A[i-2].colour = Blue and A[i].colour = Blue
        rb6 condition;
   else  /* element is at an appropriate position */
        rb7 condition;
  else
if predecessor or successor not blue then
       SwapXY action; /* swap only */
```

Details of the algorithm can be found in Appendix A.1.

Notice that there are eight points within the algorithm (marked by *rb1, rb2, rb3, rb4, rb5, rb6,rb7* and *rb8*), in which the colour of an element can turn to blue if one of the eight conditions is true. These represent the possible selection of conditions for colouring elements blue. This algorithm can be applied to systematic sorting (sorting run through the first element to the last element within an array) as well as random sorting (picking a random index and then sorting the element if needed). The *lock* command of the element of array means that the state of an element, such as value, location and colour, cannot be changed by the feeder. After the *release* command has been applied to the element, this state of that element can change.

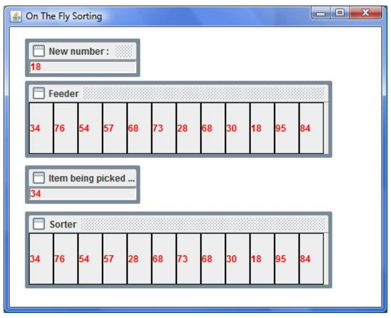
A sufficient condition for ensuring that there are enough blue elements in this algorithm depends on the location of the current element which is being sorted. The first and last elements in the array have similar sufficient conditions for being blue, which holds as long as the current element in the right place (in order) and at least one of the adjacent element is blue. Elements between the first and last element can be blue if only the current element is in ascending order and its two adjacent elements are blue. In the proof in Section 3.3.2, there are the general conditions for these possible cases, of which those in the algorithm and special cases.

The *feeder* and *sorter* in this algorithm are executed in parallel. The fairness constraint is that no action can be ignored forever, which means that the *feeder* and *sorter* will always have their turn. Furthermore, each action within the two processes will always have their turn as well to be executed. Some example outputs of the algorithm implementation are presented in Figures 3.2 and 3.3.



```
On The Fly Sorting
New number :
40
Feeder
14 63 22 19 64 66 1 40 68 70 78 21
Item being picked ...
1
Sorter
14 63 22 19 1 64 66 40 68 70 78 21
```

Figure 3.2: An example of output from an OTF sorting program, when the *feeder* puts "40" as new number and the *sorter* picks "1" to be placed in ascending order position



```
On The Fly Sorting
New number :
18
Feeder
34 76 54 57 68 73 28 68 30 18 95 84
Item being picked ...
34
Sorter
34 76 54 57 28 68 73 68 30 18 95 84
```

Figure 3.3: An example of output from OTF sorting program, when all elements in the array are unsorted and red

3.3.2 Proof of correctness

This section discusses the correctness criteria of the OTF sorting algorithm. The *feeder* and *sorter* processes must cooperate such that the following correctness criteria are satisfied:

- CC1: The blue elements are always in ascending order (*safety*).

- CC2: On all infinite sequence of actions, every red element becomes blue eventually, or gets its value changed by the feeder (*liveness*). This requires a suitable notion of fair interaction of the processes.

One of the difficulties of this formalisation of OTF sorting is that the elements get over-written by the feeder. There are several different logical approaches to this disappearance of elements and new appearance of new ones. The algorithm shows that there are changes in colour, value, or location of particular elements when they execute particular actions. There are some possibilities for specifying the changes logically in terms of *"overwriting"*.

1. Elements *persist*, which means that there is a fixed number of elements with value, colour, and location. This possibility deals with the element and frame problems when describing the specification. Frame problems address which properties of elements have been affected by the execution of an action and which properties of the elements persist.

2. Elements are *destroyed and created* with a fixed value but their colour and location can change. This possibility deals with the existence and non-existence of objects, which is hard to express in logic.

For this proof the simplest approach is to consider all elements as persisting and that the *feeder* changes an element's value and (possibly) colour. Otherwise, there is a need to deal logically with elements that may disappear and so a need to handle "existence" in a logical framework.

3.3.2.1 Array of coloured elements

Suppose there is a set S of elements. Consider a fixed size array of N elements indexed $I = \{0..N-1\}$. Some notations for describing arrays of coloured elements are as follows.

- $x@i$ means element x is at index i
- $R(x)$ means the colour of element x is *red*
- $B(x)$ means the colour of element x is *blue*

Axioms satisfied by these formulas are as follows.

Axioms 1 *(Array of coloured elements). For x, y $\in$ S and i, j $\in$ I*

$$(x@i \wedge y@i) \Rightarrow (x = y)$$
$$(x@i \wedge x@j) \Rightarrow (i = j)$$
$$R(x) \Leftrightarrow \neg B(x)$$
$$B(x) \Leftrightarrow \neg R(x)$$

3.3.2.2 Actions

The actions are described by so-called Hoare triples [30],

$$\{\varphi\}P\{\psi\}$$

where P is a program built from a set of actions, φ and ψ are formulas. This triple can be read as " If execution is begun anywhere within P with φ true, then φ will remain true until P terminates and ψ will be true when P terminates".

The basic actions are described as follows. The first action is Red(x, i), which the action of re-colouring, in which a blue element x at i turns into a red element when the value of element x is changed randomly. The location of element x, as well as properties of other elements, remains unchanged.

Axioms 2 *Action* Red*(x, i) : For all x $\in$ S, for all i $\in$ I, and for all $n, m \in \mathbb{Z}$ where $n \neq m$*

$$\{x@i\}\ \mathsf{Red}(x,i)\ \{R(x)\}$$

Frame axioms for x

$$\{x@i\}\ \mathsf{Red}(x,i)\ \{x@i\}$$

For all y $\in$ S, where $y \neq x$, for all $i, j \in I$, where $j \neq i$, and for all $n \in \mathbb{Z}$

$$\{R(y) \wedge y@j \wedge x@i\}\ \mathsf{Red}(x,i)\ \{R(y)\}$$
$$\{B(y) \wedge y@j \wedge x@i\}\ \mathsf{Red}(x,i)\ \{B(y)\}$$
$$\{y@j \wedge x@i\}\ \mathsf{Red}(x,i)\ \{y@j\}$$
$$\{V(y) = n \wedge y@j \wedge x@i\}\ \mathsf{Red}(x,i)\ \{V(y) = n\}$$

The second action is Blue(x, i), in which an element x turns into blue under certain preconditions. The value and location of element x as well as properties of other elements remain unchanged.

Axioms 3 *Action* Blue*(x, i): For all* $x \in S = \{w..z\}$*, for all* $i \in I = \{0..N-1\}$*, and for all* $n, m \in \mathbb{Z}$ *where* $n \neq m$

$$\{x@i\}\ \mathsf{Blue}(x,i)\ \{B(x)\}$$

Frame axioms for x

$$\{V(x) = n \wedge x@i\}\ \mathsf{Blue}(x,i)\ \{V(x) = n\}$$
$$\{x@i\}\ \mathsf{Blue}(x,i)\ \{x@i\}$$

For all $x, y \in S$*, where* $y \neq x$*, for all* $j \in I$*, where* $j \neq i$*, and for all* $n \in \mathbb{Z}$

$$\{R(y) \wedge x@i\}\ \mathsf{Blue}(x,i)\ \{R(y)\}$$
$$\{B(y) \wedge x@i\}\ \mathsf{Blue}(x,i)\ \{B(y)\}$$
$$\{y@j \wedge x@i\}\ \mathsf{Blue}(x,i)\ \{y@j\}$$
$$\{V(y) = n \wedge x@i\}\ \mathsf{Blue}(x,i)\ \{V(y) = n\}$$

The third action is Swap(x, y), in which an element x exchanges its location with element y. The values and colours of element x and element y as well as the properties of other elements remain unchanged.

Axioms 4 *Action* Swap*(x, y): For all* $x, y \in S$ *where* $x \neq y$*, for all i,j* $\in I$*, and for all n,* $m \in \mathbb{Z}$

$$\{x@i \wedge y@j\}\ \mathsf{Swap}(x,y)\ \{x@j \wedge y@i\}$$

Frame axioms for x *and* y

$$\{V(x) = n \wedge x@i \wedge V(y) = m \wedge y@j\}\ \mathsf{Swap}(x,y)\ \{V(x) = n \wedge V(y) = m\}$$
$$\{R(x) \wedge x@i \wedge R(y) \wedge y@j\}\ \mathsf{Swap}(x,y)\ \{R(x) \wedge R(y)\}$$
$$\{R(x) \wedge x@i \wedge B(y) \wedge y@j\}\ \mathsf{Swap}(x,y)\ \{R(x) \wedge B(y)\}$$

Frame axioms for z $\neq$ x *and* z $\neq$ y
For all $x, y, z \in S$*, where* $z \neq x$ *and* $z \neq y$*, for all* $k \in I$*, where* $k \neq i$ *and* $k \neq j$*, and for*

all $n \in \mathbb{Z}$

$$\{x@i \wedge y@j \wedge z@k\}\ \mathsf{Swap}(x,y)\ \{z@k\}$$
$$\{B(z) \wedge z@k \wedge x@i \wedge y@j\}\ \mathsf{Swap}(x,y)\ \{B(z)\}$$
$$\{R(z) \wedge z@k \wedge x@i \wedge y@j\}\ \mathsf{Swap}(x,y)\ \{R(z)\}$$
$$\{V(z) = n \wedge z@k \wedge x@i \wedge y@j\}\ \mathsf{Swap}(x,y)\ \{V(z) = n\}$$

3.3.2.3 Hoare triple axioms

Pre-condition weakening rule

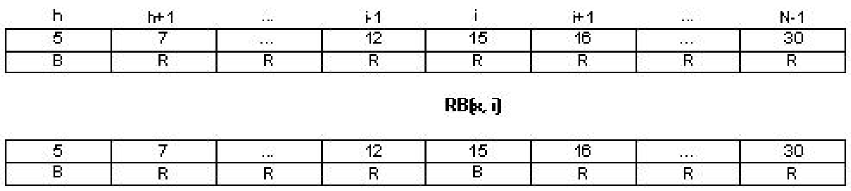
$$\frac{\varphi\ \{A\}\ \theta \quad \theta \rightarrow \psi}{\varphi\ \{A\} \psi}\ (Weaken\ PostCondition) \qquad \frac{\varphi \rightarrow \theta \quad \theta\ \{A\}\ \psi}{\varphi\ \{A\}\ \psi}\ (Strengthen\ Precondition)$$

Sequential Composition

$$\frac{\varphi_1\ \{A_1\}\ \psi \quad \psi\ \{A_2\}\ \varphi_2}{\varphi_1\ \{A_1;A_2\}\ \varphi_2}\ (SequentialComposition)$$

There are four general conditions which can occur before the execution of action Blue(x, i), which will allow colouring. The following are illustrations of these expressed as formulas. This is abbreviated: Ordered(w,x,y) = $(V(w) \leq V(x)) \wedge (V(x) \leq V(y))$.

1. **RBRight** (turn to blue in the right direction), a condition when the colour of the element at location h is blue and all values beyond and including that at h are in ascending order, as shown in Figure 3.4. The formula of this condition:



h	h+1	...	i-1	i	i+1	...	N-1
5	7	...	12	15	16	...	30
B	R	R	R	R	R	R	R

RB(x, i)

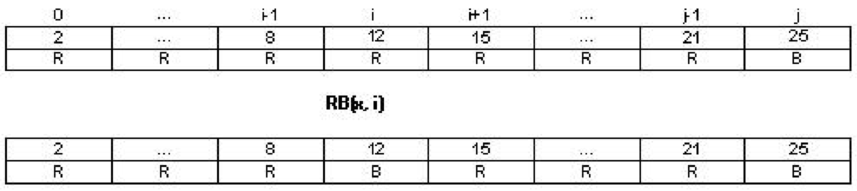
5	7	...	12	15	16	...	30
B	R	R	R	B	R	R	R

Figure 3.4: An illustration of RBRight.

For all $h, i \in I$ where $h < i < N-1$ and for all $w,x,y \in S$

$$B(w) \wedge w@h \wedge R(x) \wedge x@i \wedge R(y) \wedge y@[i+1] \wedge \mathsf{Ordered}(w,x,y)$$

2. **RBLeft** (turn to blue in the left direction): a condition when the colour of the element at location j is blue and all values in between 0 and j are in ascending order, as shown in Figure 3.5. The formula of this condition:



0	...	i-1	i	i+1	...	j-1	j
2	...	8	12	15	...	21	25
R	R	R	R	R	R	R	B

RB(x, i)

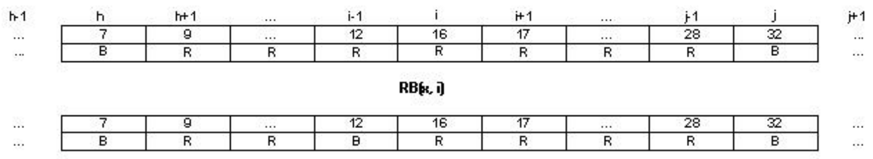
2	...	8	12	15	...	21	25
R	R	R	B	R	R	R	B

Figure 3.5: An illustration of RBLeft

For all $i, j \in I$ where $0 < i < j$ and for all $w, x, y \in S$

$$R(w) \wedge w@[i-1] \wedge R(x) \wedge x@i \wedge B(y) \wedge y@j \wedge \mathsf{Ordered}(w, x, y)$$

3. **RBMiddle** (turn to blue for an item between two blue items): a condition when the colour of the element at the location is blue and all values in between h and j are in ascending order, as shown in Figure 3.6. The formula of this condition:



h-1	h	h+1	...	i-1	i	i+1	...	j-1	j	j+1
...	7	9	...	12	16	17	...	28	32	...
...	B	R	R	R	R	R	R	R	B	...

RB(x, i)

...	7	9	...	12	16	17	...	28	32	...
...	B	R	R	B	R	R	R	R	B	...

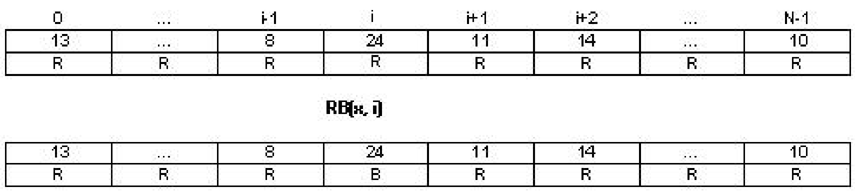
Figure 3.6: An illustration of RBMiddle

For all $h, i, j \in I$ where $h < i < j$ and for all $w, x, y \in S$

$$B(w) \wedge w@h \wedge R(x) \wedge x@i \wedge B(y) \wedge y@j \wedge \mathsf{Ordered}(w, x, y)$$

4. **RBAny** (turn to blue for an item in any position, if all items are red): a condition when the colour of all elements in the array in between 0 and $N-1$ is red, as shown in Figure 3.7. The formula of this condition:
   For all $i \in I$ where $0 < i \leq N-1$ and for all $x, y \in S$

$$R(y) \wedge y@[i-1] \wedge R(x) \wedge x@i$$



0	...	i-1	i	i+1	i+2	...	N-1
13	...	8	24	11	14	...	10
R	R	R	R	R	R	R	R

RB(x, i)

13	...	8	24	11	14	...	10
R	R	R	B	R	R	R	R

Figure 3.7: An illustration of RBAny

3.3.2.4 Actions of the algorithm

Now notice the algorithm which uses eight actions as the family of Blue(x, i), four actions as the family of Swap(x, y), and two actions as the family of Red(x, i). Those possible actions which occur within the OTF sorting algorithm in Section 3.3.1 are:

1. Item replacement, in which the colour and value of item are changed by the *feeder*. This item replacement can be executed using one of these two actions:

   (a) B(lue) $\rightarrow$ R(ed), which is denoted as BR(x, i), or

   (b) R(ed) $\rightarrow$ R(ed), which is denoted as RR(x, i), which means that the colour of the item remains red.

   Axioms:

   - axiom of change for BR(x, i) (based on axioms of Red(x, i), with $i \in I$ and $n \in \mathbb{Z}$)

     (a) the colour of x will change to red after the action

     $$\{B(x) \wedge x@i\}\ \mathsf{BR}(x,i)\ \{R(x) \wedge x@i\}$$

     Frame axioms for x

     $$\{(V(x)=n) \wedge x@i\}\ \mathsf{BR}(x,i)\ \{x@i \wedge (V(x)=n \vee V(x) \neq n)\}$$
     $$\{x@i\}\ \mathsf{BR}(x,i)\ \{x@i\}$$

     (b) the blue element x at i is not affected if there is a change of colour from blue to red at j (if $i \neq j$)

     $$\{B(x) \wedge x@i\}\ \mathsf{BR}(x,j)\ \{B(x) \wedge x@i\}$$
     $$\{(V(x)=n) \wedge x@i\}\ \mathsf{BR}(x,j)\ \{(V(x)=n) \wedge x@i\}$$

(c) action $\mathsf{BR}(x, i)$ does not affect item x when colour of x is Red

$$\{R(x) \wedge x@i\}\ \mathsf{BR}(x,i)\ \{R(x) \wedge x@i\}$$

- frame axioms for $A = \mathsf{BR}(x, i)$ have been defined by axioms of $\mathsf{Red}(x, i)$.
- axioms of change for $\mathsf{RR}(x, i)$ (based on axioms of $\mathsf{Red}(x, i)$, with $i \in I$ and $n \in \mathbb{Z}$)

  (a) the colour of x will remain red after the action

  $$\{R(x) \wedge (V(x) = n) \wedge x@i\}\ \mathsf{RR}(x,i)\ \{R(x) \wedge V(x) \wedge x@i\}$$

  (b) the Red element x at i is not affected if there is an action $\mathsf{RR}(x, j)$ at j (if $i \neq j$)

  $$\{R(x) \wedge (V(x) = n) \wedge x@i\}\ \mathsf{RR}(x,j)\ \{R(x) \wedge (V(x) = n) \wedge x@i\}$$

  (c) the action $\mathsf{RR}(x, i)$ does not affect element x when the colour of x is Blue

  $$\{B(x) \wedge (V(x) = n) \wedge x@i\}\ \mathsf{RR}(x,i)\ \{B(x) \wedge (V(x) = n) \wedge x@i\}$$

- frame axioms for $A = \mathsf{RR}(x, i)$ have been defined by axioms of $\mathsf{Red}(x, i)$.

2. swap elements by *sorter*, in which **only the location is changed**. This swap action is an instance of Swapand is denoted as SwapXY. The various possibilities for the swap action are:

   (a) **swap a Blue element with a Blue element**. The algorithm never swaps a Blue element with a Blue element, because if there is a swap between Blue elements then this means the elements are out of order. Furthermore, the sorter finds a Red element first before going through the sorting phase. Therefore, at least one element in swap action must be a Red element.

   (b) **swap a Red element with a Blue element**. The movement does not change the order of Blue elements.

   (c) **swap a Blue element with a Red element**. The movement does not change the order of Blue elements.

   (d) **swap a Red element with a Red element**. No Blue element is moved.

The axiom of change for SwapXY(x, y) (based on axioms of Swap(x, y), with $i \in I$ and $n,m \in \mathbb{Z}$) between current element and its successor:

$$\{x@i \wedge y@[i+1]\}\ \mathsf{SwapXY}(x,y)\ \{x@[i+1] \wedge y@i\}$$
$$\{V(x) = n \wedge x@iV(y) = m \wedge y@[i+1]\}$$
$$\mathsf{SwapXY}(x,y)$$
$$\{V(x) = n \wedge x@[i+1]V(y) = m \wedge y@i\}$$
$$\{R(x) \wedge x@i \wedge B(y) \wedge y@[i+1]\}\ \mathsf{SwapXY}(x,y)\ \{R(x) \wedge x@[i+1] \wedge B(y) \wedge y@i\}$$
$$\{R(x) \wedge x@i \wedge R(y) \wedge y@[i+1]\}\ \mathsf{SwapXY}(x,y)\ \{R(x) \wedge x@[i+1] \wedge R(y) \wedge y@i\}$$

For swap between the current element and its predecessor:

$$\{x@i \wedge y@[i-1]\}\ \mathsf{SwapXY}(x,y)\ \{x@[i-1] \wedge y@i\}$$
$$\{V(x) = n \wedge x@iV(y) = m \wedge y@[i-1]\}$$
$$\mathsf{SwapXY}(x,y)$$
$$\{V(x) = n \wedge x@[i-1]V(y) = m \wedge y@i\}$$
$$\{R(x) \wedge x@i \wedge B(y) \wedge y@[i-1]\}\ \mathsf{SwapXY}(x,y)\ \{R(x) \wedge x@[i-1] \wedge B(y) \wedge y@i\}$$
$$\{R(x) \wedge x@i \wedge R(y) \wedge y@[i-1]\}\ \mathsf{SwapXY}(x,y)\ \{R(x) \wedge x@[i-1] \wedge R(y) \wedge y@i\}$$

Frame axioms for A = SwapXY(x, y) has been defined by axioms of Swap(x, y).

3. colour change to determine the ordered position of elements, in which **only colour is changed** from red to blue by the *sorter*. The value of the elements remains the same. This action is denoted RB(x, i). Action RB(x, i) still needs to be justified, because this action can only be executed under certain conditions. In general, there are four conditions in the family of OTF sorting algorithm that will determine whether an element changes its colour to blue. According to the OTF sorting algorithm in Section 3.3.1, there are more specific conditions for changing an element from red to blue (where i is the current element that being sorted and N is the size of the array). These more specific conditions can be derived from the general conditions, as mentioned in Section 3.3.2, as follows.

   (a) **rb1**, in which x is the final red element in the array, and the value of x at i is less than value of y at $i-1$. Elements will be ordered after action SwapXY(x, y), as shown in Figure 3.8. There are three possible conditions:

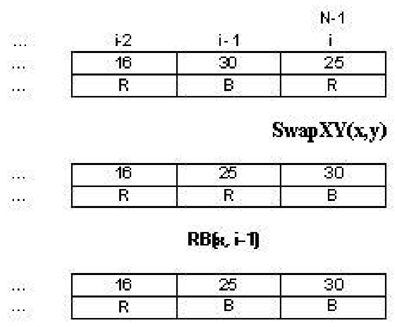


Figure 3.8: An illustration of rb1.

- *rb1a*, which is a special case of RBLeft at Section 3.3.2.

$$R(x) \wedge x@[i-1] \wedge B(y) \wedge y@i \wedge R(z) \wedge z@[i-2] \wedge \mathsf{Ordered}(z,x,y)$$

- *rb1b*, which is a special case of RBRight at Section 3.3.2.

$$R(x) \wedge x@[i-1] \wedge R(y) \wedge y@i \wedge B(z) \wedge z@[i-2] \wedge \mathsf{Ordered}(z,x,y)$$

- *rb1c*, which is a special case of RBMiddle at Section 3.3.2.

$$R(x) \wedge x@[i-1] \wedge B(y) \wedge y@i \wedge B(z) \wedge z@[i-2] \wedge \mathsf{Ordered}(z,x,y)$$

(b) **rb2**, in which x is red, x is the last element in the array, and at least one of two previous elements is blue and ordered. This is shown in Figure 3.9. There are three possibilities (one of the previous elements is blue or both of

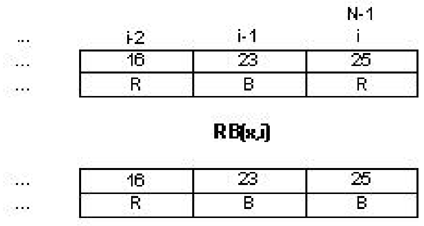


Figure 3.9: An illustration of rb2.

them are blue) and the formulas of rb2 can be derived from RBRight (rb2 formulas are the special cases of RBRight). The formulas of rb2 are:

$$R(x) \wedge x@i \wedge B(y) \wedge y@[i-1] \wedge R(z) \wedge z@[i-2] \wedge \mathsf{Ordered}(z,y,x)$$
$$R(x) \wedge x@i \wedge R(y) \wedge y@[i-1] \wedge B(z) \wedge z@[i-2] \wedge \mathsf{Ordered}(z,y,x)$$
$$R(x) \wedge x@i \wedge B(y) \wedge y@[i-1] \wedge B(z) \wedge z@[i-2] \wedge \mathsf{Ordered}(z,y,x)$$

(c) **rb3**, in which x is red, x is the first element, at least one of two adjacent elements is blue and the value of x at i is greater than value of y at $i+1$. Elements will be ordered after action SwapXY(x, y), as shown in Figure 3.10. There are three possible conditions:

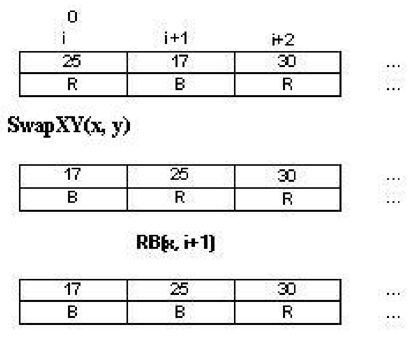


Figure 3.10: An illustration of rb3.

- *rb3a*, as a special case of RBRight

$$R(x) \wedge x@[i+1] \wedge B(y) \wedge y@i \wedge R(z) \wedge z@[i+2] \wedge \mathsf{Ordered}(z,x,y)$$

- *rb3b*, as a special case of RBLeft

$$R(x) \wedge x@[i+1] \wedge R(y) \wedge y@i \wedge B(z) \wedge z@[i+2] \wedge \mathsf{Ordered}(z,x,y)$$

- *rb3c*, as a special case of RBMiddle

$$R(x) \wedge x@[i+1] \wedge B(y) \wedge y@i \wedge B(z) \wedge z@[i+2] \wedge \mathsf{Ordered}(z,x,y)$$

(d) **rb4**, in which x is red, x is the first element in the array, at least one of two successor elements is blue, and ordered. This is shown in Figure 3.11. There are three possibilities (one of the successor elements is blue or both

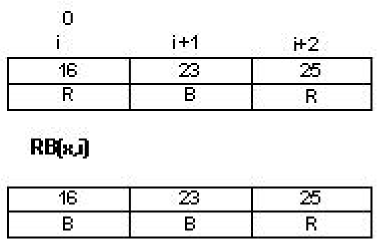


Figure 3.11: An illustration of rb4.

of them are blue) and the formulas of rb4 can be derived from RBLeft (rb4 formulas are the special cases of RBLeft). The formulas for rb4 are:

$$R(x) \wedge x@i \wedge B(y) \wedge y@[i+1] \wedge R(z) \wedge z@[i+2] \wedge \mathsf{Ordered}(x,y,z)$$
$$R(x) \wedge x@i \wedge R(y) \wedge y@[i+1] \wedge B(z) \wedge z@[i+2] \wedge \mathsf{Ordered}(x,y,z)$$
$$R(x) \wedge x@i \wedge B(y) \wedge y@[i+1] \wedge B(z) \wedge z@[i+2] \wedge \mathsf{Ordered}(x,y,z)$$

(e) **rb5**, in which x is one of the other elements in the array and ordered, as shown in Figure 3.12. The formula of this condition is derived from RB-

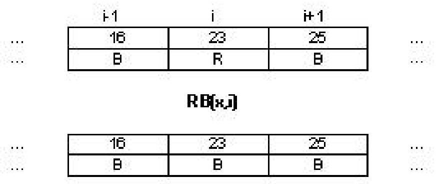


Figure 3.12: An illustration of rb5.

Middle (rb5 is a special case of RBMiddle):

$$R(x) \wedge x@i \wedge B(y) \wedge y@[i-1] \wedge B(z) \wedge z@[i+1] \wedge \mathsf{Ordered}(y,x,z)$$

(f) **rb6**, in which x is one of the other elements in the array and the value of x at i is less than value of y at $i-1$, then after action $\mathsf{SwapXY}(x,\ y)$, the elements will be ordered and x will not be the first element in the array. This is shown in Figure 3.13. The formula of this condition is derived from

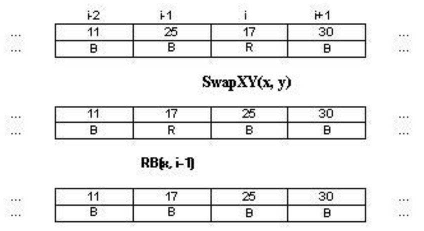


Figure 3.13: An illustration of rb6.

RBLeft (rb6 is a special case of RBLeft):

$$R(x) \land x@[i-1] \land B(y) \land y@i \land B(w) \land w@[i-2] \land \mathsf{Ordered}(w,x,y)$$

(g) **rb7**, in which x is one of the other elements in the array and the value of x at i is less than value of y at $i-1$, then after action $\mathsf{SwapXY}(x, y)$, the elements will be ordered and x will become the first element of array. This is shown in Figure 3.14. The formula of this condition is derived from RBLeft (rb7

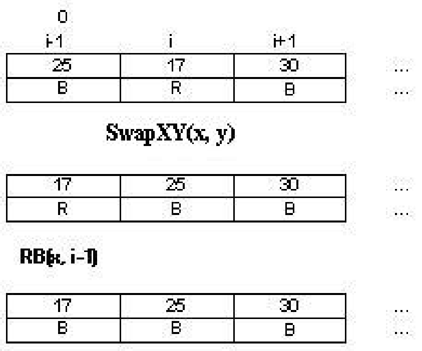


Figure 3.14: An illustration of rb7.

is a special case of RBLeft):

$$R(x) \land x@[i-1] \land B(y) \land y@i \land B(z) \land z@[i+1] \land \mathsf{Ordered}(x,y,z)$$

(h) **rb8**, in which colour of all elements, including x, is red, as shown in Figure 3.7. The formula for this condition is derived from RBAny (rb8 is a special case of RBAny):

$$R(x) \land x@[i+1] \land R(y) \land y@i$$

Therefore, the axioms of change for RB(x, i) (based on axioms of Blue(x, i)), with $rb \in \{rb1..rb8\}$:

- Red element x at i is not affected by an action RB(x, j), if $i \neq j$

$$\{R(x) \wedge (V(x) = n) \wedge x@i\}$$
$$\mathsf{RB}(x, j)$$
$$\{R(x) \wedge (V(x) = n) \wedge x@i\}$$

- the colour of x will change to Blue after the action

$$\{R(x) \wedge (V(x) = n) \wedge x@i \wedge rb\}$$
$$\mathsf{RB}(x, i)$$
$$\{B(x) \wedge (V(x) = n) \wedge x@i\}$$

- the action RB(x, i) does not affect element x when colour of x is Blue

$$\{B(x) \wedge x@i \wedge (V(x) = n)\}$$
$$\mathsf{RB}(x, i)$$
$$\{B(x) \wedge x@i \wedge (V(x) = n)\}$$

Frame axioms for A = RB(x, i) has been defined in axioms of Blue(x, i).

There are basic actions and atomic actions. Section 3.3.1 discusses basic actions and specific basic actions for the algorithm in this section. An atomic actions are those which cannot be interfered with other processes. For the algorithm, these are:

- SwapToBlue(x, i, y, j) $\equiv$ SwapXY(x, y); RB(x, j)
- The other actions in the algorithm, such as: BR(x, i), RR(x, i) and RB(x, i)

3.4 Safety proof

The OTF sorting algorithm has to ensure that "blue elements are always in order". This is expressed as an *invariant* φ.

For all x, $y \in S$ where $x \neq y$ and for all i, $j \in I$,

$$\boxed{x@i \wedge y@j \wedge B(x) \wedge B(y) \wedge (i \leq j) \Rightarrow (V(x) \leq V(y))}$$

with the initial condition for the algorithm (all elements are blue and ordered)

$$\boxed{\forall x \in S.(B(x)) \wedge \forall i \in I.\forall x,y \in S.(i < j \wedge x@i \wedge y@j \Rightarrow V(x) \leq V(y))}$$

Prove that, for every action A:

1. The initial condition of the array $\Rightarrow \varphi$
2. $\{ \varphi \} A \{ \varphi \}$

Then conclude that for any sequence of actions $A_1, ..., A_n$

$$A_1; ...; A_n \ \{\varphi\}$$

The proof of safety property proceeds as follows. First, the initial condition of the array is that all elements are in ascending order and blue so φ holds. Now, prove that for each action, φ holds before and after the action.

Lemma 1 ({ φ } BR(x, i) { φ }) .
Proof. *The order of the blue elements in the array can be affected if the value of x is replaced by the action* BR*(x, i) or* RR*(x, i). Given φ holds before the action* BR*(x, i), by the axiom of change in action* BR*(x, i) and by definition of* BR*(x, i), then the colour of x is changed into red and other items are persist. Hence, by the uniqueness of the element as well, φ holds as well after the action* BR*(x, i). The proof for action* RR*(x, i) is similar.* ■

Lemma 2 ({ φ } SwapXY(x, y) {φ }) .
Proof. *Given that the value of element x at i is greater than element y at $i+1$ and y is blue. Show that if φ holds before the action* SwapXY*(x, y) then φ also holds after the action. By uniqueness, axioms of* Swap*(x, y), and axioms of change for* SwapXY*(x, y), then after the action* SwapXY*(x, y), element x exchanges its location with y but the value and colour persist, as well as property of the other elements. Furthermore, since the location of other elements persist, the order of blue elements is not affected after action* SwapXY*(x, y). Therefore, φ holds after the action. The proof for action* SwapXY*(x, y) for a swap between current element and its predecessor is similar.* ■

Lemma 3 ($\{rb1 \wedge \varphi\}$ RB(x, j) $\{\varphi\}$) .

Proof. *Suppose $rb1$ holds before action* RB*(x, j). Recall that* SwapToBlue*(x, i, y, j)* = SwapXY*(x, y);* RB*(x, j). Show that $rb1$ holds after the action* SwapXY*(x, y). By lemma 2, φ holds after the swap, therefore φ holds before action* RB*(x, j). There is a need to show that if φ holds before the action then φ holds after the action. By uniqueness, axioms of* Blue*(x, i), and axioms of change for* RB*(x, j), then after the action* RB*(x, j), the colour of element x turns into blue but the value and location persist, as well as property of the other elements. Furthermore, since the location and colour of other elements persist, the order of blue elements is not affected after action* RB*(x, j). Therefore, φ holds after the action. The proof for action* RB*(x, j) with $rb6$ or $rb7$ conditions are similar.* ■

Lemma 4 ($\{rb2 \wedge \varphi\}$ RB(x, i) $\{\varphi\}$) .

Proof. *Suppose $rb2$ ((x is red, x is the last element and is in ordered position)) holds before action* RB*(x, i). There is a need to show that if φ holds before the action then φ holds after the action. By uniqueness, axioms of* Blue*(x, i), and axioms of change for* RB*(x, i), then after the action* RB*(x, i), the colour of element x turns into blue but the value and location persist, as well as property of the other elements. Furthermore, since the location and colour of other elements persist, the order of blue elements is not affected after action* RB*(x, i). Therefore, φ holds after the action. The proof for action* RB*(x, i) with $rb4$ condition is similar.* ■

Lemma 5 ($\{rb3 \wedge \varphi\}$ RB(x, j) $\{\varphi\}$) .

Proof. *Suppose $rb3$ holds before action* RB*(x, i). Recall that* SwapToBlue*(x, i, y, j)* = SwapXY*(x, y);* RB*(x, j). Show that $rb3$ should hold after the action* SwapXY*(x, y). By lemma 2, φ holds after the swap, therefore φ holds before action* RB*(x,j). Need to show that if φ holds before the action then φ holds after the action. By uniqueness, axioms of* Blue*(x, i), and axioms of change for* RB*(x, j), then after the action* RB*(x, j), colour of element x turn into blue but the value and location persist, as well as property of the other elements. Furthermore, since the location and colour of other elements persist, the order of blue elements is not affected after action* RB*(x, j). Therefore, φ holds after the action.* ■

Lemma 6 ($\{rb5 \wedge \varphi\}$ RB(x, i) $\{\varphi\}$) .

Proof. *Suppose $rb5$ holds before action* RB*(x, i). There is a need to show that if φ holds before the action then φ holds after the action. By uniqueness, axioms of* Blue*(x, i), and axioms of change for* RB*(x, i), then after the action* RB*(x, i), the colour of element* *x turns into blue but the value and location persist, as well as property of the other elements. Furthermore, since the location and colour of other elements persist, the order of blue elements is not affected after action* RB*(x, i). Therefore, φ holds after the action.* ■

Lemma 7 ($\{rb8 \wedge \varphi\}$ RB(x, i) $\{\varphi\}$) ***Proof.*** *Suppose $rb8$ (all elements are red) holds before action* RB*(x, i). By definition of B(x), φ holds as well before the action. By uniqueness and by axioms of* Blue*(x, i) and axioms of change for* RB*(x, i), then after the action* RB*(x, i), the colour of element x turns into blue but the value and location persist, as well as property of the other elements. Therefore, φ holds after the action.*
■

Theorem 1 (For any formula φ) ,

1. *φ holds in initial state*
2. *for all atomic actions A,* $\{\varphi\}\ A\ \{\varphi\}$

Then $A_1;...;A_n \quad \{\varphi\}$ *for all n.*
Proof. *By induction on n*
$n = 0$, *φ holds in initial state.*
For n+1, there is $(A_1;...;A_n) \quad \{\varphi\}$ *and* $\{\varphi\}\ A_{n+1}\ \{\varphi\}$. *By composition rule, then*

$$(A_1;...;A_n);\ A_{n+1}\ \{\varphi\}$$

■

Theorem 2 (Safety: Blue elements always in ascending order) . *For* $\varphi \equiv$

$$\forall x, y \in S. \forall i, j \in I. ((x \neq y) \wedge x@i \wedge y@j \wedge B(x) \wedge B(y) \wedge (i \leq j) \Rightarrow (V(x) \leq V(y))$$

1. *φ holds in initial state*
2. *for all atomic actions A,* $\{\varphi\}\ A\ \{\varphi\}$

Then $A_1;...;A_n \quad \{\varphi\}$ *for all n.*
Proof. *φ holds for initial state and lemma 1 until 9 show that for all atomic actions A,* $\{\varphi\}\ A\ \{\varphi\}$ *holds.* ■

3.5 Liveness

Liveness deals with tracing the state of the elements every time. The following two things should be regarded in proof of OTF sorting liveness property: *different kinds of OTF sorting algorithms* and *fairness*. There are three types of OTF sorting algorithms. First, systematic OTF sorting. Second, the random OTF sorting, and thirdly, search-the-first-blue (right/left) OTF sorting.

Various kinds of liveness based on its fairness strength are: (a) liveness with weak fairness; (b) liveness with stronger fairness; and (c) liveness with even stronger fairness. For liveness with weak fairness, if the *feeder* is disabled then each element eventually becomes blue (*runs to termination*). This is similar to saying that if *feeder* is disabled then for each *x*, P(*x remains red*) = 0. In liveness with stronger fairness, for each element *x*, if *x* is not overwritten at some point in the trace then *x* eventually becomes blue (requires fairness). This is similar to saying that "An element *x* is allowed to become blue (*enabled*)." For liveness with even stronger fairness, if an element can be coloured blue infinitely often, then it will be coloured blue.

Possible proof results are: liveness (a) holds but not for random OTF sorting algorithm; liveness (b) not clear, if random OTF sorting algorithm then liveness (b) does not hold but may do so probabilistically. Keys to any proof of liveness property are selecting the relevant (fairness) form of traces (an "infinite" list of actions) and reasoning over any traces. This may well require some "output" probabilities, e.g. the algorithm operates so that P(*x becoming blue*) $>$ 0.5 for all *x*.

This section discusses the proof of *liveness* property regarding the fairness requirement. The liveness property (CC2) is formulated as follows:

$$\mathsf{Red}(e) \;\Rightarrow\; (\Diamond\ (\mathsf{Blue}(e)\ \vee\ e\ \text{overwritten by the feeder}))$$

which means that "Each element eventually becomes blue or the *feeder* overwrites it."

Notice the part of liveness: "or the *feeder* overwrites it." This part is related to the existence of an array element in OTF sorting. This example uses the constant domain model, in which the domain of each possible world is the same as every other. Here, there is an element within the array (called *InArray(e)*), as well as elements outside the array (called *OutArray(e)*). Furthermore, if the *feeder* overwrites the value of an element, this means that the *feeder* **swaps** the value of element between *InArray(e)* and *OutArray(e)*. Although values of an element *e* may be duplicated, the existence of each element in the domain differentiates by its "id", such as e_1, e_2 and so on. Therefore the liveness can be re-defined as follows:

$$\mathsf{Red}(e) \ \Rightarrow \ (\Diamond\ \mathsf{Blue}(e)\ \vee\ \mathit{OutArray}(e))$$

with

$$\mathsf{Red}(e) \ \Rightarrow \ \mathit{InArray}(e)$$

3.5.1 Definition

Axioms 5 ***(Diamond).***

$$L:\ \Diamond\ \phi\ \equiv\ \exists L'\ \geq\ L.\ L':\phi$$

Axioms 6 ***(Box).***

$$L:\ \Box\ \phi\ \equiv\ \forall L'\ \geq\ L.\ L':\phi$$

Axioms 7 ***(Fair for element e).*** *The sorter is* ***Fair for element e*** *if in any trace in which e is re-assigned only finitely often, the following hold :*

$$F1:\ \ e\ \ \textit{is pre-}\mathsf{RB}(e)\ \ \textit{enabled infinitely often}\ \ (\textit{Re-arrangement})$$
$$F2:\ \ \textit{if pre-}\mathsf{RB}(e)\ \ \textit{is enabled infinitely often then}\ \ \Diamond\ \mathsf{Blue}(e)\ \ (\textit{Colouring})$$

The *re-assignment* of *e* means that value of element *e* is overwritten by the feeder.

3.5.2 Proposition

For any element *e*, if the sorter is fair for *e* then

$$P1:\ \mathsf{Red}(e)\ \Rightarrow\ \Diamond\ \mathsf{Blue}(e)$$
$$\textbf{hence}\ \ P2:\ \Diamond\ \Box\ \mathsf{Blue}(e)$$

3.5.3 Proof

From the algorithm, the initial condition that the colour for all elements in the array is Blue. Figure 3.15 shows that after the initial conditions, the state of each element can

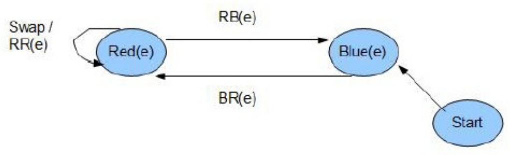


Figure 3.15: OTF sorting automaton of an element *e*, with actions from feeder and sorter.

change from Blue to Red by the *feeder* process through an action BR(*e*). Afterwards, an element *e* can remain Red for some period of time through the Swap action or the RR(*e*) action of the *feeder* again, and so the *sorter* process through RB(*e*) action the element *e* will be Blue again. Alternatively, the colour of an element *e* can directly change to Blue again, and therefore at least element *e* changes to Red only once.

We show for traces which are fair for element *e* that P1 and P2 hold. By fairness F1 and F2, and by modus ponens, then P1. F1 and F2 are sufficient with *e* not re-assigned and they are necessary (weaken to *e* re-assigned finitely often). If *e* is blue then $\Box$ Blue(e) because only one action can change colour of *e*, BR(e), but this is excluded from the traces. If *e* is red then it may have action Swap being applied which may lead to pre-RB(*e*) being enabled.

We can show that:

$$\frac{\Diamond\ \mathsf{Blue}(e)}{\Diamond\ \Box\ \mathsf{Blue}(e)}$$

through traces, for some N

$$(\mathsf{BR}(e)(\mathit{Swap}|\mathsf{RR}(e)) * \mathsf{RB}(e))^N$$

means that initially, a Blue element *e* will be Red through BR(*e*). Then the action Swap or RR(*e*) may be applied to the element *e* for some times and eventually action RB(*e*) takes place. These sequences of actions can occur N times and element *e* is always Blue in the end.

3.6 Discussion

The frame problems are expressed using the frame axioms. There are frame axioms for the current element which is being sorted and frame axioms for other elements, which hold before and after the execution of the action. These frame axioms are needed to express what has been changed due to the action. Furthermore, the frame axioms show what remain the same after the action. The OTF sorting example requires such frame axioms because to obtain the appropriate position for an element the system has to ensure that the other elements remain in the same position (except the element which is involved in swap action).

Here the example only uses the indices to refer to the overwriting and do the correctness prove CC1 (safety), but it may be hard to prove the liveness property because then we have to use the counterpart function to denote the identity of the elements every time (showing the traces). Moreover, this example uses constant domain models. Therefore, to overcome the existence of element that has been overwritten by the *feeder*, the OTF example uses the definitions of elements in the array (InArray(e)) and elements outside the array (OutArray(e)).

To reason about the OTF sorting, we use colouring as external knowledge. This colouring, which is a common practice in mathematics, is inspired by Dijkstra et al. [12] and in Gries [21] work OTF Garbage Collection. Blue denotes an element is in an appropriate position, while red denotes that the position of an element in the array can still change to an appropriate position. The following chapter, on dynamic network modelling, discusses colouring to illustrate message-passing.

This introductory example illustrates reliability and demonic process. The changes of value of elements in the array by the *feeder* when the *sorter* attempts to order the elements reveal reliability. Here, the *feeder*'s role is as the demonic process which keeps changing the correct order of elements in the array. Furthermore, this example shows that we do not always need termination in dynamic systems. Something good eventually happens in the system (in this example, liveness property has been proved) without the need for the system to terminate.

OTF sorting example uses invariant to proof safety properties. The invariant ϕ expresses the condition in which *blue elements are always in order*. When proving dynamic network model in Chapter 5, the proofs use concurrent systems proof techniques, trace properties and LTL to establish the correctness of message-passing in dynamic networks.

Chapter 4

Dynamic network models as concurrent systems

After the discussion in Chapter 3 about how to model dynamic systems, this chapter discusses modelling dynamic networks. We begin with a discussion about how to describe a dynamic network model and what features should be considered. This is followed by a brief explanation of hierarchical approaches to model and various possible dynamic network models as concurrent systems. In the main discussion, a formal model of dynamic networks is presented through its definitions and axioms. Following this, the two models, *Model A* and *Model B* are introduced and are then explored further in the next chapter (Chapter 5).

4.1 Description

Why model dynamic networks? Dynamic networks are widespread in communications technology and include the Internet, peer-to-peer networks, mobile networks and wireless networks. The networks may have the edges down, the nodes may move, or there may be routing instability due to the changing of connections. These systems are very difficult to analyse, and their behaviour and correctness are hard to formulate and establish. To undertake formal reasoning about such systems, abstract models are essential to separate general reasoning about message-passing and updating of routing tables from the details of how these are implemented in the particular networks.

Dynamic networks have been modelled using various approaches. Some of these models use high-level model approaches, as been mentioned in Section 2.6. The high-level models do not assume that the network stops changing at some point, do not restrict the number or frequency of changes, and maintain long-term stability [34, 48, 11]. Algorithms of self-stabilizing systems provide correctness [10] in terms of safety (which holds even while the network is dynamic) and liveness (which only holds when the network stops changing, and may never converge). Here, we consider a novel approach to modelling, using a number of processes running concurrently.

The dynamic network is modelled as a demonic process which runs concurrently with routing updates and message-passing. In this model approach, there are at least two types of processes. One process has a role to disrupt the connectivity of dynamic networks. Another processes attempt to run the normal execution, such as: deliver the messages and update the routing table. All of these processes keep running concurrently and do not need to terminate for correctness.

Before discussing further how to model dynamic networks as concurrent systems, the following ideas need to be considered (see also Section 2.6 regarding a brief survey of dynamic network models).

1. *Mode of communication.* Communication can be synchronous or asynchronous; message-passing along “channels” or via shared memory; and broadcast vs. point-to-point. These modes will distinguish between resource sharing and information spreading in a dynamic network model.

2. The *mechanisms for spreading information* within the dynamic networks. We know that there are some mechanisms for spreading information within the network: point-to-point, broadcast, and flooding. If edge status changes, the incident nodes inform this to other nodes. The question is: how quickly? If it is very fast then we need a process which keep nodes within the network informed about the actual edge status. This is difficult using a route info message because such a message combines point-to-point with broadcast; and the dynamic means that consensus (i.e. on what is the best route) is never be reached.

3. The distinction between a *true path*, a *broken path* (non-true path) and a *live path*. In terms of the routing information, a *true path* is a path which is available and a *broken path* is an unavailable path. Meanwhile, a *live path* is an available path within the network.

4. The *routing mechanism*, which can be instantaneous and global, or propagated (this depends on the dynamics of the networks). When we instantaneously update the routing table, the convergence time is ignored, but we can consider

route validity and message sending at some point. This applies in all three of the following cases: at intermediate nodes, with a path to the destination; at intermediate nodes, without a path to the destination; or at the destination already.

5. *Process relations*. There are three kinds of process relations of concurrent systems, as follows.

   (a) consensus: all processes identical

   (b) similar processes sharing resources

   (c) on-the-fly: completely different processes

The dynamic network models as concurrent systems are developed using asynchronous communication, shared memory and do not distinguish the information spreading mechanism (see Section 4.2 for further discussion). The models recognise true path, broken path and available path, and also distinguish the routing mechanism into a different type of models. The process relations of dynamic network models are on-the-fly.

4.2 A hierarchy of models

Hierarchical approaches start by developing an abstraction of a dynamic network model. The abstraction level of the models has the following two key features:

1. The model abstractions do not distinguish local and global routing;

2. The model abstractions do not specify the mechanisms for distributing/spreading the information status of links within the network.

The reason for this is because we want to describe a high-level abstraction of a model that can be refined if we want to add further details. Distinguishing local and global routing occurs when we detail the hierarchy and topology of networks. We need to specify the mechanism for spreading information within the networks when we use particular routing algorithms. In fact, all of these details start from the high-level abstraction model.

The hierarchical model approach utilises a separation concern in modelling a dynamic network. Three properties of dynamic networks are formulated as the result of the separation concern. These are: *graph properties*, *message properties*, and *routing properties*. Graph properties presents the assumption of graph/network connectivity.

This is related to how we describe the path existence within the network. Here the paths exist infinitely often.

Message properties indicate the behaviour of messages during the delivery from source to destination. These properties are concerned with message changes of status, e.g. the type of actions that make a message move along a path. They also indicate how often a message is looked at. We describe the properties of messages that are needed: message eventually green and message eventually move.

Routing properties describe how the routing table is populated/correctness of the routing table. Changes on the connectivity of networks that occurs frequently over time affecting the content of the routing table. To remain correct, the routing table must be updated according to the changes. Here we need to show that the routing table is correct sufficiently often.

How do we define a "well-behaved" dynamic network using such properties ? A "well-behaved" dynamic network are described through the three properties above using fairness. The formulation depend on the type of the model that we want to describe (which is explained later in Section 4.5 and 4.6). In dynamic networks as concurrent systems, these fairness are formulated as expressions of inter-process probabilities as well as intra-process probabilities. Therefore, through different type of fairness, we can build various description using the three properties (graph, message and routing properties).

This level of abstraction can be extended further to a more specific model. A dynamic network with routing table that instantaneously update needs the relation definition between graph properties and message properties. This relation definition between graph properties and message properties will slightly differ from those in a model with updated routing table, which also involves routing properties. For example, if we want to describe the routing mechanism for information spreading within the network in more detail (e.g. mobile ad hoc network with proactive routing algorithms), then we can use the three properties detailed above. The details of the mechanism are described through the relations between the three properties. Fairness conditions for the models can be derived from those general properties as well by changing the type of relation. In summary, for more realistic network and more levels of abstraction, we can use the same techniques.

The dynamic network model is built using the hierarchical approach, so it has levels of correctness according to this type of abstraction. These levels of correctness are discussed in Chapter 5. Furthermore, the abstraction of dynamic network models should also covers timing and probabilistic features. This will be the focus of future work.

4.3 Various possible dynamic network models as concurrent systems

This section discusses about developing dynamic network models in concurrent systems and consider various possible models. These various models are built based on the combination of routing information as well as the concurrent processes involved in the systems. There is also a discussion about how these models cover the existing routing algorithms. Finally, two specific models of these various models to be explored further in Section 4.5 and 4.6.

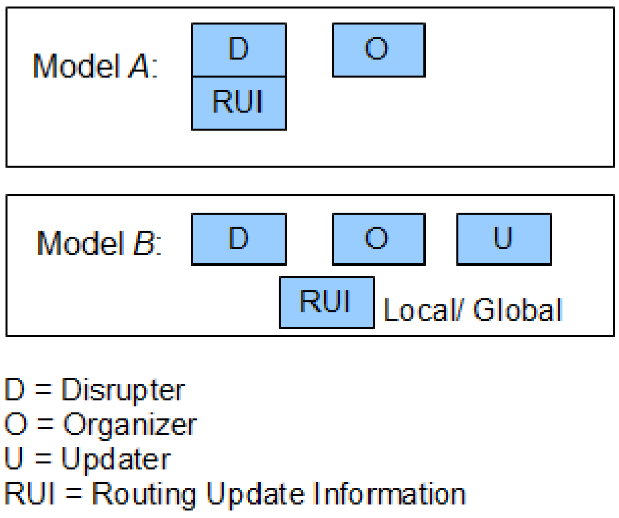


Figure 4.1: Various Dynamic Network Models.

A dynamic network model may have a global routing table or local routing tables; and possibly has synchronous/asynchronous communication between its processes or no communication at all except via shared memory/variables. Therefore, there are two possible dynamic network models here:

- *Model A*, using instantaneously routing update. Here, routing tables always correct: i.e., they contain the actual connectivity of the network at that instant. As shown in Figure 4.1, *Model A* has two process, *D* (*Disrupter*) and *O* (*Organizer*), in which there is also embedded routing update information on *D*. This routing update information will be adjusted when there is a change of edge status.

- *Model B*, in which the routing table could be wrong (using local routing table makes more sense, but it could be a global routing table). We are concerned with the correctness of the routing table but not with the mechanism. As shown in Figure 4.1, *Model B* has three process: *D* (*Disrupter*), *O* (*Organizer*) and *U* (*Updater*). The routing update information may be provided locally or globally within the network. The *Model B* might have distributed routing tables and as well as a distributed *Organizer* and a distributed *Updater*, with routing update information being provided locally (on each node).

Dynamic network models also can be categorised based on which process is the main orientation of the systems. This categories are *Disrupter-oriented*, *Organizer-oriented*, *Updater-oriented*, and combination of these three. *Disrupter-oriented* means that every time a disruption occurs, the *Updater* process will recalculate the routing information. This type of model may involve inter-process communication. An example of a routing algorithm which is covered by this model is the proactive routing algorithms.

Organizer-oriented means that if there is a message that needs to be forwarded then the *Updater* will try to find the available route for it. We can use two shared variables in this type of model. One variable retains the source and destination information. This variable is written by the *Organizer* and read by the *Updater*. The second variable is used for preserving the available route at the moment, which is written by the *Updater* and read by the *Organizer*. An example of routing algorithm which is covered by this model is reactive routing (on-demand routing).

Updater-oriented means that when the *Updater* finds an available route, this information will be passed to the *Organizer*. The *Organizer* then can forward messages using these available routes. With this type of model, we can increase the possibility of a message being delivered to its destination. A combination of these three: *Disrupter-*, *Organizer-*, and *Updater*-oriented is similar to the supervised evolution model.

One of the objectives in building a dynamic network model is to establish the correct behaviours of the systems, particularly those related to routing algorithms. Alotaibi and Mukherjee [1] survey existing routing algorithms for wireless ad hoc and mesh networks, and discuss the routing algorithms associated with the advantages and disadvantages of each category. Thus the various dynamic network models being explored on how they cover these types of routing algorithms. A dynamic network model as concurrent systems might be in one of the following characteristics:

- has a global routing update information; global *Updater* || *Organizer*; and routing update information is stored in a routing table or involve a discovery of routing update information at a time. The mechanism to obtain routing update information is an I tell you, you tell me mechanism. This type of models has an association with wireless mesh network routing algorithms [1].

- has a global routing update information; local *Updater* || *Organizer*; and routing update information involves a discovery of routing update information at a time and path information of a node id in network topology. The mechanism for getting routing update information is "Ill ask you when I need it". This type of models is *Organizer-oriented*, where if there is a message that needs to be forwarded then the *Updater* will try to find the available route for it. Furthermore, this model uses shared variables. Therefore, we do not need to use communication between processes in this model. Routing update information is recalculated by the *Updater* based on the message and the *Organizer*'s need. This type of models has an association with reactive routing algorithms and geographical routing algorithms with flooding technique [1].

- has local routing update informations; global *Updater* || *Organizer*; and routing update information is stored in routing table or involve discovery of routing update information at a time. The mechanism to obtain routing update information is an "I tell you, you tell me" mechanism. This type of models has an association with cluster-based routing algorithms and wireless mesh network routing algorithms [1].

- has more than two processes: *Disrupter*, *Organizer*, and *Updater*; focus on local routing update information local *Organizer* || *Updater*; has local routing update information with local *Updater* || *Organizer*. The type of routing update information in this model might be stored in a routing table or a discovery of routing update information at a time. The mechanism for getting routing update information is an "I tell you, you tell me" mechanism. This type of models has an association with proactive routing algorithms and geographic routing algorithms with single path or multi-path techniques [1].

4.4 Formal dynamic network models

Now we describe the dynamic network model in a formal form. This formal form enable us to define properties for the dynamic network models that discussed in Section 4.5 and 4.6. Consider a graph $\mathsf{G} = (\mathsf{N}, \mathsf{E})$ where N is a set of nodes and E is a set of edges. This graph provides the basic connectivity of a network. To introduce dynamics (for edges) we consider the edges to be in an *on* or *off* state. Therefore we introduce edge status L, which changes value when a disruption occurs for an edge. Routing update information is determined by L. There is a set of messages M, where each message is at a node defining a function $\mathsf{M} \to \mathsf{N}$.

A network therefore consists of (1) a graph G; (2) a collection of messages at nodes $\mathsf{M} \subseteq \{(\mathsf{s}, \mathsf{d}, contents, \mathsf{n}) : \mathsf{s}, \mathsf{d}, \mathsf{n} \in \mathsf{N}\}$ where n, s and d denote node, source, and destination respectively; (3) a routing table RT; with the following structure:

$$NetworkModel\ \ \mathsf{G} = \langle \mathsf{N}, \mathsf{E} \rangle$$
$$Nodes\ \ \mathsf{N} \subseteq \{\mathsf{NodeId} : \mathsf{NodeId} \in \mathcal{Z}\}$$
$$Edges\ \ \mathsf{E} \subseteq \{(u, v) : u, v \in \mathsf{N}\ \times\ \mathsf{N}\}$$
$$Message\ \ \mathsf{M} \subseteq \{(\mathsf{s}, \mathsf{d}, contents, \mathsf{n}) : \mathsf{s}, \mathsf{d}, \mathsf{n} \in \mathsf{N}\}$$
$$\mathsf{OnOffState}\ =\ \{\mathsf{On}, \mathsf{Off}\}$$
$$Path\ \ \mathsf{P}\ =\ (\mathsf{N}\ \times\ \mathsf{N})^*$$
$$Route\ \ \mathsf{R}\ \subseteq \{\mathsf{d}\ \times\ \mathsf{P} : \mathsf{d} \in \mathsf{N}\}$$
$$RoutingTable\ \ \mathsf{RT} = \mathsf{N}\ \rightharpoonup\ \mathsf{R}$$
$$LinkStatus\ \ \mathsf{L} = (\mathsf{E} \to \mathsf{OnOffState})$$

The state of the network may change by an action:

$$\langle \mathsf{g}, \mathsf{m}, \mathsf{L}, \mathsf{rt} \rangle \overset{action}{\to} \langle \mathsf{g}', \mathsf{m}', \mathsf{L}', \mathsf{rt}' \rangle$$

where $\mathsf{g} \in \mathsf{G}$, $\mathsf{m} \in \mathsf{M}$, $\mathsf{L}\ \in \mathsf{L}$ and $\mathsf{rt}\ \in \mathsf{RT}$. The *actions* are those of the processes which model the network.

Each message is coloured according to its status (following techniques from Gries [21] and Dijkstra [12]'s reasoning about on-the-fly garbage collection). Messages move within the simple network as in Figure 4.2 according to the available path p. Here we observe the progress of a message m which is delivered from source s to a destination d through the given route from the router. Notification of progress can be

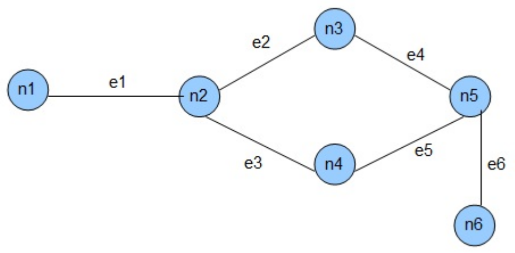


Figure 4.2: An example of simple network with nodes n_i and edges e_i.

done by embedding. Notification of progress in dynamic networks can be defined as follows:

- black : at destination
- green : progressing along the original route (continue with the same route)
- red : the route is being blocked by edge disruption (the current route is blocked)

A message m will be colored **red** by the *Organizer*:

- when m is picked by the *Organizer* and the link to the next node along the route that leads to the destination is not connected, or
- when m is picked by the *Organizer* and there is no route leading to the destination

A message m will **remain red** if there is still no path for progressing the message through to the destination. A message m will be colored green by the *Organizer*:

- when m is picked by the *Organizer* and the link to next node along the route leads to the destination is connected; or
- when m is picked by the *Organizer* and the link to the next node along the route that leads to the destination is not connected, but there is an available alternate route which leads to the destination

A message m will **remain green** if m moves along the path. This coloring needs the information that the link to the next node will lead the message m to the destination (this is the requirement for routing calculation).

In the "on-the-fly" sorting example, the *sorter* chooses *an* element *e* (so not for *all* elements *e*) such that "if BR(e) action only finitely often then $\Diamond\Box$ Blue(e)". There is an interaction between the elements in the array, but in dynamic networks, the interaction is between message m and the routing information. The colouring is not involved with the message-moving decision process. This colouring of the message notification is only being used as an indicator to understand the behaviour of message passing in dynamic networks.

Therefore we also now have a coloured model for those network models which may be changed by an action as follows.

$$\langle \mathsf{g}, \mathsf{m}, \mathsf{L}, \mathsf{rt}, \mathsf{m}_C \rangle \overset{action}{\rightarrow} \langle \mathsf{g}', \mathsf{m}', \mathsf{L}', \mathsf{rt}', \mathsf{m}'_C \rangle$$

where m_C is a coloured-notification for message m. These colours are being used to enable us to reason about the model and do not exist in the real network model.

We define the following predicates on state:

- *disconnected(*e*)*

$$disconnected(\mathsf{e})(\langle \mathsf{g}, \mathsf{m}, \mathsf{L}, \mathsf{rt}, \mathsf{m}_C \rangle) \;= \mathsf{L}(\mathsf{e}) \;=\; \mathsf{Off}$$

- *connected(*e*)*

$$connected(\mathsf{e})(\langle \mathsf{g}, \mathsf{m}, \mathsf{L}, \mathsf{rt}, \mathsf{m}_C \rangle) \;= \mathsf{L}(\mathsf{e}) \;=\; \mathsf{On}$$

- *valid(*p*)*

$$valid(\mathsf{p})(\langle \mathsf{g}, \mathsf{m}, \mathsf{L}, \mathsf{rt}, \mathsf{m}_C \rangle) \;= \forall \mathsf{e} \in \mathsf{p}.\; connected(\mathsf{e})$$

  valid(p) is the property of a system which determines externally the validity of a path p.

- *avail(*p*)*

$$\begin{gathered} avail(\mathsf{p})(\langle \mathsf{g}, \mathsf{m}, \mathsf{L}, \mathsf{rt}, \mathsf{m}_C \rangle) \;= \\ \mathsf{rt}(\mathsf{n}, \mathsf{d}, \mathsf{p}) \;\wedge\; \mathsf{n} \in \mathsf{N} \;\wedge\; dest(\mathsf{p}) = \mathsf{d} \;\wedge\; \forall \mathsf{e} \in \mathsf{p}.\; connected(\mathsf{e}) \end{gathered}$$

- *path(*m, p*)*

$$\begin{gathered} path(\mathsf{m}, \mathsf{p})(\langle \mathsf{g}, \mathsf{m}, \mathsf{L}, \mathsf{rt}, \mathsf{m}_C \rangle) \;\Rightarrow \\ avail(\mathsf{p}) \;\wedge\; dest(\mathsf{m}) = dest(\mathsf{p}) \end{gathered}$$

path(m, p) means that a path pat this instance from the current position of m to the destination.

- *is_avail(*m, p*)*

$$\begin{aligned} is\_avail(\mathsf{m},\mathsf{p})(\langle \mathsf{g},\mathsf{m},\mathsf{L},\mathsf{rt},\mathsf{m}_C\rangle) \;\Rightarrow \\ path(\mathsf{m},\mathsf{p}) \;\wedge\; valid(\mathsf{p}) \end{aligned}$$

- axioms of colouring

$$\begin{aligned} red(\mathsf{m}) &\equiv (\neg green(\mathsf{m}) \wedge \neg black(\mathsf{m})) \\ green(\mathsf{m}) &\equiv (\neg red(\mathsf{m}) \wedge \neg black(\mathsf{m})) \\ black(\mathsf{m}) &\equiv (\neg red(\mathsf{m}) \wedge \neg green(\mathsf{m})) \end{aligned}$$

4.4.1 Definitions

A trace is a finite or infinite sequence

$$\tau = S_1 \overset{a_1}{\rightarrow}_{P_1} S_2 \overset{a_2}{\rightarrow}_{P_2} S_3 \overset{a_2}{\rightarrow}_{P_3} \cdots$$

where $P_i \in \{Disrupter, Updater, Organizer\}$,
a_i is an action of P_i
We define process projections on traces as follows

$$\begin{aligned} \mathsf{S} = \langle \mathsf{N}, \mathsf{D}_S, \mathsf{O}_S, \mathsf{U}_S \rangle \\ \Pi_{Disrupter}(\langle \mathsf{N}, \mathsf{D}_S, \mathsf{O}_S, \mathsf{U}_S \rangle) = \langle \mathsf{N}, \mathsf{D}_S \rangle \end{aligned}$$

Projection of *Organizer* process trace, $\Pi_{Organizer}(\mathsf{S})$, is that part of S which the *Organizer* has access to.

$$\begin{aligned} \mathsf{S} = \langle \mathsf{N}, \mathsf{D}_S, \mathsf{O}_S, \mathsf{U}_S \rangle \\ \Pi_{Organizer}(\langle \mathsf{N}, \mathsf{D}_S, \mathsf{O}_S, \mathsf{U}_S \rangle) = \langle \mathsf{N}, \mathsf{O}_S \rangle \end{aligned}$$

Projection of *Updater* process trace, $\Pi_{Updater}(\mathsf{S})$, is that part of S which the *Updater* has access to.

$$\mathsf{S} = \langle \mathsf{N}, \mathsf{D}_S, \mathsf{O}_S, \mathsf{U}_S \rangle$$
$$\Pi_{Updater}(\langle \mathsf{N}, \mathsf{D}_S, \mathsf{O}_S, \mathsf{U}_S \rangle) = \langle \mathsf{N}, \mathsf{U}_S \rangle$$

For each P in $\{Disrupter, Updater, Organizer\}$

$$\Pi_P((S_1 \xrightarrow{a1}_{P1} S_2).\tau) =$$
$$\Pi_P(\tau)\ \ if\ \ P1 \neq P,$$
$$(\Pi_P(S_1) \xrightarrow{a1} \Pi_P(S_2)).\Pi_P(\tau)\ \ otherwise$$

τ is a valid trace and for each process P_i, $\Pi_{Pi}(\tau)$ is a P_i trace.

Therefore, for example, we can define for $j < |\tau|$:

$$\Pi_{Disrupter}(\tau), j \models \{disconnected(\mathsf{e})\}\ \ \mathsf{DTC}(\mathsf{e})\ \ \{connected(\mathsf{e})\}$$

4.4.2 Actions

Figure 4.3 shows the *Disrupter*||*Updater*||*Organizer* automata with shared variables. D, U, O denotes the *Disrupter*, *Updater*, and *Organizer* processes respectively. L is a shared variable of link status, R is a shared variable of routing tables and M is a variable of the message list. 1 denotes the input for the *Disrupter* process, in which the output (denoted by 2) as the result of action in *Disrupter* process, will be the input for the *Updater* process. The *Disrupter* process accesses variable L when performing its actions. The *Updater* process accesses the shared variable L and R for performing its actions. The output of the *Updater* process (denoted by 3) will be the input for the *Organizer* process. The *Organizer* process will access the shared variable R and the variable M for performing its actions.

Actions of *Disrupter* process are:

- DTC(e), disconnected to connected e.

Action 1 *Action* DTC(e) *:*

$$\{disconnected(\mathsf{e})\}\ \ \mathsf{DTC}(\mathsf{e})\ \ \{connected(\mathsf{e})\ \}$$
$$\{connected(\mathsf{e})\}\ \ \mathsf{DTC}(\mathsf{e})\ \ \{connected(\mathsf{e})\}$$

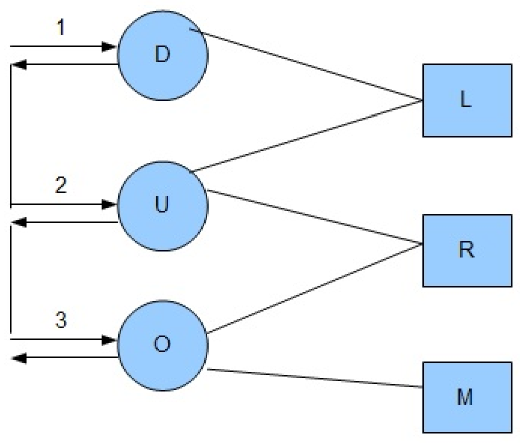


Figure 4.3: *Disrupter*||*Updater*||*Organizer* automata with shared variables.

- CTD(e), connected to disconnected e.

  Action 2 *Action* CTD(e) *:*

$$\{connected(\mathsf{e})\}\ \mathsf{CTD(e)}\ \{disconnected(\mathsf{e})\}$$
$$\{disconnected(\mathsf{e})\}\ \mathsf{CTD(e)}\ \{disconnected(\mathsf{e})\}$$

The *Updater* process are as follows:

- UNAv(p) : marks a particular path p as unavailable.

  Action 3 *Action* UNAv(p) *:*

$$\{avail(\mathsf{p})\}\ \mathsf{UNAv(p)}\ \{\neg avail(\mathsf{p})\}$$

- UAv(p) : marks a particular path p as available.

  Action 4 *Action* UAv(p) *:*

$$\{\neg avail(\mathsf{p})\}\ \mathsf{UAv(p)}\ \{avail(\mathsf{p})\}$$

Figure 4.4 shows the automata for the *Disrupter* and *Updater* processes. The actions of the *Disrupter* process, DTC(e) and CTD(e), alternately turn the state e_state to 'disconnected' or 'connected'. The probability of each action to get their turn can be adjusted to represent different types of dynamic networks. The similar adjustment can be applied to the actions of the *Updater* process, $\mathsf{UAv(p)}$ and $\mathsf{UNAv(p)}$, which alternately turn the state is_avail(p) to ‘avail’ and ‘¬ avail’.

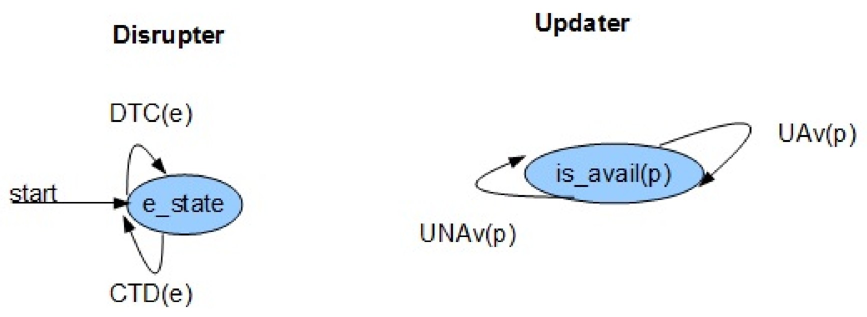


Figure 4.4: *Disrupter* and *Updater* automata.

Figure 4.5 shows the automata for *Organizer* process. Some actions that belong to the *Organizer* process are:

- RG(m) is the action in which a red message turns into green when there is a given route or an alternate route available.

  Action 5 *Action* RG(m) *:*

$$\{red(\mathsf{m}) \wedge valid(\mathsf{p})\}\ \mathsf{RG(m)}\ \{green(\mathsf{m}) \wedge valid(\mathsf{p})\}$$
$$\{green(\mathsf{m}) \wedge valid(\mathsf{p})\}\ \mathsf{RG(m)}\ \{green(\mathsf{m}) \wedge valid(\mathsf{p})\}$$

- GR(m) is the action in which a green message turns into red when the given route is blocked or no alternate route available.

  Action 6 *Action* GR(m) *:*

$$\{green(\mathsf{m}) \wedge \neg valid(\mathsf{p})\}\ \mathsf{GR(m)}\ \{red(\mathsf{m}) \wedge \neg valid(\mathsf{p})\}$$
$$\{red(\mathsf{m}) \wedge \neg valid(\mathsf{p})\}\ \mathsf{GR(m)}\ \{red(\mathsf{m}) \wedge \neg valid(\mathsf{p})\}$$

- GB(m) is the action in which a green message turns into black when message m reach the destination.

Action 7 *Action* GB(m) *:*

$$\{green(\mathsf{m}) \wedge valid(\mathsf{p}) \wedge (at(\mathsf{m}) = dest(\mathsf{m}))\}$$
$$\mathsf{GB(m)}$$
$$\{black(\mathsf{m}) \wedge valid(\mathsf{p}) \wedge (at(\mathsf{m}) = dest(\mathsf{m})\}$$
$$\{black(\mathsf{m}) \wedge valid(\mathsf{p}) \wedge (at(\mathsf{m}) = dest(\mathsf{m})\}$$
$$\mathsf{GB(m)}$$
$$\{black(\mathsf{m}) \wedge valid(\mathsf{p}) \wedge (at(\mathsf{m}) = dest(\mathsf{m})\}$$

- BB(m) is the action in which a black message stays black when message m has arrived at the destination.

Action 8 *Action* BB(m) *:*

$$\{black(\mathsf{m})\} \ \mathsf{BB(m)} \ \{black(\mathsf{m})\}$$

- move(m, p)

Action 9 *Action* move(m, *j*) *:*

$$\{at(\mathsf{m}) = i \wedge green(\mathsf{m}) \wedge valid(\mathsf{p}) \wedge path(\mathsf{m},\mathsf{p})\}$$
$$\mathsf{move}(\mathsf{m}, j)$$
$$\{at(\mathsf{m}) = j \wedge i \neq j \wedge green(\mathsf{m}) \wedge valid(\mathsf{p}) \wedge path(\mathsf{m},\mathsf{p})\}$$

- wait(m)

Action 10 *Action* wait(m) *:*

$$\{at(\mathsf{m}) = i \wedge red(\mathsf{m}) \wedge \neg valid(\mathsf{p}) \wedge \neg path(\mathsf{m},\mathsf{p})\}$$
$$\mathsf{wait(m)}$$
$$\{at(\mathsf{m}) = i \wedge red(\mathsf{m}) \wedge \neg valid(\mathsf{p}) \wedge \neg path(\mathsf{m},\mathsf{p})\}$$
$$\{at(\mathsf{m}) = i \wedge green(\mathsf{m}) \wedge \neg valid(\mathsf{p}) \wedge \neg path(\mathsf{m},\mathsf{p})\}$$
$$\mathsf{wait(m)}$$
$$\{at(\mathsf{m}) = i \wedge red(\mathsf{m}) \wedge \neg valid(\mathsf{p}) \wedge \neg path(\mathsf{m},\mathsf{p})\}$$

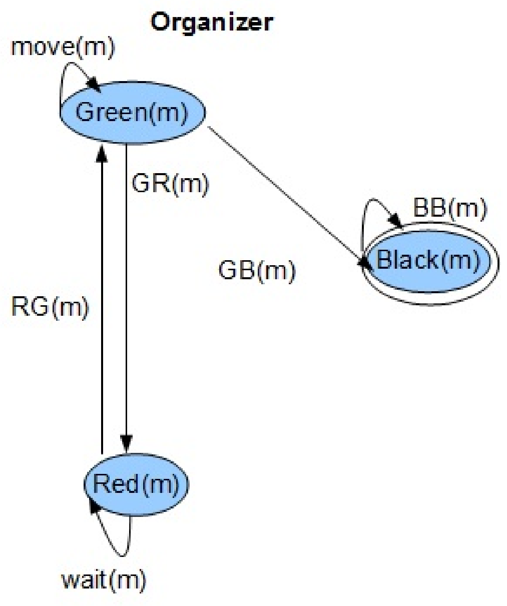


Figure 4.5: *Organizer* automata.

The aim of the dynamic network model is to establish the correctness of the network at the relevant level of abstraction. Correctness is

$$\forall \mathsf{m} \in M.\Diamond\Box black(\mathsf{m})$$

which means that every message m will eventually be delivered. One of the ideas to prove correctness is: using is_connect(p) to indicate whether path p is connected or not at some point; ensuring that the network is sufficiently connected sufficiently often; and ensuring that the routing table information is sufficiently correct sufficiently often. We also need to consider if there is any communication between processes and think about the arrival of new messages. Clearly in realistic networks eventual delivery is not good enough. The delivery should be "on time".

As discussed in Section 4.3, the dynamic network model is considered as concurrent systems with at least two kinds of processes. Each process may have its own structure. *Model A* and *Model B* are developed and shown in Figure 4.6 and 4.7, respectively, with a simple network as in Figure 4.2. Further explanation in the following sections.

4.5 *Model A*

Model A, as shown in Figure 4.6, is a two-process model with instantaneous updates, in which the routing tables are always correct. The two processes are a “demonic” *Disrupter* (which disrupts the connectivity of dynamic networks) and an *Organizer* (which attempts to deliver messages). The *Disrupter* process can disrupt the connection of an edge (so it becomes “on” or “off”).

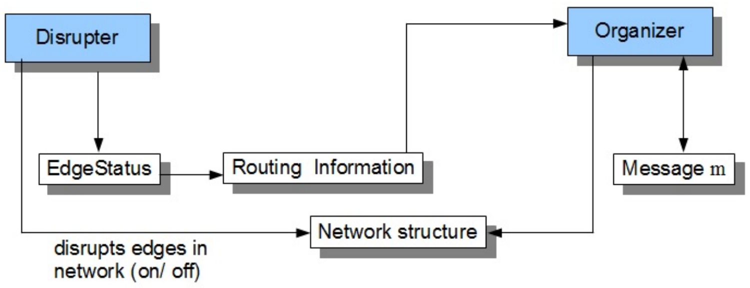


Figure 4.6: *Model A* (Two-process dynamic network model).

As shown in Figure 4.6, the *Disrupter* process can disrupt the connection of an edge (edge state becomes “on” or “off”) and the changes are written in a shared variable Link Status.

For *Model A*, the modal properties we need are:

1. Paths exist infinitely often:

   P1: $\forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\Diamond\ pathX(\mathsf{n}_1, \mathsf{n}_2)$

2. Red messages eventually become green (messages are looked at sufficiently often):

   P2: $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((\Box\Diamond\ pathX(at(\mathsf{m}), dest(\mathsf{m})))\ \wedge\ red(\mathsf{m})$
   $\Rightarrow\ \Diamond\ (green(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m}, \mathsf{p}))))$

P1 is a **graph connectivity** condition.

P2 is a simple form (which we will strengthen later) of a **fairness condition**; “if a path is available infinitely often then a path is allocated sometime in the future”. Notice that the interest here is in the formulation: it expresses properties of dynamic networks as trace properties, some in terms of fairness of process interaction other as connectivity properties of graphs. P2 is in a particularly simple form which we will need to extend: it **does not** consider message moves for example. We now consider the process update model.

4.6 *Model B*

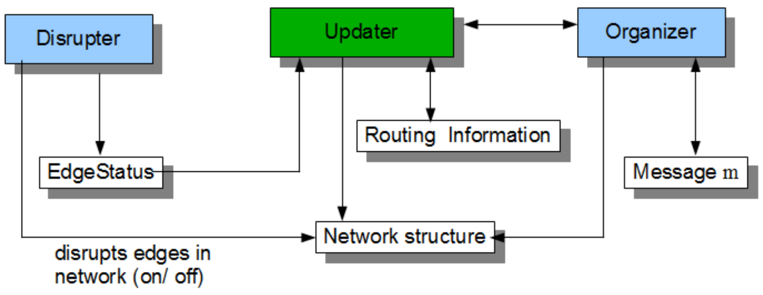


Figure 4.7: *Model B* (Three-process dynamic network model).

Model B presents a more realistic routing table update, adding a third process called an *Updater*, as shown in Figure 4.7. Here, the routing tables may not be correct at a particular time but routing is still possible. The *Updater* runs concurrently, recalculating the routing update information to obtain actual available paths. If there is an available path, the *Organizer* process runs concurrently and can send a message to the next node along a path.

The *Updater* process will read this Link Status to keep up-to-date with the state of every edge within the network. Changes in Link Status may influence the routing update information. The *Updater* process then recalculates the routing update information to obtain actual available paths. The *Organizer* process communicates with the *Updater* process about whether there are any paths available for a particular message m. If there is an available path, the *Organizer* process can send those message m to the next node along the path being given.

For *Model B*, we need an additional property which indicates that the routing table is populated sufficiently often. This is formulated as follows.

P3: $\forall\ n_1, n_2 \in N.\ \Box\ ((\Box\Diamond\ pathX(n_1, n_2))\ \Rightarrow\ \Diamond\ (\exists p \in P(n_1, n_2).\ rt(n_1, n_2, p)))$

We also need to modify P2 to P2′ since paths for messages are obtained from the routing table, replacing $pathX(n_1, n_2)$ with $rt(n_1, n_2, p)$. We modify P2′ to P2′′′ to extend the **fairness** requirement, which includes the routing tables when the message m is at its position. The model also needs routing tables which are correct sufficiently often. In *Model B*, the condition P2$''$ is replaced by P2$'''$. Again this is a simplified form of the fairness property, which we extend later.

$$\begin{aligned}\mathsf{P2}'\colon \forall \mathsf{m} \in \mathsf{M}.\ \Box\ (((\Box\Diamond\ \exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}),\mathit{dest}(\mathsf{m})).\ \mathsf{rt}(\mathit{at}(\mathsf{m}),\mathit{dest}(\mathsf{m}),\mathsf{p}))\ \wedge\ \mathit{red}(\mathsf{m}))\\ \Rightarrow\ \Diamond\ (\mathit{green}(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}),\mathit{dest}(\mathsf{m})).\ \mathit{path}(\mathsf{m},\mathsf{p}))))\end{aligned}$$

Further details for *Model A* and *Model B* properties are presented in Chapter 5 together with the proofs.

4.7 Discussion

Here we model dynamic networks as concurrent systems using shared resources: a graph/network and a routing information table. Therefore, the formalism that is used here does not consider about communication between process. If we want to use other formalisms, such as CSP (Communicating Sequential Processes) to describe the communication between processes, then first we have to change the properties for each process (*Disrupter, Organizer* and *Updater*).

Chapter 5

Proving the correctness of message-passing in dynamic networks

This chapter shows that we can establish the correctness of dynamic networks at suitable levels of abstraction. By modelling at this level of abstraction, we are able to prove the properties of networks independently of the mechanisms in actual networks and therefore provide a factorisation of the proofs of correctness for actual dynamic networks.

We use discrete-time LTL [15] to describe the properties of execution traces of these multi-process systems. Some of the key properties which enable us to reason about network correctness are expressed as *fairness* constraints [36] in concurrent process models. We use strong fairness at the event level to express, for example, the relative frequency of network change to message motion and of routing table updates to network change.

Our aim is to formulate and prove the correctness of message-passing using the two abstract models being discussed in Chapter 4. We need to prove that, under certain conditions, all messages eventually reach their destination. We introduce a colouring of message according to their states. This is inspired by Gries [21] and Dijkstra [12]'s reasoning about on-the-fly garbage collection.

5.1 Using concurrent systems, trace properties and linear temporal logic to establish the correctness of message-passing in dynamic networks

We begin with a simple example using a colour-coding of messages and a simple trace property, showing how it decomposes into a connectivity requirement and a fairness condition. Then develop this into a full proof that all messages eventually reach their destinations, requiring fairness properties, connectivity and finiteness properties of networks and a livelock-free property. Two models of message-passing networks are considered.

5.1.1 Proof of a trace property of a dynamic network as a concurrent system

We wish to prove that any red message becomes green eventually under sufficient trace properties.
A. For the *Instantaneous Update* model, the properties we need are:

1. Paths exist sufficiently often
2. Messages get attention sufficiently often

B. For the *Update Process* model, we need an additional property

3. The routing table is populated sufficiently often.

We begin with the Instantaneous Update model.

5.1.1.1 Instantaneous Update model

The properties required can be expressed as:

P1: $\forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\Diamond\ \mathit{pathX}(\mathsf{n}_1, \mathsf{n}_2)$

P2: $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((\Box\Diamond\ \mathit{pathX}(\mathit{at}(\mathsf{m}), \mathit{dest}(\mathsf{m})))\ \wedge\ \mathit{red}(\mathsf{m})$
$\Rightarrow\ \Diamond\ (\mathit{green}(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}), \mathit{dest}(\mathsf{m})).\ \mathit{path}(\mathsf{m}, \mathsf{p}))))$

P1 is a **graph connectivity** condition (see note (4) below).
P2 is a simple form (which we will strengthen later) of a **fairness condition**: "if a path is available infinitely often then a path is allocated sometime in the future".

Lemma 1 *For traces satisfying P1 and P2, we have:*

$$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (\mathit{red}(\mathsf{m})\ \Rightarrow\ \Diamond\ \mathit{green}(\mathsf{m}))$$

Proof: Immediate
Note:

1. The interest here is in the following formulation: expressing the properties of dynamic networks as trace properties, some in terms of the fairness of process interaction, others as connectivity properties of graphs.

2. P2 is in a particularly simple form which we will need to extend: it **does not** consider message moves for example. We now consider the process update model.

5.1.1.2 The Instantaneous Update model as *Model A*

The following properties express the implementation of Instantaneous Update model as (*Model A*).

1. Paths exist sufficiently often

   G_connect : $\forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\Diamond\ \mathit{pathX}(\mathsf{n}_1, \mathsf{n}_2)$

2. Messages get attention sufficiently often

   M_Look1: $\forall \mathsf{m} \in \mathsf{M}.\forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\ ((\Box\Diamond\ \mathit{pathX}(\mathsf{n}_1, \mathsf{n}_2))\ \wedge\ \mathit{red}(\mathsf{m})$
   $\Rightarrow\ \Diamond\ (\mathit{green}(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}), \mathit{dest}(\mathsf{m})).\ \mathit{path}(\mathsf{m}, \mathsf{p}))))$

Lemma 1 . *For traces satisfying G_connect, and M_Look1*

$$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (\mathit{red}(\mathsf{m})\ \Rightarrow\ \Diamond \mathit{green}(\mathsf{m}))$$

Proof:

1. red(m) *hypothesis*

2. $\Diamond\ green(\mathsf{m})$ **by MP on [G_connect] and [M_Look1]**

3. hence $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (red(\mathsf{m}) \Rightarrow \Diamond green(\mathsf{m}))$

5.1.1.3 Update Process Model

In this case, messages obtain their paths from the routing table and so we need to express the idea that the routing table is correct sufficiently often. Here is a version of this property

P3: $\forall\ \mathsf{n}_1,\mathsf{n}_2 \in \mathsf{N}.\ \Box\ ((\Box\Diamond\ pathX(\mathsf{n}_1,\mathsf{n}_2)) \Rightarrow \Diamond\ (\exists \mathsf{p} \in \mathsf{P}(\mathsf{n}_1,\mathsf{n}_2).\ \mathsf{rt}(\mathsf{n}_1,\mathsf{n}_2,\mathsf{p})))$

We now need to modify P2 since paths for messages are obtained from the routing table. Again, this is a simplified form of the fairness property, which we extend later.

P2′: $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (((\Box\Diamond\ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}),dest(\mathsf{m})).\ \mathsf{rt}(at(\mathsf{m}),dest(\mathsf{m}),\mathsf{p}))\ \wedge\ red(\mathsf{m}))$
$\Rightarrow \Diamond\ (green(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}),dest(\mathsf{m})).\ path(\mathsf{m},\mathsf{p}))))$

Lemma 2 *(Lemma 1′) For traces satisfying P1, P2′ and P3, we have:*

$$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (red(\mathsf{m}) \Rightarrow \Diamond\ green(\mathsf{m}))$$

Proof
From P1 and P3 we have

$$\forall \mathsf{m} \in \mathsf{M}.\ \forall \mathsf{n}_1,\mathsf{n}_2 \in \mathsf{N}.\ (\Box\Diamond\ \exists \mathsf{p} \in \mathsf{P}(\mathsf{n}_1,\mathsf{n}_2).\mathsf{rt}(\mathsf{n}_1,\mathsf{n}_2,\mathsf{p}))$$

Then if for any $\mathsf{m} \in \mathsf{M}$ red(m) at any point, we have from P2′ $\Diamond\ green(\mathsf{m})$, hence

$$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (red(\mathsf{m}) \Rightarrow \Diamond\ green(\mathsf{m}))$$

Note that although the messages eventually become green in this model: they may not be allocated a **valid** path without additional conditions.
Notes:

1. The Update Process Model may have local routing table rt_n, n$\in$ N, or a global table rt. The account is the **same** as we abstract out the actual updating process.

2. We will need (but not so far) a **frame property**: given properties of the states

remain unchanged unless they are explicitly changed (eg. a message stays where it is unless it is moved).

3. The proofs so far are **trivial**. The difficulty is formulating at the right level of abstraction and isolating the required trace properties (*separation of concerns*). Later we should relate these properties to the behaviour of the system of processes and their actions.

4. The connectivity condition is very weak. It can be strengthened, e.g. with "durations", in which case the strengthened form is used in place of connectivity everywhere (eg. in P2 and P2′, weakening these conditions).

5.1.1.4 The Update Process Model as *Model B*

The following properties express the implementation of the Update Process Model as *Model B*.

1. Paths exist sufficiently often

   G_connect : $\forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\Diamond\ \mathit{pathX}(\mathsf{n}_1, \mathsf{n}_2)$

2. Messages get attention sufficiently often

   Msg_Look1′: $\forall \mathsf{m} \in \mathsf{M}.$
   $\Box\ (((\Box\Diamond\ \exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}), \mathit{dest}(\mathsf{m})).\ (\mathsf{rt}(\mathit{at}(\mathsf{m}), \mathit{dest}(\mathsf{m}), \mathsf{p})\ \wedge\ \mathit{avail}(\mathsf{p})))\ \wedge\ \mathit{red}(\mathsf{m}))$
   $\Rightarrow\ \Diamond\ (\mathit{green}(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}), \mathit{dest}(\mathsf{m})).\ \mathit{path}(\mathsf{m}, \mathsf{p}))))$

3. The routing table is populated sufficiently often.

   RT_correct : $\forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\ (\Box\Diamond\ \mathit{pathX}(\mathsf{n}_1, \mathsf{n}_2)\ \Rightarrow$
   $(\Diamond\ \exists \mathsf{p} \in \mathsf{P}(\mathsf{n}_1, \mathsf{n}_2).\ (\mathsf{rt}(\mathsf{n}_1, \mathsf{n}_2, \mathsf{p})\ \wedge\ \mathit{avail}(\mathsf{p}))))$

Lemma 1 *For traces satisfying G_connect, RT_correct, and Msg_Look1′*

$$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (\mathit{red}(\mathsf{m})\ \Rightarrow\ \Diamond \mathit{green}(\mathsf{m}))$$

Proof:

1. $\mathsf{red(m)}$ *hypothesis*

2. $\forall \mathsf{m} \in \mathsf{M}.\ \Box\Diamond\ \exists \mathsf{p} \in \mathsf{P}(\mathsf{n}_1, \mathsf{n}_2).\ (\mathsf{rt}(\mathsf{n}_1, \mathsf{n}_2, \mathsf{p})\ \wedge\ avail(\mathsf{p}))$ **by MP on [G_connect] and [RT_correct]**

3. $\Diamond\ green(\mathsf{m})$ **by MP on [2] and [Msg_Look1']**

4. hence $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (red(\mathsf{m})\ \Rightarrow\ \Diamond green(\mathsf{m}))$

5.1.2 Proof that all messages reach their destinations

To complete the proof in terms of trace properties, we need to establish that:

A. messages eventually move, and then using

- **(a)** finiteness of path
- **(b)** no livelock
- **(c)** modified P2/ P2'

and, for the Update Process model,

- **(d)** routing tables are **correct** sufficiently often

B. show that each message eventually reaches its destination.

We consider each model separately.

5.1.2.1 Instantaneous Update Model

1. Messages eventually move.
   For this we need the notion of messages moving; this requires:

   (a) the message-moving process (the *Organizer*) to access each message sufficiently often, and

   (b) that when it is accessed, it can be moved at some time.

We modify P2 (for this model) to extend the **fairness** requirement, as P2$''$ (fairness)

P2”: $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((\Box\Diamond\ pathX(at(\mathsf{m}), dest(\mathsf{m}))) \Rightarrow$
$\Diamond\ (black(\mathsf{m}) \wedge$
$(red(\mathsf{m})\ \Rightarrow\ \bigcirc(green(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m},\mathsf{p})))) \wedge$
$(\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ (green(\mathsf{m})\ \wedge\ path(\mathsf{m},\mathsf{p})\ \wedge\ at(\mathsf{m}) \neq dest(\mathsf{m})\ \Rightarrow$
$(up(1st\_e(\mathsf{p}))\ \wedge\ \bigcirc(green(\mathsf{m})\ \wedge\ path(\mathsf{m}, tl(\mathsf{p}))\ \wedge\ at(\mathsf{m}) = next\_node(\mathsf{p}))))) \wedge$
$((green(\mathsf{m})\ \wedge\ at(\mathsf{m}) = dest(\mathsf{m}))\ \Rightarrow$
$\bigcirc\ black(\mathsf{m}))))$

up(e) means e is connected.
This may need further development: it attempts to say that if paths are available infinitely often then something desirable happens:

1. a message becomes/remains black, or
2. a red message becomes green and is given a path, or
3. a green message advances

5.1.2.2 The Instantaneous Update Model as *Model A*

Following properties express the implementation of the Instantaneous Update Model as *Model A* for the reaching destination case.

Msg_Look2:
$\forall \mathsf{m} \in \mathsf{M}. \forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\ ((\Box\Diamond\ \ pathX(at(\mathsf{m}), dest(\mathsf{m}))) \Rightarrow$
$\Diamond\ (black(\mathsf{m}) \wedge$
$(red(\mathsf{m})\ \Rightarrow\ \bigcirc(green(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m},\mathsf{p})))) \wedge$
$(\ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).((green(\mathsf{m})\ \wedge\ path(\mathsf{m},\mathsf{p})\ \wedge\ at(\mathsf{m}) \neq dest(\mathsf{m}))$
$\Rightarrow\ (up(\text{1st\_e}(\mathsf{p}))\ \wedge\ \bigcirc(green(\mathsf{m})\ \wedge\ path(\mathsf{m}, \text{suffix}(\mathsf{p}))\ \wedge$
$at(\mathsf{m}) = \text{next\_node}(\mathsf{p}))))) \wedge$
$((green(\mathsf{m})\ \wedge\ at(\mathsf{m}) = dest(\mathsf{m}))\ \Rightarrow\ \bigcirc\ black(\mathsf{m}))))$

We need to define livelock-free, finiteness of paths and some other properties as follows.

Properties 1 *(**Livelock-free**). Here we define **Livelock-free** as:*

$$\forall \mathsf{m} \in \mathsf{M}.\ \neg\ \Box\Diamond\ (\mathit{green}(\mathsf{m})\ \Rightarrow\ \bigcirc \mathit{red}(\mathsf{m}))$$

which means that $\mathsf{GR}(\mathsf{m})$ finitely often.

Properties 2 *(**Finiteness of paths**). Here we define **Finiteness of paths** as:*

$$\forall \mathsf{m} \in \mathsf{M}.\ ((\mathit{green}(\mathsf{m})\ \wedge\ \neg\Diamond\ \mathit{red}(\mathsf{m}))\ \Rightarrow\ (\Diamond\ \mathit{black}(\mathsf{m})))$$

which means that if a message m is green and there is no potential for it to become red eventually then message m will eventually turn black . This can only holds if the *Organizer* checks message m infinitely often (Msg_Look2).

We want to show that for each m there is a point in the trace at which $\Box\ \mathit{green}(\mathsf{m})\ \vee\ \mathit{black}(\mathsf{m})$ hence $\Diamond \mathit{black}(\mathsf{m})$.

Lemma 2 . *For traces satisfying G_connect, Msg_Look2, and Lemma 1:*

$$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (\exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}),\mathit{dest}(\mathsf{m})).\ (\mathit{green}(\mathsf{m})\ \wedge\ \mathit{path}(\mathsf{m},\mathsf{p})\ \Rightarrow\ \Diamond\ \mathit{moved}(\mathsf{m},\mathsf{p})))$$

with

$$\begin{aligned}&\mathit{moved}(\mathsf{m},\mathsf{p}) \equiv\\ &((\mathit{green}(\mathsf{m})\ \wedge\ \mathit{path}(\mathsf{m},\mathsf{p})\ \wedge\ \mathit{at}(\mathsf{m}) \neq \mathit{dest}(\mathsf{m}))\ \Rightarrow\\ &(\mathit{up}(\text{1st\_e}(\mathsf{p}))\ \wedge\ \bigcirc(\mathit{green}(\mathsf{m})\ \wedge\ \mathit{path}(\mathsf{m},\text{suffix}(\mathsf{p}))\ \wedge\ \mathit{at}(\mathsf{m}) = \text{next\_node}(\mathsf{p}))))\end{aligned}$$

Proof:

1. $\mathit{green}(\mathsf{m})\ \wedge\ \exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}),\mathit{dest}(\mathsf{m})).\ \mathit{path}(\mathsf{m},\mathsf{p})$ **follow from proof of Lemma 1**

2. $\mathit{at}(\mathsf{m}) \neq \mathit{dest}(\mathsf{m})$ **hypothesis**

3. $\forall \mathsf{m} \in \mathsf{M}.\ (\Diamond\ (\exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}),\mathit{dest}(\mathsf{m})).\ ((\mathit{green}(\mathsf{m})\ \wedge\ \mathit{path}(\mathsf{m},\mathsf{p})\ \wedge$ $\mathit{at}(\mathsf{m}) \neq \mathit{dest}(\mathsf{m}))\ \Rightarrow\ (\mathit{up}(\text{1st\_e}(\mathsf{p}))\ \wedge\ \bigcirc(\mathit{green}(\mathsf{m})\ \wedge\ \mathit{path}(\mathsf{m},\text{suffix}(\mathsf{p}))\ \wedge$ $\mathit{at}(\mathsf{m}) = \text{next\_node}(\mathsf{p}))))))$ **by MP on [G_connect] and [Msg_Look2]**

4. hence $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ (green(\mathsf{m})\ \wedge\ path(\mathsf{m},\mathsf{p})$ $\Rightarrow\ \Diamond\ moved(\mathsf{m},\mathsf{p})))$

Lemma 3 . *For traces satisfying finiteness of path, Livelock-free definition, Lemma 1, Lemma 2, G_connect and Msg_Look2:*

$$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((green(\mathsf{m})\ \wedge\ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m},\mathsf{p}))\ \Rightarrow\ \Diamond black(\mathsf{m}))$$

Proof:

1. $green(\mathsf{m})\ \wedge\ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m},\mathsf{p})$ **follows from proof of Lemma 1**
2. $\Diamond\neg red(\mathsf{m})$ **([Livelock-free def.] and [Lemma1])**
3. $\forall \mathsf{m} \in \mathsf{M}.\ ((green(\mathsf{m})\ \wedge\ at(\mathsf{m}) = dest(\mathsf{m}))\ \Rightarrow\ \bigcirc\ black(\mathsf{m}))$ **by MP on [G_connect] and [Msg_Look2]**
4. $\Diamond\ black(\mathsf{m})$ **by MP on [1],[2], [Lemma 2], [3] and [finiteness of path]**
5. hence $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((green(\mathsf{m})\ \wedge\ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m},\mathsf{p}))\ \Rightarrow$ $\Diamond black(\mathsf{m}))$

By Lemma 1, 2, and 3, we have Theorem : $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (red(\mathsf{m}) \Rightarrow \Diamond black(\mathsf{m}))$

5.1.2.3 Update Process Model

1. Messages eventually move.
   For this we need the notion of messages moving; this requires:
   (a) the message-moving process (the *Organizer*) to access each message sufficiently often, and
   (b) that when it is accessed, it can be moved at some time.

We modify P2 (for this model) to extend the **fairness** requirement, as P2''' (fairness)

P2''': $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((\Box\Diamond\ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ (\mathsf{rt}(at(\mathsf{m}), dest(\mathsf{m}), \mathsf{p}))) \Rightarrow$
$\Diamond\ (black(\mathsf{m}) \wedge$
$(red(\mathsf{m}) \ \Rightarrow\ \bigcirc(green(\mathsf{m}) \ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m}, \mathsf{p})))) \wedge$
$(\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ ((green(\mathsf{m}) \ \wedge\ at(\mathsf{m}) \neq dest(\mathsf{m}) \ \wedge$
$pathX(at(\mathsf{m}), dest(\mathsf{m}) \ \wedge\ path(\mathsf{m}, \mathsf{p}))) \ \Rightarrow$
$(up(1st\_e(\mathsf{p})) \ \wedge\ \bigcirc(green(\mathsf{m}) \ \wedge\ path(\mathsf{m}, tl(\mathsf{p})) \ \wedge$
$at(\mathsf{m}) = next\_node(\mathsf{p}))))) \wedge$
$((green(\mathsf{m}) \ \wedge\ at(\mathsf{m}) = dest(\mathsf{m})) \ \Rightarrow\ \bigcirc\ black(\mathsf{m}))))$

up(e) means e is connected.
This may need further development: it attempts to say that if paths are available infinitely often then something desirable happens:

1. a message becomes/ remains black, or

2. a red message becomes green and is given an available path at routing table, or

3. a green message advances using path p which is **available at routing table and exists in the network (valid)**.

5.1.2.4 The Update Process Model as *Model B*

Following properties express the implementation of the Update Process Model as *Model B* for the reaching destination case.

Msg_Look2':

$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((\Box\Diamond\ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ (\mathsf{rt}(at(\mathsf{m}), dest(\mathsf{m}), \mathsf{p})\ \wedge\ avail(\mathsf{p})))$
$\Rightarrow\ (\Diamond\ (black(\mathsf{m})\ \vee$
$((red(\mathsf{m})\ \Rightarrow\ \bigcirc(green(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m},\mathsf{p}))))\ \wedge$
$(\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ ((green(\mathsf{m})\ \wedge\ at(\mathsf{m}) \neq dest(\mathsf{m})\ \wedge$
$pathX(at(\mathsf{m}), dest(\mathsf{m})\ \wedge\ path(\mathsf{m},\mathsf{p})))\ \Rightarrow$
$(up(1st\_e(\mathsf{p}))\ \wedge\ \bigcirc(green(\mathsf{m})\ \wedge\ path(\mathsf{m}, tl(\mathsf{p}))\ \wedge\ at(\mathsf{m}) = next\_node(\mathsf{p})))))\ \wedge$
$((green(\mathsf{m})\ \wedge\ at(\mathsf{m}) = dest(\mathsf{m}))\ \Rightarrow\ \bigcirc\ black(\mathsf{m}))))$

Lemma 2 . *For traces satisfying G_connect, Msg_Look2', and Lemma 1:*

$$\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ (green(\mathsf{m})\ \wedge\ path(\mathsf{m},\mathsf{p})\ \Rightarrow\ \Diamond\ moved(\mathsf{m},\mathsf{p})))$$

with

$moved(\mathsf{m},\mathsf{p}) \equiv$
$((green(\mathsf{m})\ \wedge\ at(\mathsf{m}) \neq dest(\mathsf{m})\ \wedge\ pathX(at(\mathsf{m}), dest(\mathsf{m})\ \wedge\ path(\mathsf{m},\mathsf{p})))\ \Rightarrow$
$(up(1st\_e(\mathsf{p}))\ \wedge\ \bigcirc(green(\mathsf{m})\ \wedge\ path(\mathsf{m}, tl(\mathsf{p}))\ \wedge\ at(\mathsf{m}) = next\_node(\mathsf{p}))))$

Proof:

1. $green(\mathsf{m})\ \wedge\ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m},\mathsf{p})$ **follows from proof of Lemma 1**
2. $at(\mathsf{m}) \neq dest(\mathsf{m})$ **hypothesis**
3. $\forall \mathsf{m} \in \mathsf{M}.\ \Box\Diamond\ \exists \mathsf{p} \in \mathsf{P}(\mathsf{n}_1, \mathsf{n}_2).\ (\mathsf{rt}(\mathsf{n}_1, \mathsf{n}_2, \mathsf{p})\ \wedge\ avail(\mathsf{p}))$ **by MP on [G_connect] and [RT_correct]**
4. $\forall \mathsf{m} \in \mathsf{M}.\ (\Diamond\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ ((green(\mathsf{m})\ \wedge\ at(\mathsf{m}) \neq dest(\mathsf{m})\ \wedge$
   $pathX(at(\mathsf{m}), dest(\mathsf{m})\ \wedge\ path(\mathsf{m},\mathsf{p})))\ \Rightarrow$

$(up(1st\_e(\mathsf{p})) \ \wedge \ \bigcirc(green(\mathsf{m}) \ \wedge \ path(\mathsf{m},tl(\mathsf{p})) \ \wedge \ at(\mathsf{m}) = next\_node(\mathsf{p})))))$ **by MP on [3] and [Msg_Look2']**

5. hence $\forall \mathsf{m} \in \mathsf{M}. \ \Box \ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}),dest(\mathsf{m})). \ (green(\mathsf{m}) \ \wedge \ path(\mathsf{m},\mathsf{p}) \ \Rightarrow \ \Diamond \ moved(\mathsf{m},\mathsf{p})))$

Lemma 3 . *For traces satisfying finiteness of path, Livelock-free definition, Lemma 1, Lemma 2, G_connect and Msg_Look2':*

$$\forall \mathsf{m} \in \mathsf{M}. \ \Box \ ((green(\mathsf{m}) \ \wedge \ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}),dest(\mathsf{m})). \ path(\mathsf{m},\mathsf{p})) \ \Rightarrow \ \Diamond black(\mathsf{m}))$$

Proof:

1. $green(\mathsf{m}) \ \wedge \ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}),dest(\mathsf{m})). \ path(\mathsf{m},\mathsf{p})$ **follow from proof of Lemma 1**
2. $\Diamond \neg red(\mathsf{m})$ **([Livelock-free def.] and [Lemma1])**
3. $\forall \mathsf{m} \in \mathsf{M}. \ ((green(\mathsf{m}) \ \wedge \ at(\mathsf{m}) = dest(\mathsf{m})) \ \Rightarrow \ \bigcirc \ black(\mathsf{m}))$ **by MP on [G_connect] and [Msg_Look2']**
4. $\Diamond \ black(\mathsf{m})$ **by MP on [1],[2], [Lemma 2], [3] and [finiteness of path]**
5. hence $\forall \mathsf{m} \in \mathsf{M}. \ \Box \ ((green(\mathsf{m}) \ \wedge \ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}),dest(\mathsf{m})). \ path(\mathsf{m},\mathsf{p})) \ \Rightarrow \ \Diamond black(\mathsf{m}))$

By Lemma 1, 2, and 3, we have Theorem : $\forall \mathsf{m} \in \mathsf{M}. \ \Box \ (red(\mathsf{m}) \Rightarrow \Diamond black(\mathsf{m}))$
Corollary
We know that :

$$\{black(\mathsf{m})\} \ \mathsf{BB}(\mathsf{m}) \ \{black(\mathsf{m})\}$$

and no other actions change from black (m) to another colour. Therefore:

$$\Diamond black(\mathsf{m}) \ \Rightarrow \ \Diamond\Box \ black(\mathsf{m})$$

Proof of interference free of the three processes (*Disrupter*, *Organizer*, and *Updater*) is presented in Appendix A.5.

5.2 Proof techniques for concurrent systems and frame properties

Proofs in this dynamic network research do not mention the frame problem (which has been explained in Section 2.8), even though this feature has been used in preliminary cases of on-the-fly sorting (Section 3.3.2). The dynamic network model does not mention the state of other messages when a particular message reaches a node or destination. The reasons behind this issue are related to the changing of individuals and the changing of properties. Here, the dynamic network model uses a constant domain model and therefore the model uses the same set at each world. Moreover, instead of expressing the *existence* about individual changing, the dynamic network model express the *usable* links (up and down). Regarding the changing of properties, this dynamic network model contains descriptions of the form of actions required implicitly for the type of changes, particularly through the general properties of the dynamic network model. These general properties can be regarded as the explanation-closure axioms for the model.

The logical reason why there are no frame properties for dynamic networks is as follows. First order logic and modal/temporal logic use a Kripke model. This dynamic networks research uses a constant domain model, which employs the same set at each world. So, instead of existence, our focus is the usability of individuals in the model. In formula, rather than:

$$\Box\ \forall x.\phi$$

the formula that holds for constant domain model is:

$$\frac{\forall x.\Box\phi}{\Box\phi[a/x]}$$

This is about changing individuals and this only holds for the constant domain model.

The OTF sorting case does not use separation concerns to abstract the model. Therefore, this case needs and uses frame axioms, and this case is much more simple than dynamic networks. In the dynamic networks case, instead of using frame axioms, it uses separation concerns for mechanisms, so it uses a hierarchical model approach, then defines the general properties which implicitly point to the actions which are required for a type of change (in this case, to change the colour of the message and to move the messages).

5.3 Discussion

Here we prove :

$$\forall \mathsf{m} \in \mathsf{M}.\Diamond\Box black(\mathsf{m})$$

which means that every message $\mathsf{m}$will eventually be delivered. This will need the network which to be sufficiently connected sufficiently often, for the routing table/information to be sufficiently correct sufficiently often and for there to be no persistent livelock.

Theorem

For any message m, if properties (1-6) and fairness conditions (1-6) holds then

$$P1: \ red(\mathsf{m}) \ \Rightarrow \ \Diamond \ black(\mathsf{m})$$

To prove that any message $\mathsf{m}$will eventually reach the destination, there is a definition of a general fairness $\mathcal{F}$ as follows:

If each message can infinitely often reach its destination $\Rightarrow$ All messages will eventually reach their destinations.

$$\mathcal{F}: \forall \ \mathsf{m} \in \mathsf{M}. \ (\Box\Diamond \ ((path(\mathsf{m},\mathsf{p})) \ \wedge \ dest(\mathsf{m}) = \mathsf{d} \ \wedge \ dest(\mathsf{p}) = \mathsf{d}) \ \Rightarrow \\ \Diamond at(\mathsf{m}) = \mathsf{d})$$

If this condition holds, and following condition also holds : Every message can infinitely often reach its destination,

$$\mathcal{T}: \ \Box\Diamond \ (path(\mathsf{m},\mathsf{p}) \ \wedge \ dest(\mathsf{m}) = \mathsf{d} \ \wedge \ dest(\mathsf{p}) = \mathsf{d})$$

then :

$$(\mathcal{F} \ \wedge \ \mathcal{T}) \ \Rightarrow \ (\mathcal{E}_1 \ \wedge \ \mathcal{E}_2)$$

with:

Each pair of nodes is linked infinitely often by path

$$\mathcal{E}_1: \forall \ \mathsf{e} \in \mathsf{p}. \ \Box\Diamond \ connected(\mathsf{e})$$

and each message is moving infinitely often.

$$\mathcal{E}_2 : \forall\ \mathsf{m} \in \mathsf{M}.\ \Box\Diamond\ path(\mathsf{m},\mathsf{p})$$

General fairness $\mathcal{F}$ is refined based on the meaning of *can infinitely often*, as follows:

1. The *Organizer* process is enabled infinitely often so message $\mathsf{m}$can move along the route to reach the destination (fairness of the *Organizer* $\|$ *Updater*).

$$\mathcal{F}_1 :\ \Box\Diamond\ avail(\mathsf{p})\ \Rightarrow\ \Diamond\ path(\mathsf{m},\mathsf{p})$$

2. The *Disrupter* process is enabled infinitely often but the message still can get through (fairness of *Disrupter* $\|$ *Organizer*).

$$\mathcal{F}_2 :\ \Box\Diamond\ connected(\mathsf{e})\ \Rightarrow\ \Diamond\ path(\mathsf{m},\mathsf{p})$$

3. If the *Disrupter* process is enabled infinitely often then eventually there is a path to move the message (fairness of *Disrupter* $\|$ *Updater*).

$$\begin{aligned}&\mathcal{F}_3 : \forall n_1, n_2 \in \mathsf{N}.\\ &\quad \Box\ (\Box\Diamond\ \exists \mathsf{p} \in \mathsf{P}(n_1,n_2).\exists \mathsf{e} \in \mathsf{E}(n_1,n_2).\ ((elmt(\mathsf{e},\mathsf{p})\ \wedge\ disconnected(\mathsf{e}))\\ &\quad \Rightarrow\ \bigcirc connected(\mathsf{e}))\ \Rightarrow\ \Diamond\ (\mathsf{rt}(n_1,n_2,\mathsf{p})\ \wedge\ avail(\mathsf{p})))\end{aligned}$$

By implication, it is obvious that $\mathcal{F}_3 \wedge \mathcal{F}_1 \Rightarrow \mathcal{F}_2$. This description cannot capture: how a message m moves along the route; livelock behaviour that can occur in traces; and how the message m eventually reach the destination. To observe the progress of the message m on traces, therefore we use the colour notification and then define some axioms for the action of *Disrupter*, *Organizer*, and *Updater* processes. The model uses LTL to express the behaviour of traces.

Chapter 6

Simulation of Dynamic Network models

This chapter focuses on implementation of the two abstract models that been described in Chapter 4. There are discussions which relating the abstract to implementation, why and how the abstract models are implemented in multi-threaded Java programs. These also include incorporation of the runtime verification systems RULER [8], to analyse execution traces to test whether model instances satisfy the modal correctness for message delivery.

6.1 Sequential vs Concurrent programs

A sequential program consists of some instructions in which each instruction executed one after the other from start to finish. Sequential program execution is deterministic. Using the same input data, a sequential program will always execute the same sequence of instructions and will always produce the same output. The order of its instructions specifies the order of execution. The successors of instructions must be executed without any overlap with another instruction. Therefore, the program executes a sequence of instructions in a pre-defined order.

A concurrent program is a set of sequential programs that can be executed in parallel [8]. Suppose we have three concurrent processes P || Q || R. The executions of P, Q and R may overlap. However, the instructions from each process must be strictly ordered. Furthermore, different executions of a concurrent program will produce different instructions ordering. The concurrent program execution is non-deterministic. The concurrent program can deliver different results when repeatedly run with the same input data.

There are two basic units of execution in concurrent programming: processes and threads. Java program concerns with threads. Here, the dynamic network models are implemented as multi threads program.

The implementation of *Disrupter*||*Updater*||*Organizer* model uses several shared variables (see also Figure 4.3 in Section 4.4.2), namely link status L, and routing tables R. The initial input is given by executing one of the actions of the *Disrupter* process. The output as the result of action in the *Disrupter* process will be the input for the *Updater* process. The *Disrupter* process accesses shared variable L when performing its actions. The *Updater* process accesses the shared variable L and R for performing its actions. The output of *Updater* process will be the input for *Organizer* process. The *Organizer* process will access shared variable R and variable M (variable of the message list) for performing its actions.

Another process, called "Router" process, has a role to recalculate the path. The "Router" process description is as follows.

```
Process Router
begin
  while true
    if (there is an action from disrupter) then
      update routing table instantaneously
    endif
end
```

There is a different implementation of process *Router* in a program, depend on which model of symple dynamic network system that we use. In *Model A*, *Router* process is implemented as a *ghost* process called *pathFinder* in which this process is not regarded as the part of *Model A*. For implementation of *Model B*, *Router* process will be an additional process called *Updater* process. Appendix A.4 presents the *Updater* algorithm.

The source code which defines each processes are as follows.

```
...
disrupter = new Thread(
        (Runnable) new AutomataInterpreter(IdDisrupter, myGraph, frame, desk, graph, parent, v0,
        v1, v2, v3, v4, v5,  mPos1, mPos2, mPos3, mPos4, mPos5, mPos6, mPos12, mPos22, mPos32,
        mPos42, mPos52, mPos62,  frame2, myMessage, endTime, output, myRoutingTable, graphComponent,
        MyGPar, e12, e23, e24, e35, e45, e56),  "Disrupter");
organizer = new Thread(
        (Runnable) new AutomataInterpreter(IdOrganizer, myGraph, frame, desk, graph, parent, v0,
        v1, v2, v3, v4, v5,  mPos1, mPos2, mPos3, mPos4, mPos5, mPos6, mPos12, mPos22, mPos32,
        mPos42, mPos52, mPos62, frame2, myMessage, endTime, output, myRoutingTable, graphComponent,
        MyGPar, e12, e23, e24, e35, e45, e56),  "Organiser");
```

```
if (model.trim().contentEquals("DN model : Disrupter || Organizer || Updater")) {
      updater = new Thread(
      (Runnable) new AutomataInterpreter(IdUpdater, myGraph, frame, desk, graph, parent, v0,
      v1, v2, v3, v4, v5, mPos1, mPos2, mPos3, mPos4, mPos5, mPos6, mPos12, mPos22, mPos32,
      mPos42, mPos52, mPos62, frame2, myMessage, endTime, output, myRoutingTable,
      graphComponent,  MyGPar, e12, e23, e24, e35, e45, e56), "Updater");
      updater.start();
}
else
      {
   pathFinder = new Thread(
            (Runnable) new pathFinderProc(sleepPathF, myGraph, myMessage, MyGPar),
            "pathFinder");
   pathFinder.start();
      }
                  ...
```

AutomataInterpreter class is used as the base class for *Disrupter*, *Organizer*, and *Updater* process. The *pathFinder* class has its own base class. This class will determine which threads running over time.

6.2 Implementation of *Model A*

Here, we implement the algorithm for *Model A*. Therefore, we have *Disrupter* thread and *Organizer* thread. Appendix A.2 and A.3 present the *Disrupter* algorithm and *Organizer* algorithm respectively. The implementation of alternate change between connect and disconnect state of a link may as *uniform changes* (i.e. after connecting then disconnecting, or vice versa), or *random changes* (i.e. after connecting then it remains connected for multiple times of turn).

no.	**state1**	**pred**	**action**	**state2**
1	1	d	up	1
2	1	c	down	1

Table 6.1: *Disrupter* automata definition.

Table 6.1 presents the *Disrupter* automata definition. This shows that if the precondition is ‘d’ (disconnect) then the expected action is ‘up’ (connected). Meanwhile, if the precondition is ‘c’ (connected) then the expected action is ‘down’ (disconnected). Notice that the definition represents the uniform changes.

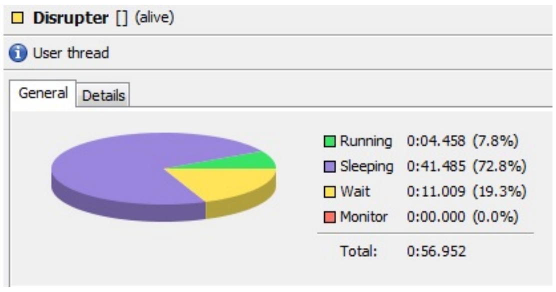
Table 6.2 presents the *Organizer* automata definition of *Model A*. Notation: ‘1’, ‘2’, and ‘3’ denote the state; ‘s’, ‘i’, ‘d’ and ‘ep’ denote source, intermediate, destination and empty path, respectively; ‘edgeNotOK(path[index])’ means that there is a disrupted edge in the path; ‘AltR’ means that there is an alternative route, so the action includes ‘red’ and ‘green’ (i.e. blocked then available); and ‘edgeOK(path[index])’ means that there is no disrupted edge in the path (path is available).

no.	**state1**	**currentPos**	**pred**	**action**	**state2**
1	1	s	edgeNotOK(path[index])	wait	1
2	1	i	edgeNotOK(path[index])	wait	1
3	1	s	edgeOK(path[index])	green	2
4	1	i	edgeOK(path[index])	green	2
5	1	ep	-	green	2
				black	3
6	2	s	edgeNotOK(path[index])	red	1
7	2	i	edgeNotOK(path[index])	red	1
8	2	s	edgeNotOK(path[index]) + AltR	red	1
				green	2
9	2	i	edgeNotOK(path[index]) + AltR	red	1
				green	2
10	2	s	edgeOK(path[index])	move	2
11	2	i	edgeOK(path[index])	move	2
12	2	d	edgeOK(path[index])	move	2
				black	3
13	2	ep	-	black	3

Table 6.2: *Organizer* automata definition of *Model A*.

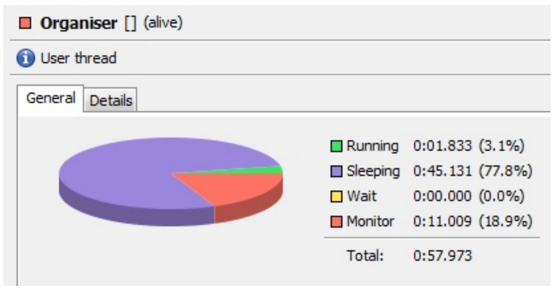
Routing update occurs instantaneously. When there is a change in link status within network caused by *Disrupter*, then a *pathFinder* process will update the available route for the messages while *Disrupter* and *Organizer* wait until it finishes updating.

Figure 6.1, 6.2, and 6.3 show the actual part of the Java threads running. Here we can see that most of the running part is dominated by the system, not the program. Therefore, for instance, if we set the frequency of *Disrupter* || *Organizer* = 1 || 1, then this does not mean that the running part of each thread will be definitely 1 || 1. Moreover, there is a *pathFinder* thread that will lock the shared variables when it recalculates the available path; while the *Disrupter* process in the waiting position and the *Organizer* monitor the event before getting the access to the shared variables.


Disrupter [] (alive)
User thread
General | Details

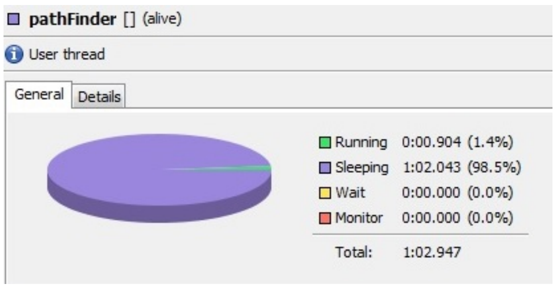
```
🟩 Running   0:04.458 (7.8%)
🟪 Sleeping  0:41.485 (72.8%)
🟨 Wait      0:11.009 (19.3%)
🟥 Monitor   0:00.000 (0.0%)
   Total:    0:56.952
```

Figure 6.1: The *Disrupter* running part of the implementation of *Model A*.


Organiser [] (alive)
User thread
General | Details

```
🟩 Running   0:01.833 (3.1%)
🟪 Sleeping  0:45.131 (77.8%)
🟨 Wait      0:00.000 (0.0%)
🟥 Monitor   0:11.009 (18.9%)
   Total:    0:57.973
```

Figure 6.2: The *Organizer* running part of the implementation of *Model A*.


pathFinder [] (alive)
User thread
General | Details

```
🟩 Running   0:00.904 (1.4%)
🟪 Sleeping  1:02.043 (98.5%)
🟨 Wait      0:00.000 (0.0%)
🟥 Monitor   0:00.000 (0.0%)
   Total:    1:02.947
```

Figure 6.3: The *pathFinder* running part of the implementation of *Model A*.

6.3 Implementation of *Model B*

Implementation of *Model B* consists of three threads: *Disrupter*, *Organizer*, and *Updater*. The automata for *Disrupter* remains the same. Table 6.3 presents the *Organizer* automata definition of *Model B*. Notation: '1', '2', and '3' denote the state; 's', 'i', 'd' and 'ep' denote source, intermediate, destination and empty path, respectively; 'notAvail(p)' means that the path 'p' is unavailable; 'AltR' means that there is an alternative route, so the action includes 'red' and 'green' (i.e. blocked then available); and 'avail(p)' means that the path is available.

no.	**state1**	**currentPos**	**pred**	**action**	**state2**
1	1	s	notAvail(p)	wait	1
2	1	i	notAvail(p)	wait	1
3	1	s	avail(p)	green	2
4	1	i	avail(p)	green	2
5	1	ep	-	green	2
				black	3
6	2	s	notAvail(p)	red	1
7	2	i	notAvail(p)	red	1
8	2	s	notAvail(p) + AltR	red	1
				green	2
9	2	i	notAvail(p) + AltR	red	1
				green	2
10	2	s	avail(p)	move	2
11	2	i	avail(p)	move	2
12	2	d	avail(p)	move	2
				black	3
13	2	ep	-	black	3

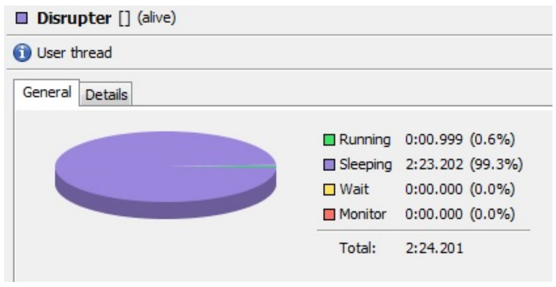
Table 6.3: *Organizer* automata definition of *Model B*.

Table 6.4 presents the *Updater* automata definition. Notation: '1', and '2' denote the state; 'edgeNotOKPath(e, p)' means that there is a disrupted edge 'e' in the path 'p' so the path 'p' is unavailable; and 'edgeOKPath(e, p)' means that there is no disrupted edge in the path 'p' (path 'p' is available).

Figure 6.4, 6.5, and 6.6 show the actual part of the Java threads running. Here we can see that most of the running part is dominated by the system, not the program. Therefore, for instance, if we set the frequency of *Disrupter* || *Organizer* || *Updater* = 1 || 1 || 1, then this does not mean that the running part of each threads will be definitely 1 || 1 || 1.

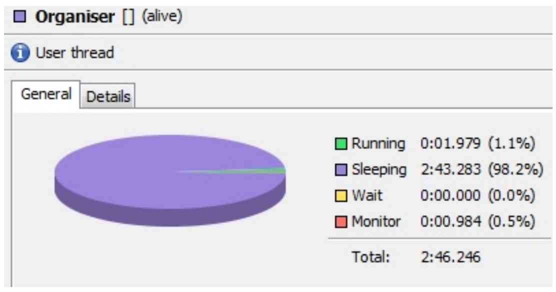
no.	state1	pred	action	state2
1	1	edgeOKPath(e, p)	avail(p)	2
2	1	edgeNotOKPath(e, p)	unavail(p)	1
3	2	edgeOKPath(e, p)	avail(p)	2
4	2	edgeNotOKPath(e, p)	unavail(p)	1

Table 6.4: *Updater* automata definition.



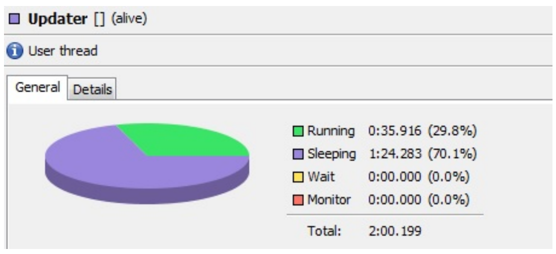
```
🟩 Running   0:00.999 (0.6%)
🟪 Sleeping  2:23.202 (99.3%)
🟨 Wait      0:00.000 (0.0%)
🟥 Monitor   0:00.000 (0.0%)
   Total:    2:24.201
```

Figure 6.4: The *Disrupter* running part of the implementation of *Model B*.



```
🟩 Running   0:01.979 (1.1%)
🟪 Sleeping  2:43.283 (98.2%)
🟨 Wait      0:00.000 (0.0%)
🟥 Monitor   0:00.984 (0.5%)
   Total:    2:46.246
```

Figure 6.5: The *Organizer* running part of the implementation of *Model B*.



```
🟩 Running   0:35.916 (29.8%)
🟪 Sleeping  1:24.283 (70.1%)
🟨 Wait      0:00.000 (0.0%)
🟥 Monitor   0:00.000 (0.0%)
   Total:    2:00.199
```

Figure 6.6: The *Updater* running part of the implementation of *Model B*.

6.4 Colouring decision

The crucial thing of this dynamic network model is when the colouring (either red or green) takes place, who did that colouring and what the reason of this. Colouring of the message base on what the message believes is doing. Type of dynamic network models differentiate this mechanism.

In *Model A* (*Organizer* and *Disrupter*), the *Organizer* colour the message m based on the availability of path for the message. Only the *Organizer* process can do the colouring. *Organizer* pick any message m to be delivered then check the message position on network and check the state of edge to the next node. If all conditions are satisfied then *Organizer* colour the message as green, otherwise red.

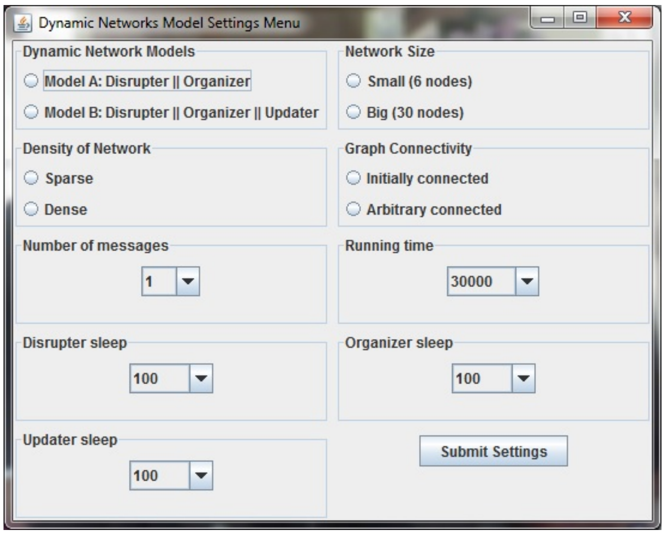
In *Model B* (*Organizer*, *Updater* and *Disrupter*), only *Organizer* can do the colouring as well. The *Organizer* picks any message m to be delivered, then checks the message position on network, check the state of edge to the next node, and check the availability of path p in the routing table. If all conditions are satisfied then *Organizer* colour the message as green, otherwise red.

The different action will take based on current state of *Organizer*. If the current state is 'red' then if conditions are satisfied the message m will be coloured green, otherwise message m will wait (red(m)). If current state is 'green', then if all conditions are satisfied, message will be moved and coloured green. Otherwise, message m will be coloured red, and if there is an alternate path, then message m will be coloured green(m).

6.5 Example of a program run

Figure 6.7 provide setting menu to choose the parameters such as:

- model options: *Model A* (*Disrupter* || *Organizer*), or *Model B* (*Disrupter* || *Organizer* || *Updater*);
- graph options for the network: Initially connected (all edges within the network are 'on' (initially connected)) or Arbitrary connected (initially, some edges status are 'on').
- options for the number of messages that will be used in the simulation. The number of messages is fixed along run time and may influence the frequency of message being looked at by the *Organizer*.


Dynamic Networks Model Settings Menu

Dynamic Network Models
- ○ Model A: Disrupter || Organizer
- ○ Model B: Disrupter || Organizer || Updater

Network Size
- ○ Small (6 nodes)
- ○ Big (30 nodes)

Density of Network
- ○ Sparse
- ○ Dense

Graph Connectivity
- ○ Initially connected
- ○ Arbitrary connected

Number of messages: 1

Running time: 30000

Disrupter sleep: 100

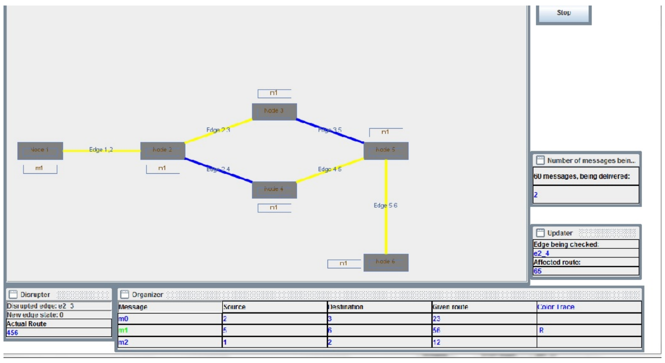
Organizer sleep: 100

Updater sleep: 100

Figure 6.7: Setting menu before running the multi-threaded programs.

- the options for setting the running time for simulation program.
- the options for setting the frequency for how long the *Disrupter* process to sleep, in msecs.
- the options for setting the frequency for how long the *Organizer* process to sleep, in msecs.
- the options for setting the frequency for how long the *Updater* process to sleep, in msecs.

Figure 6.8 shows the visualisation of the three process running. Not all the messages can be shown in the picture.



Number of messages bein...
60 messages, being delivered:
2

Updater
Edge being checked:
e2_4
Affected route:
65

Disrupter
Disrupted edge: e2_3
New edge state: 0
Actual Route
456

Organizer

Message	Source	Destination	Given route	Color Trace
m0	2	3	23	
m1	5	6	56	R
m2	1	2	12	

Figure 6.8: Example of program running of *Model B* on a sparse network.

Chapter 7

RV and trace properties of dynamic network models

In this chapter, we discuss the use of a runtime verification to analyse execution traces to test whether model instances satisfy modal correctness for message delivery. This is also used to obtain the inter-probabilities and intra-probabilities of the dynamic network models. We have implemented *Model A* and *Model B*; and use an experimental runtime verification tool RULER, developed by Barringer et al. [5]. RULER is a rule-based runtime verification with dynamic rules. This is an experimental use of runtime verification of concurrent models. Some trace properties required are properties properly of infinite traces. We show how to use runtime verification to examine finite traces and relate this to the overall network behaviour.

7.1 Verification of the dynamic network model simulation using RULER

Consider the following question. Suppose a dynamic network is implemented as a concurrent system using multiple Java threads. When do the network properties (expressed as properties of execution traces of concurrent systems as in Chapter 5) hold and, therefore, by the proofs in Chapter 5, when are all messages eventually delivered?

There are several approaches. We could prove the implementation manually or we could use a verification technique such as model checking. A typical approach is to generate the state space of the system and then apply a model-checking algorithm for it to determine whether the system satisfies given temporal logic formulas. This approach may lead to the state explosion problem, and concurrency is a major contributor to state explosion [60]. Here, dynamic networks are modelled as concurrent systems; therefore, there is a need to find another approach to avoid the state explosion when the network being observed increases in size. Moreover, we need a tool that can monitor not only the behaviour of systems as it terminate/stop but also at runtime. Using model checking, we can obtain a "yes" or "no" as the result. By verifying using runtime verification, we can also obtain the probabilities of each process to get the proper result (satisfying the properties of the model). Therefore, here runtime verification tool is used.

An approach is introduced based on runtime verification (RV) [39], as pictured in Figure 7.1. The two abstract models have been implemented as concurrent systems in a form of multi-threaded programs. Notice that by adapting the runtime verification system RULER[5], that used to monitor finite traces, we observe limiting finite traces and examine the behaviour when the traces increase in length. We have traces generated from a probabilistic machine (i.e., the multi-threaded programs). Probability becomes one when the traces get longer. The probability of the implementation model will not reflect the dynamic networks as in temporal logic model. For example, $\Box \Diamond \phi \Rightarrow \Diamond \psi$ is always satisfied in the finite traces. However, we need to know to what extend or whether the models are becoming fair in this finite sense. The role of runtime verification as a technique for reasoning about the implementation to check whether or not the properties required hold. Therefore, devise a runtime verification approach and then explore whether it may reflect the behaviour of dynamic networks.

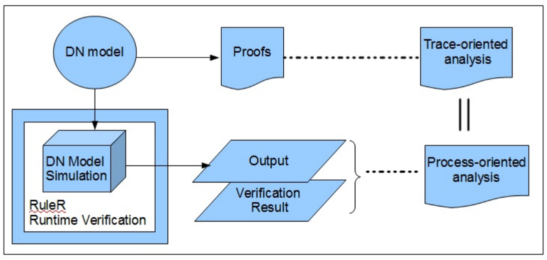


Figure 7.1: Runtime verification of the implementation of dynamic network models.

As Figure 7.1 shows, proofs of dynamic network models are obtained by proving the correctness of message-passing in such networks, as explained in Chapter 5. However, this proof does not show what type of actions are responsible for the changes of states. The model is then implemented into a multi-threaded program as a concurrent system. This implementation uses the experimental runtime verification RULER, developed by Barringer et al. [5]. By including a RULER specification to monitor output of traces from the program, this can be used to verify the properties of the dynamic network model that have been manually proved. Stream output as a result from the program execution verified by RULER can be regarded as 'proof' of process-oriented analysis.

Validation of the dynamic network model using runtime verification means we have to deal with the "transformation" of infinite traces into finite traces. The model uses the semantics of LTL which defines infinite traces while the RV monitors finite traces. Suppose we have trace $\tau \in \tau^{\infty}$ and ϕ is a property from model M. Using such an approach means we want to have $\mathsf{M} \models \phi$ iff $\forall\ \tau \in \tau^{\infty}(\mathsf{M}).\ \ \tau \models \phi$, based on finite traces $\tau|_n \models \phi$, with n increasing in size.

In general, there are two kinds of model:

1. *Enforced (built-in) model*, in which we initially set the parameter of the traces to have some properties. Therefore, if τ is a trace that enforced to satisfy the properties C1, C2 and C3, then it should be justified by runtime verification (RV) that the trace also satisfies the formula Φ. However, this is not interesting to explore any further because this means that we can only get what we already want to observe through setting whether there is a livelock.

2. *Check model*, in which we adjust the parameters and observe the traces. For example, check using the runtime verification, whether or not livelock possibly occurs within the traces. This means a trace τ is being checked using runtime verification whether it satisfies the properties C1, C2 and C3. Furthermore, the traces being checked by runtime verification whether the trace also satisfies the formula Φ.

We need to prove that all messages eventually reach their destinations.

$$\forall \mathsf{m}.\mathsf{Conditions} \rightarrow \Diamond \mathit{black}(\mathsf{m})$$

with Conditions for *Model A* are

- Prop1 : Paths exist sufficiently often
- Prop2 : Messages get attention sufficiently often

- Prop3 : Messages get attention sufficiently often and when they are accessed, they can be moved at some time (modified Prop2)
- Fair1 : finiteness of path
- Fair2 : no livelock (livelock-free)

and Conditions for *Model B* are

- Prop1 : Paths exist sufficiently often
- Prop4 : Messages get attention sufficiently often and paths for messages are obtained from the routing table.
- Prop5 : Messages get attention sufficiently often and paths for messages are obtained from the routing table, and when they are accessed, they can be moved at some time (modified Prop4)
- Prop6 : The routing table is populated sufficiently often (for *Model B*)
- Fair1 : finiteness of path
- Fair2 : no livelock (livelock-free)

Chapter 5 have discussed these properties. Here, the names of the properties are starting with “Prop” or “Fair” for easy to read purpose.

A trace is a sequence of states. Infinite traces are essential for reasoning about eventualities. For infinite traces of dynamic network models, the length of traces is greater than the observed output of the programs. When the traces are short, the relation between if and then should be weak. When the traces are longer, then the relation between if and then will be much confident. It is needed to move from infinite to finite traces and see when the traces are increasing in length. It is not about whether the finite traces are fair. It is about whether we think the prefix is fair.

A series of experiments for each *Model A* and *Model B* being performed in which different parameters are used as follows. We vary:

1. the probabilities that each process is accessing the network, i.e., the number of times that a process is allowed to do its job within the specified execution time. This is represented through sleep time for each process of *Model A* (as well as of *Model B*).

2. the length of run of the program. The length of the run could affect the length of the traces and the occurrences of required properties. The variations of the time used here are 30000, 60000, 120000, 180000, 240000, and 300000 μsecs.

3. density of initial graph. Density of network, consists of sparse network and dense network. The sparse network which used in here consist of six nodes and six edges. The dense network consists of six nodes and eight edges.

4. the number of messages in the network. This number of messages could affect the chance of a message being visited by the *Organizer* process. The higher number of messages within the same length of run (the same length of traces), the lower the chance of each message being looked at.

5. connectivity of original graph. Graph connectivity, is divided into "initially connected" and "arbitrarily connected". The graph parameter "initially connected" means that all edges within the graph initially have edge state *on*. The graph parameter "arbitrarily connected" means that initially the state of each edge may *on* or *off*, which determined randomly.

6. number of runs. A different number of runs should give a different accurate number or result.

7. Livelock-free: this parameter is implemented in two ways. Firstly, Enforced livelock-free model, in which model is being defined initially whether to produce a livelock-free traces or not. Secondly, *Check livelock model*, in which livelock occurrence is checked at runtime, whether the condition exist within the traces.

Whether or not a system satisfies the properties required for message delivery depends on inter-process interaction and the parameters involved in this. Runtime verification is particularly suitable here as it is the relationship of these parameters with the execution traces that determine the correctness of the dynamically allocated inter-process interaction.

Table 7.1 shows the average number of events which recorded by RULER for various execution time on the model instance *Disrupter* $\|$ *Organizer* = 1 $\|$ 1 and the *Disrupter* $\|$ *Organizer* $\|$ *Updater* = 1 $\|$ 1 $\|$ 1. We show how to use RV to examine finite traces and relate this to the overall network behaviour.

Time	**1 ∥ 1**	**1 ∥ 1 ∥ 1**
30000	585	6609
60000	970	12647
120000	1488	23440
180000	1895	31272
240000	2022	41479
300000	2794	49182

Table 7.1: The average number of events.

7.2 A brief explanation of using RuleR

Before we discuss *Model A* and *Model B* verification, here is a simple example of RuleR usage. RuleR is a Java-based program that implements the monitoring of systems using specifications presented as rule systems. It provides a simple Java interface to enable its direct use in other Java applications. This interface can be used from within an AspectJ instrumentation of a Java application.

The core of a RuleR rule system is a collection of named rules. The following example is taken from [5], and is about writing a monitor to check the validity of the responses given in answer to a simple arithmetical quiz. The system is to observe and monitor a sequence of question and answer events. The following temporal formula in Figure 7.2 expresses the constraint that every question is followed by a correct answer.

$$\begin{aligned}&\Box \forall x, y : int \cdot question(x,y) \Rightarrow \\ &\qquad \bigcirc((\neg \exists u, v, z : int \cdot answer(z) \vee question(u,v))\ \mathcal{U} answer(x+y))\end{aligned}$$

Figure 7.2: An example from [5]: the property required in terms of temporal logic.

The required behaviour is specified in terms of RuleR through the following rule system. The rule system, *SumCheck*, defines, using the **observes** keyword, that *question* is an observation event with two integer arguments and that *answer* takes a single integer argument. Two rules are specified with the keyword **state**; these indicate that the rules use state persistence (once the rule is active, it will remain active until it is successfully used). RuleR has two other persistence attributes for rules: *always persistence* (introduced by the keyword **always**) and *single shot persistence* (introduced by the keyword **step**).

The first rule in Figure 7.3 is *Check*. Its body comprises just a single rule (enclosed by parentheses), which will activate the *Response* rule as soon as a question observation occurs. When this happens, the state persistence means that the *Check*

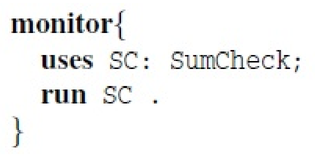
```
ruler SumCheck{
  observes question(int, int), answer(int);

  state Check{
    question(x:int, y:int) -> Response(x+y);
  }
  state Response(required:int){
    answer(z:int)
      {: z != required ->
           print("Wrong answer! Expected "
                    + required + " but given " + z),
           Check;
         default -> Check;
      :}
    question(x:int, y:int) ->
      print("Unexpected question! Previous one unanswered"),
      Response(required);
  }
  initials Check;
  forbidden Response;
}
```

Figure 7.3: An example from [5]: a RULER specification based on the required behaviour.

rule is deactivated. Otherwise, if a *question* observation does not occur, then the state persistence of *Check* will keep the rule active for the next monitoring step.

The second rule defines the *Response* rule. It is supplied with the integer value that expected to be given to the first occurrence of an answer that follows activation of the *Response* rule. As a factored rule, the rule has two parts; its precondition *answer(z:**int**)* has been factored out of the subsequent two sub-rules. The condition part of the first sub-rule checks for an invalid response. If the condition of the first sub-rule doesn't hold, then the next rule in the list is attempted. The rule has **default** keyword, meaning that it will always be able to be used. The second part of the *Response* rule handles an occurrence of an undesired *question* event. The keyword **initials** defines which rules are initially active. The keyword **forbidden** defines which rules are not allowed to be active at the end of monitoring a finite sequence of observations.

Afterwards, we should create a monitor based on the *SumCheck* rule system, as shown in Figure 7.4. Therefore, the rule system *SumCheck* and the monitor definition are the specification input to the RULER system. The **monitor** definition creates an instance of the *SumCheck* rule system, namely SC, and then runs it. An interesting



```
monitor{
  uses SC: SumCheck;
  run SC .
}
```

Figure 7.4: An example from [5]: a monitor based on the *SumCheck* rule system.

experimental feature of RULER is its monitor combination capability, namely *chaining*. Monitor chaining is used in the following experiments when calculating the total event/percentage of each process in association with the model properties, also when checking the message eventually green (see Appendix B.3). Refer to [5] for further tutorial of RULER.

7.3 Experiments of *Model A*

For *Model A* (the Instantaneous Update model), to validate model means

$$\text{if } \mathcal{M} \models \text{Prop1} \wedge \text{Prop2} \wedge \text{Prop3} \wedge \text{Fair1} \wedge \text{Fair2} \text{ then } \mathcal{M} \models \forall \mathsf{m}. \Diamond \mathit{black}(\mathsf{m})$$

with the details of properties as follows.

1. Paths exist infinitely often:

   Prop1: $\forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\Diamond\ \mathit{pathX}(\mathsf{n}_1, \mathsf{n}_2)$

2. Red messages eventually become green (messages are looked at sufficiently often):

   Prop2: $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((\Box\Diamond\ \mathit{pathX}(\mathit{at}(\mathsf{m}), \mathit{dest}(\mathsf{m})))\ \wedge\ \mathit{red}(\mathsf{m})$
   $\Rightarrow\ \Diamond\ (\mathit{green}(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}), \mathit{dest}(\mathsf{m})).\ \mathit{path}(\mathsf{m}, \mathsf{p}))))$

3. The *Organizer* should access each message sufficiently often, and when it is

accessed, it can be moved.

Prop3: $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((\Box\Diamond\ pathX(at(\mathsf{m}), dest(\mathsf{m}))) \Rightarrow$
$\Diamond\ (black(\mathsf{m})\ \wedge$
$(red(\mathsf{m})\ \Rightarrow\ \bigcirc(green(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m},\mathsf{p}))))$
$\wedge$
$(\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ (green(\mathsf{m})\ \wedge\ path(\mathsf{m},\mathsf{p})\ \wedge\ at(\mathsf{m}) \neq dest(\mathsf{m})$
$\Rightarrow$
$(up(1st\_e(\mathsf{p}))\ \wedge\ \bigcirc(green(\mathsf{m})\ \wedge\ path(\mathsf{m}, tl(\mathsf{p}))$
$\wedge\ at(\mathsf{m}) = next\_node(\mathsf{p})))))\ \wedge\ ((green(\mathsf{m})\ \wedge\ at(\mathsf{m}) = dest(\mathsf{m}))\ \Rightarrow$
$\bigcirc\ black(\mathsf{m}))))$

4. **Finiteness of paths**, which means that if a message m is green and there is no potential for it to become red eventually then message m will eventually become black . This can only hold if the *Organizer* checks message m infinitely often (P2). We define this formally as follows.

Fair1: $\forall \mathsf{m} \in \mathsf{M}.\ ((green(\mathsf{m})\ \wedge\ \neg\Diamond\ red(\mathsf{m}))\ \Rightarrow\ (\Diamond\ black(\mathsf{m})))$

5. Fair2 (Livelock-free), as defined in Chapter 5.

Fair2: $\forall \mathsf{m} \in \mathsf{M}.\ \neg\ \Box\Diamond\ (green(\mathsf{m})\ \Rightarrow\ \bigcirc red(\mathsf{m}))$

7.3.1 RULER specification for *Model A*

To trace the number of messages which reach the destination, check whether there is any livelock potential, and the path and occurrence of disruption, a RULER monitor specification as below is defined, and then run with the Java program. *Model A* (the Instantaneous Update model) uses the specification as listed in Appendix B.1.

The specification uses monitor chain features as follows. Two monitors are defined: *PathXCountTrace* and *Calculate*. The first will capture the number of events from the two processes and send the sub total during runtime to a second monitor. The output from *PathXCountTrace* (i.e., totEv, pOut, pOut2, pOut3, pOut4, and pOut5) becomes the input for *Calculate*. This second monitor will calculate the percentage of each sub total. These percentages and sub total numbers are associated with the Prop1, Prop2, and Prop3 properties in *Model A*.

```
monitor{
  uses PXC: PathXCountTrace, C: Calculate;
  locals tEv(int), pgb(int), ppath(int), pall(int), porg(int),
        pdis(int), c0(int), c02(int), c03(int), c04(int), c05(int);
  run (PXC(tEv, pgb, ppath, pall, porg, pdis) >> C(tEv, pgb, ppath,
      pall, porg, pdis, c0, c02, c03, c04, c05)) .
}
```

Initially, all variables which will record the number of occurrences are set to zero. Moreover, the monitoring starts from state Red (which will always be activated) and identifies the occurrence of Black (liveness properties).

```
    initials Red, GBt(0), DisT(0), PathT(0), OrgT(0), AllOrT(0),
          GRt(0), RedGt(0), TotC(0);
    forbidden Black;
    outputs totEv, pOut, pOut2, pOut3, pOut4, pOut5;
```

The variables are explained as follows.

- DisT: records the total number of *Disrupter* actions during runtime, based on the *Disrupter* actions (DTC and CTD)

```
 state DisT(DisCount:int) {
   DTC(g:obj, a:int, b:int) -> DisT(DisCount+1),
   print("DisT = " + (DisCount+1)), pOut5(DisCount+1);
   CTD(g:obj, a:int, b:int) -> DisT(DisCount+1),
   print("DisT = " + (DisCount+1)), pOut5(DisCount+1);
 }
```

  Here the two actions (CTD and DTC) are regarded as one action because the selection of the disruptive action is defined (if a randomly chosen edge is connected at the moment, then next time it should be disconnected, and vice versa). Therefore, here the intra-process probabilities has equal value for each action within the *Disrupter*. Another form of disruption is to let the system decide randomly at runtime which disruption is selected as the next occurrence, which allows us to obtain the data for intra-process probabilities. The pOut5 observer records total events of DTC and CTD action.

- PathT: records the number of path existence in *Model A* during runtime, based on the finding of the available route by *pathFinder* process (which runs every time after a disruption occurs, to obtain an instantaneous update). The result of the calculation for this number by (*Calculate*), namely PathX, is associated with properties Prop1 of *Model A*.

```
  state PathT(PathXCount:int) {
  write_Path(p:obj) -> PathT(PathXCount+1),
  print("PathXCount = " + (PathXCount+1)), pOut2(PathXCount+1);
}
```

- The variables for recording the actions of the *Organizer* are grouped as follows:

  1. AllOrg: records the total number of *Organizer* action during runtime, based on all the actions of *Organizer* process (RG, GR, wait and move) except the GB action. The result of the calculation of this number by (*Calculate*), namely %AllOrg, is associated with properties Prop2 of *Model A*.

```
 state AllOrT(AllTCount:int) {
   RGr(m:obj, idx:int, p:string, ps:int) -> AllOrT(AllTCount+1),
     print("AllTCount = " + (AllTCount+1)), pOut3(AllTCount+1);
   move(m:obj, idx:int, p:int) -> AllOrT(AllTCount+1),
     print("AllTCount = " + (AllTCount+1)), pOut3(AllTCount+1);
   GR(m:obj, idx:int) -> AllOrT(AllTCount+1),
     print("AllTCount = " + (AllTCount+1)), pOut3(AllTCount+1);
   waitM(m:obj, idx:int) -> AllOrT(AllTCount+1),
     print("AllTCount = " + (AllTCount+1)), pOut3(AllTCount+1);
 }
```

  2. OrgT: records the number of message becomes green and message move action during runtime, based on the action RG and move. The result of the calculation for this number, namely %OrgT, is associated with properties Prop3 of *Model A*.

```
 state OrgT(OrgTCount:int) {
   RGr(m:obj, idx:int, p:string, ps:int) -> OrgT(OrgTCount+1),
     print("OrgTCount = " + (OrgTCount+1)), pOut4(OrgTCount+1);
   move(m:obj, idx:int, p:int) -> OrgT(OrgTCount+1),
     print("OrgTCount = " + (OrgTCount+1)), pOut4(OrgTCount+1);
 }
```

  3. GRt : records the number of message becomes red during runtime, based on GR action which is belong to the *Organizer*.

```
state GRt(GRCount:int){
  GR(m:obj, idx:int) -> GRt(GRCount+1), print("GRt = " + (GRCount+1)); }
}
```

  4. RedGt : records the number of messages becomes green during runtime, based on RG action which is belong to the *Organizer*.

```
state RedGt(RedGCount:int){
  RGr(m:obj, idx:int, p:string, pos:int) -> RedGt(RedGCount+1),
                                  print("RedGt = " + (RedGCount+1)); }
```

5. GBt : records the number of message becoming black during runtime, based on GB action which belongs to the *Organizer*. The result of the calculation of this number is associated with properties $\Diamond$ *black*(m) of *Model A*.

```
state GBt(GBcount:int) {
  GB(m:obj, idx:int) -> GBt(GBcount+1),
     print("GB = " + (GBcount+1)), pOut(GBcount+1);
}
```

Notice that verification in this experiment is much more as stream processing rather than pure runtime verification.

The following is the state of monitor *Calculate* which calculates the total number/percentage.

```
state CountA(cct:int, cct2:int, cct3:int, cct4:int, cct5:int, ttc:int) {
   pIn(ct:int), tIn(tc:int) -> CountA(ct, cct2, cct3, cct4, cct5, tc),
      print("%EB = " + (ct/60.0)*100 + ", PathX = " + cct2 + ",
      %AllOrg = " + (cct3/(tc*1.0))*100 + ", %OrgT = " + (cct4/(tc*1.0))*100 + ",
      %DisT = " + (cct5/(tc*1.0))*100 + ", tc = " + tc) ;
... (see Appendix B.1 for the details)
}
```

The percentage number of messages that are eventually black, namely %EB, is obtained from the number of messages that are eventually black within the execution time, relative to the number of messages within the network (in this case, 60). The number of available paths (which is needed by the messages) existing within the execution time is calculated in total during runtime as PathX. %AllOrg, %OrgT, and %DisT are the percentage of AllOrg, OrgT, and DisT, respectively, relative to the total event *tc* within the execution time.

For checking the livelock, the RULER monitor specification is as listed in Appendix B.1. This checking is associated with properties Fair2 of *Model A*.

```
always Red{
  RoutingUpdate(id:int, s:int, x:string, d:int, y:obj, z:obj, p:string) ->
                           redG(id, 1, p), print("message m " + id + "path = " + p);
  RGr(m:obj, idx:int, p:string, ps:int) -> Black(m, idx);
}
```

Here, at the occurrence of RoutingUpdate action, the state redG will be activated. This state will detect whether a particular message is in the position where livelock may occur (i.e., on a node within a circular path). For example, in the following rule *redG*, if the event RG(m) occurs for the same message id and the message is allocated the same path fewer than six times then the rule activates the redG only (and record the occurrence of a particular path for particular message).

However, suppose that the event RG(m) occurs for the same message id and the message is allocated the same path fewer than six times, and its position is within a circular path (here, denoted by “pos==2”). Therefore, the rule redG is activated and tell that there is a potential livelock when a message m has been allocated a particular path more than six times. This number is the average number of a message has potential livelock within 60000 μsecs execution time, which obtained in the Enforced livelock model.

```
state redG(id:int, ctp:int, pm:string) {
 RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 <6 -> redG(id, ctp+1, pm);
 RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 >=6, pos==2 ->
     redG(id, ctp+1, pm), print("m" + idm + " potential livelock,
     path " + rp + " " + (ctp+1) + " times");
 ... (see Appendix B.1 for the details)
}
```

The important thing is how to determine the livelock pattern. At a certain point, there is a repeat in pattern but we do not know how far it will be until it occurs again. This problem is almost like the how to determine a decimal as a common fraction (a rational number) problem (e.g.: 1.23234...). A decimal number is called a repeating decimal if at some point it becomes periodic, which represents a rational number. Moreover, the problem is also how to determine the sequence of number **is** a repeating decimal.

The obvious thing to do is to adjust the parameter. We should just look at the particular message and determine the behaviour. In general, we are looking for the pattern of behaviour. The result might be a *false positive*, where at certain point there is a livelock, but after hundreds of time there could be no more. Otherwise, the result might be a *false negative*, in which it looks as if there is no livelock within the traces, but after some times there is a livelock. This pattern determination problem might be regarded as part of *specification inference/specification mining* [40], in which look at the event then check whether there is the same pattern, and some works [53] with a focus on specification mining for API property inference.

7.3.2 Results for *Model A*

The results of these experiments are presented based on the parameters mentioned in Section 7.1. Firstly, through the *Experiment 1*, we can check whether the traces of the program satisfies the properties of *Model A*. Moreover, if the properties of *Model A* are satisfied by the traces, then we expect that the results of the *Experiment 1* will show whether all the messages are eventually delivered (whether red(m) eventually black (m)).

After we know the possibilities of the messages are eventually delivered, now we suspect that if the length of the traces increase, then the chance for the messages eventually delivered will be higher. Therefore, the *Experiment 2* is being set out to check the traces using various execution times. Furthermore, the impact of livelock potential occurrence of the number of messages eventually delivered is explored through *Experiment 1* and *Experiment 2*.

Both *Experiment 1* and *Experiment 2* use the same density of the graph. Here, the dynamic changes of dynamic network models are established from the changes of edge status over time. Therefore, the *Experiment 3* is performed to learn whether the density of the graph will affect the number of message becoming black/delivered. Particularly, we can know through the Experiment 3, whether the density of the graph affects the path availability within the network; and moreover, whether the density of the graph affects the percentage of messages that are delivered.

Properties of *Model A* shows that the messages should get visited sufficiently often to become eventually green (moreover, eventually black). *The Experiment 4* is performed to know whether the chance of the messages to become eventually green/black also will be affected by the various numbers of the messages within the network.

The initial conditions for the program itself may affect the result. Therefore, the *Experiment 5* explore the possibility of initial conditions effect, particularly using the different initial graph. The number of runs also can affect the results of the program, therefore the *Experiment 6* is carried out to show this.

7.3.2.1 Experiment 1: probabilities between the two processes

Aim: Experiment 1 aims to observe whether the traces satisfy properties Prop1, Prop2, Prop3, Fair1, and Fair2 if we set the probabilities of the two processes (*Disrupter* and *Organizer*) accessing the network. Furthermore, it aims to observe whether all messages are eventually delivered.

Setting: The following parameters are fixed. Using *Check model* for *Model A*, the number of messages in the network is 60, the execution time is 60000 μsecs, and run on a sparse network. The graph is initially connected, and at the start of the program, all edges have the status *on*. We consider 17 different settings of the probabilities of process access to the network from 256 || 1 (*Disrupter* sleeps 256 times longer than

Organizer), through to 1 || 256 (*Organizer* sleeps 256 times longer than *Disrupter*). Each model instance has 10 runs of the program. Events from the run of each model instance are recorded in the trace log files. The information from 10 runs are calculated as the average and become a point in the result chart.

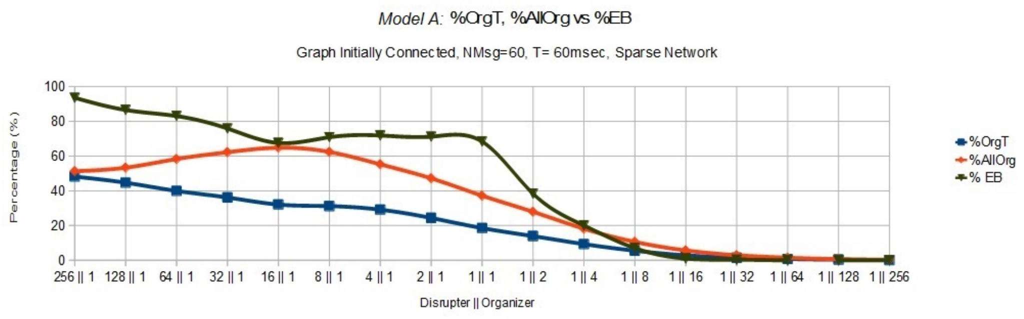


Figure 7.5: Check model of *Model A* with number of messages = 60, Time = 60000 μsecs, with graph initially connected

Results: The x-axes in Figure 7.5 denote the probabilities of the two processes (*Disrupter* and *Organizer*) in accessing the network. On the y-axes, we plot three different properties of the executions (an average over the 10 runs). These are: %AllOrg (percentage of *Organizer* events within the traces (how many times the messages are visited)), which associated with Prop2 properties); %OrgT (percentage of how many times messages eventually moved within the traces), and %EB (the number of messages $\Diamond$ black (m)).

The curves of %EB in Figure 7.5 show the trend in the number of messages that are getting delivered. The curves show that the number of messages delivered is at highest peak when the *Disrupter* sleeps 256 times more than *Organizer*. Afterwards, the number of messages delivered decreases towards zero. Notice that there is a sharp decrease after probabilities between *Disrupter* and *Organizer* = 1 || 1. This is caused by *AllOrg* as the number of messages visited by the *Organizer* (Prop2) and *OrgT* as the number of messages that eventually move (Prop3) sharply drop after the 1 || 1 point, as shown in Figure 7.6. This fact from Figure 7.5 and Figure 7.6 shows that Prop2 and Prop3 are a necessary condition for the messages to eventually become black.

Figure 7.5 also shows that when the *Disrupter* is less disruptive, then the number of %AllOrg and %OrgT are high, as is the number of %EB. The interesting fact in Figure 7.5 is that the curve of %EB decreases when the *Organizer* is less active. However, at the point where *Disrupter* || *Organizer* = 16 || 1, the number of %EB slightly drops for a while then increases again after that point and follows the trend of the %EB's curve.

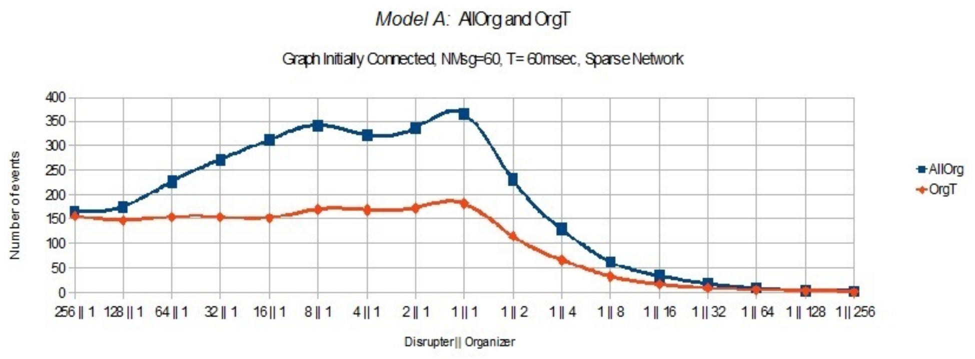


Figure 7.6: AllOrg and OrgT (associated with Prop2 and Prop3 respectively) of *Model A*.

This occurrence is related to the properties of Prop1 and Fair2 (captured by *PathX* and *potential livelock* respectively, using RULER).

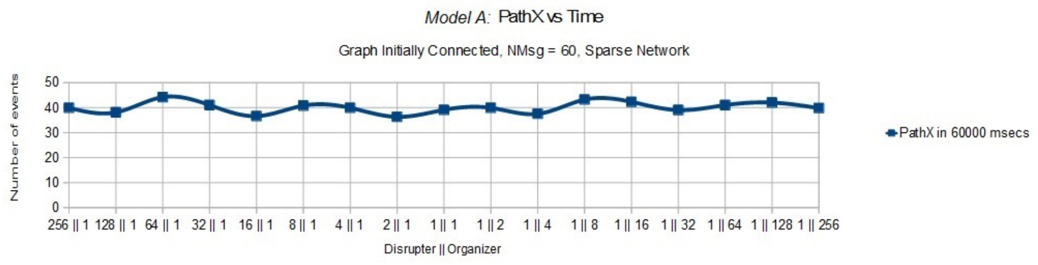


Figure 7.7: PathX of *Model A* with number of messages = 60, Time = 60000 μsecs, with graph initially connected

The curves of PathX in Figure 7.7 show the trend of path existence within the network with the above setting. The x-axes in Figure 7.7 denote the probabilities of two processes (*Disrupter* and *Organizer*) in accessing the network. The y-axes of Figure 7.7 plot the PathX in the number of events unit. The number of PathX for each probabilities between the two process is within the range of 35 to 45 events.

This pattern shows that the number of PathX exhibits no significant changes between different probabilities (we will focus on PathX in more detail in a discussion section (Section 7.6) of this chapter). Moreover, this pattern shows that Prop1 is satisfied by each different probability between the two processes. Therefore, according to *Model A* properties, if a path exists infinitely often and the message is visited sufficiently often, then the number of %EB should not drop slightly at the point where *Disrupter* || *Organizer* = 16 || 1. The possible properties that can prevent this drop is livelock-free Fair2.

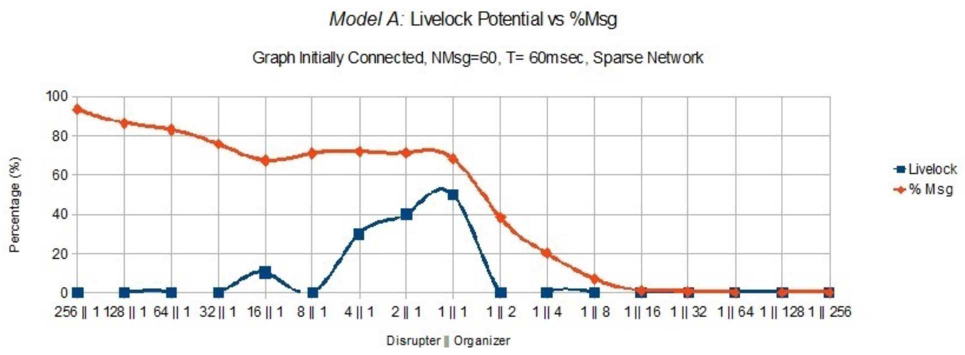


Figure 7.8: Potential livelock of *Model A*, with graph initially connected

Figure 7.8 shows the potential livelock on *Model A* vs. %EB, where the x-axes denotes the probabilities of two processes (*Disrupter* and *Organizer*) in accessing the network. On the y-axes, we plot Fair2 properties as the potential livelock relative to different model instances and %EB as the number of messages that eventually turn black.

The curve in Figure 7.8 shows that at the point where *Disrupter* || *Organizer* = 16 || 1, there is a potential livelock (about 10%). This potential livelock is responsible for the slower rate of %EB. Notice that the potential livelock for running time = 60000 μsecs higher at the points where *Disrupter* || *Organizer* = 4 || 1, 2 || 1 and 1 || 1 but the %EB for these points remain stable. One of the reasons is that although the potential livelock is high at these points, the number of messages being caught by the livelock is still lower than at the point where *Disrupter* || *Organizer* = 16 || 1. For example, at the point where *Disrupter* || *Organizer* = 16 || 1, there is one message being caught by the potential of livelock and this message not eventually black at the end of the program execution. On the contrary, at the point where *Disrupter* || *Organizer* = 1 || 1, there are 50% potential livelock and two third of the messages being caught by livelock are eventually black.

If we compare variations of time, then the plot of PathX is shown in Figure 7.9, with x-axes denoting the execution/running time and y-axes plotting the PathX over time (*Disrupter* || *Organizer* = 1 || 1). The trend of PathX decreases when the time becomes longer, since potential livelock of the long running time is higher, as shown in Figure 7.14 later in Section 7.3.2.2. However, at some point (i.e., where Time = 1000000 μsecs), PathX slightly increases to the same level when Time = 30000 μsecs. Afterwards, the curve slightly drops to the same level with Time = 120000 μsecs. Therefore, the pattern of PathX has been always within the particular range.

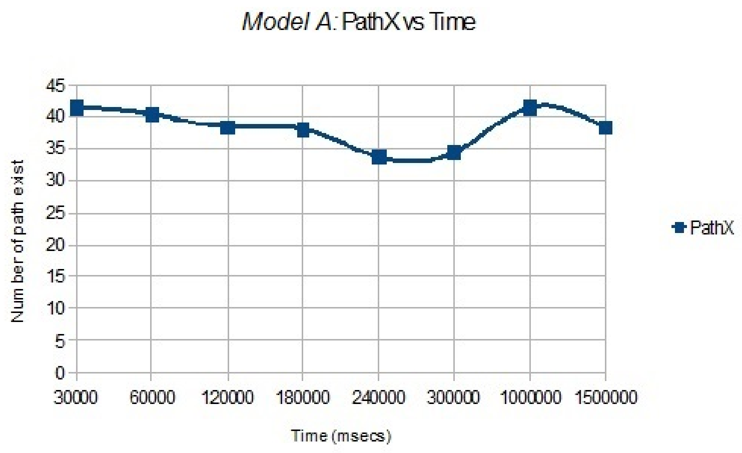


Figure 7.9: PathX of *Model A* vs. Time, with graph initially connected

Conclusion: From the results above, we can conclude as follows.

- The result shows that: if path existence occurs infinitely often and the messages are looked at sufficiently often, then the number of messages that eventually reach their destinations increased. This result supports the proof of Lemma 1, 2 and 3, hence the Theorem that been discussed in Chapter 5.
- Potential livelock occurrence slows the rate of messages eventually black.
- The number of messages being caught and affected by potential livelock influences the number of messages eventually black within the execution time.
- The properties Prop1 (i.e. the path exists infinitely often) is sufficient but not necessary for message eventually black, whether the path existences are related to the messages or not.
- The properties Prop2 and Prop3 are sufficient and necessary for message eventually black.

7.3.2.2 Experiment 2: variation of execution time

Aim: Experiment 2 aims to observe whether the traces satisfy properties Prop1, Prop2, Prop3, Fair1, and Fair2 when the programs run for various execution times, in a model instance. Furthermore, it aims to observe whether all messages are eventually delivered.

Setting: The following parameters are fixed. Using *Check model* for *Model A*, the number of messages that will be delivered is 60, and they run on a sparse network. The graph is initially connected, which means that at the start of the program, all edges have the status *on*. The model instance that used here is 1 || 1 combination, therefore the *Disrupter* and the *Organizer* have the same chance of being sleep and awake. This experiment runs on the variation of execution time for the model instance 1 || 1(30000, 60000, 120000, 240000, 300000, 1000000, and 1500000 μsecs), and repeat for three times. Events from each run are recorded in the trace log files. These information from three runs are calculated as the average and becomes a point in the result chart.

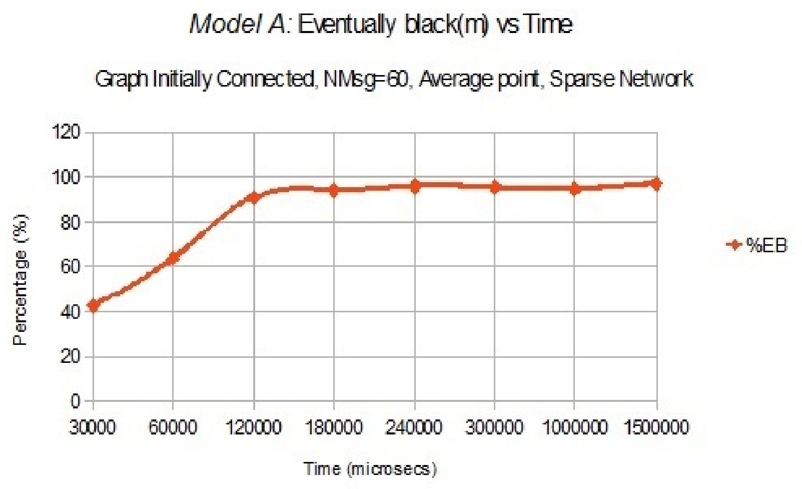


Figure 7.10: *Model A*: Eventually black(m) vs. Time.

Result: Figure 7.10 shows the number of messages that eventually turn black relative to the different execution time. The x-axes denote various execution time, while on the y-axes we plot the curve for %EB over time. The %EB increases sharply from the point where Time = 60000 μsecs to the point where Time = 120000 μsecs. Afterwards, the %EB only slightly increases, which is related to the *Organizer*'s activity and potential livelock over time.

Figure 7.11 shows the AllOrg and OrgT relative to the variation of execution time. The x-axes denote the variation of time while the y-axes plot the two properties of execution. The number of messages visited (AllOrg) increases sharply over time, even after the point where Time = 120000 μsecs. However, the number of messages that eventually turns green (OrgT) only slightly increases after that point. Although the number of AllOrg and OrgT tends to increase when the execution time is longer, the potential livelock also increases starting from the point where Time = 60000 μsecs

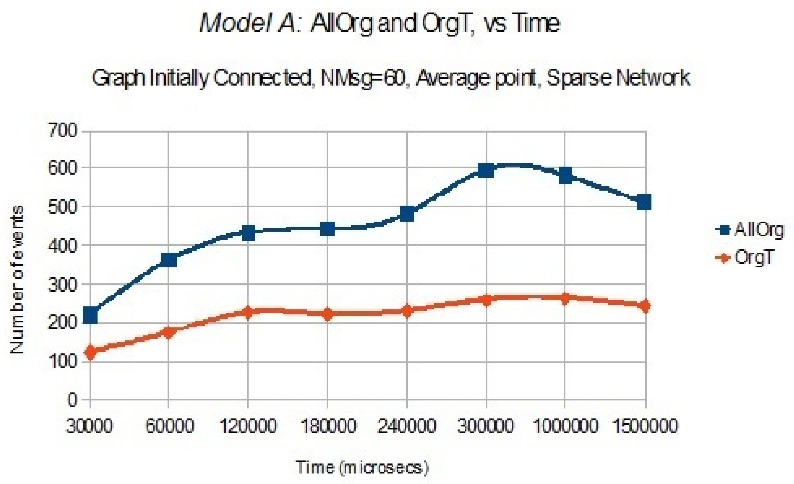


Figure 7.11: *Model A*: AllOrg and OrgT vs. Time

to the point where Time = 120000 μsecs. Figure 7.12 shows this trend, with x-axes denoting Time and y-axes plotting the potential livelock over increasing Time.

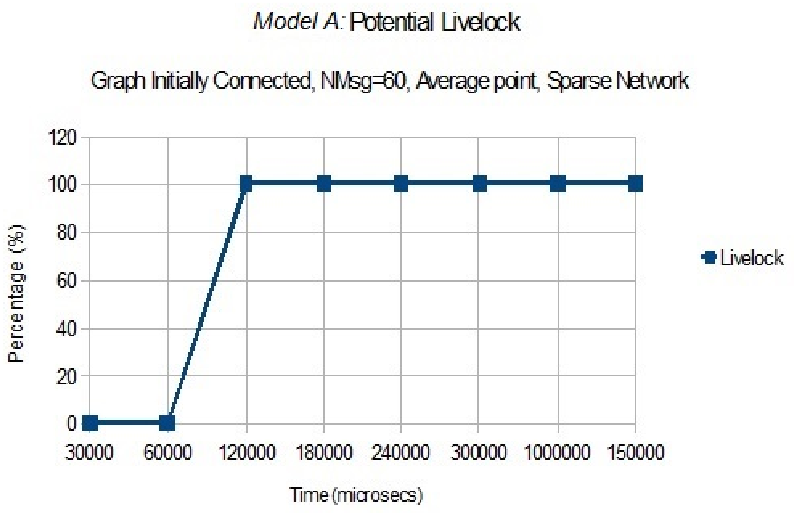


Figure 7.12: *Model A*: Potential livelock vs Time

Figure 7.13 shows the comparison of %EB from Time = 60000 μsecs, Time = 180000 μsecs, and Time = 240000 μsecs relative to probabilities between the two processes. The x-axes in Figure 7.13 denote the probabilities between the two processes in accessing the networks and y-axes plot three properties of execution from three variations of time. The curves show that there exists the same pattern of less *Disrupter* activity to less *Organizer* activity, and the longer the traces the higher the number of %EB.

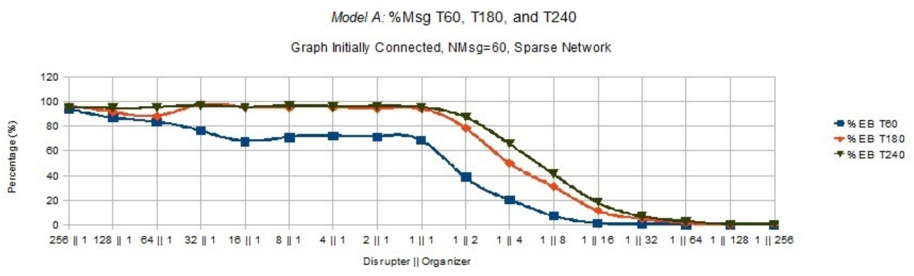


Figure 7.13: Dynamic network model of *Model A* with Time = 60000 μsecs, Time = 180000 μsecs, and Time = 240000 μsecs.

Figure 7.14 shows the comparison of livelock relative to three variations of time, with the x-axes denote the probabilities of the two processes (*Disrupter* and *Organizer*) in accessing the network. On y-axes, we plot Fair2 properties as the result of three different lengths of run: 60000, 180000, and 240000 μsecs (denotes by T60, T180, and T240 respectively). The curves show the similar pattern of potential livelock.

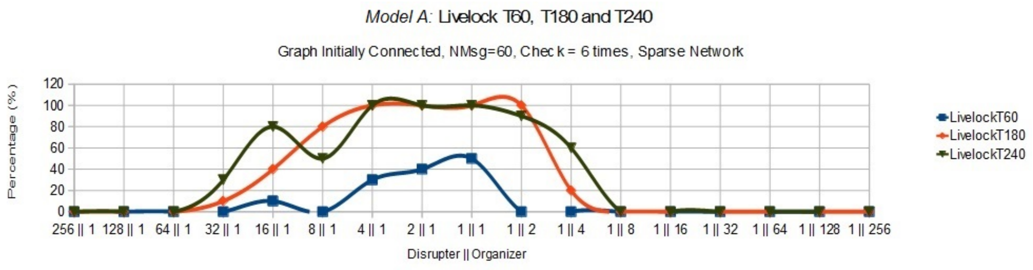


Figure 7.14: Potential Livelock of *Model A* vs. Time, with graph initially connected.

Conclusion: Using this model, there is no need for dynamic changes to cease to obtain the liveness property (i.e., messages eventually delivered). This is shown by the longer the traces, the higher number of messages being delivered in combination 1 || 1.

7.3.2.3 Experiment 3: density of initial graph

Aim: Experiment 3 aims to observe whether the traces satisfy properties Prop1, Prop2, Prop3, Fair1, and Fair2 when the programs are run on different density of the initial graph. Furthermore, it aims to observe whether all messages are eventually delivered.
Setting: The following parameters are fixed. Using *Check model* for *Model A*, the number of messages that are being delivered is 60, and the graph is initially connected

(i.e., the start of the programs, all edges have the status *on*). We consider 17 different settings of the probabilities of process access to the network from 256 || 1 (*Disrupter* sleeps 256 times longer than *Organizer*), through to 1 || 256 (*Organizer* sleeps 256 times longer than *Disrupter*). Each model instance has 10 runs of program, and are run first on a sparse network, then on a dense network. Events from the run of each model instance are recorded in the trace log files. The information from 10 runs are calculated as the average and become a point in the result chart.

Results: In Figure 7.15, x-axes denote the probabilities between the two processes and y-axes plot two properties of execution with variation of the density of the graph. Figure 7.15 shows that there is no significant different number of %EB if we use different densities of the graph. Furthermore, if we have the longer traces as in Figure 7.16 where time = 180000 μsecs, we still obtain a similar pattern of curves.

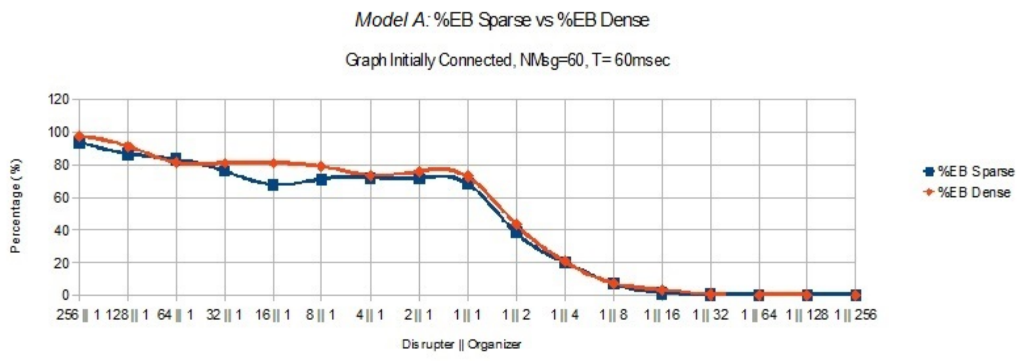


Figure 7.15: *Model A*: Number of messages that are eventually black on sparse network vs. dense network.

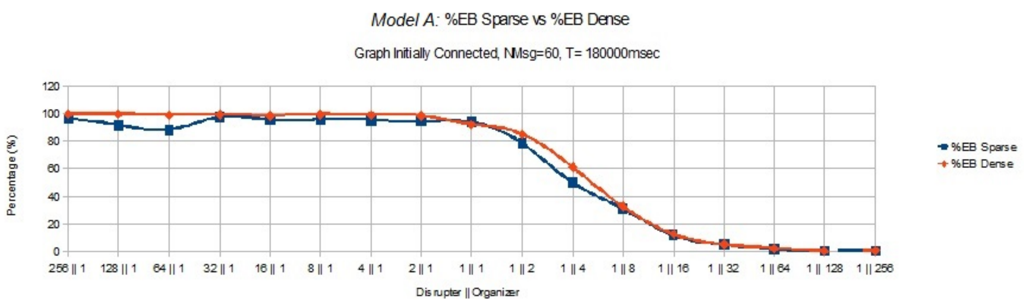


Figure 7.16: *Model A*: Number of messages that are eventually black on sparse network vs. dense network, with time = 180000 μsecs.

The density of a graph also does not affect the potential of livelock. Figure 7.17

shows this, with x-axes denoting the probabilities between the two processes in accessing the networks and y-axes plotting the livelock potential for different densities of the graph. Furthermore, Figure 7.18 shows a similar pattern in the longer traces.

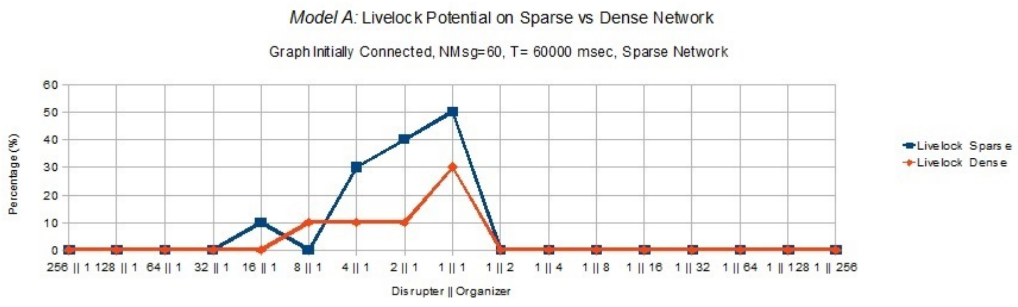


Figure 7.17: *Model A*: The livelock potential on sparse network vs. dense network, with time = 60000 μsecs.

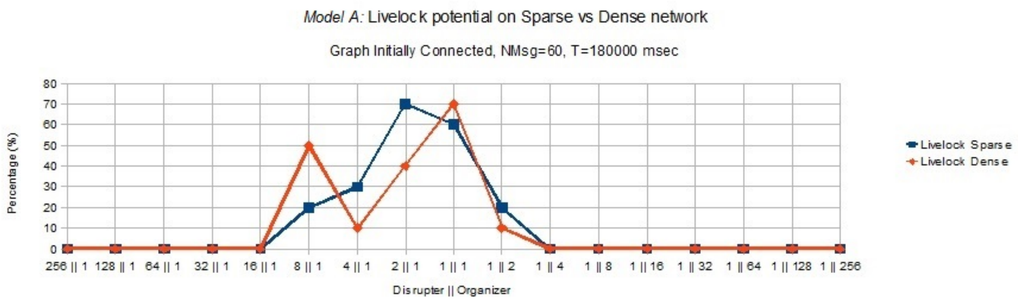


Figure 7.18: *Model A*: The livelock potential on sparse network vs. dense network, with time = 180000 μsecs.

Conclusion: For *Model A*, there is no relation between the density of the graph with the number of messages eventually turning black. Moreover, the potential of livelock shows a similar pattern for both sparse and dense networks over time.

7.3.2.4 Experiment 4: the number of messages in network

Aim: Experiment 4 aims to observe whether the traces satisfy properties Prop1, Prop2, Prop3, Fair1, and Fair2 when the programs run for different number of messages in the network. Furthermore, it aims to observe whether all messages are eventually delivered.

Setting: The following parameters are fixed. Using the *Check model* for *Model A*, the graph is initially connected (i.e., at the start of programs, all edges have the status *on*) and run on a sparse network. The model instance that used here is a probability of 1 || 1, therefore the *Disrupter* and *Organizer* have the same chance of being asleep and awake, with execution time = 60000 μsecs. This experiment run with various numbers of messages in the network, i.e., 10, 20, 30, 40, 50, 60, 70, and 80 messages, and is repeated for three times. Events from each run are recorded in the trace log files. The information from three runs are calculated as the average and become a point in the result chart.

Results:

In Figure 7.19, the x-axes denote the number of messages in the networks, in-

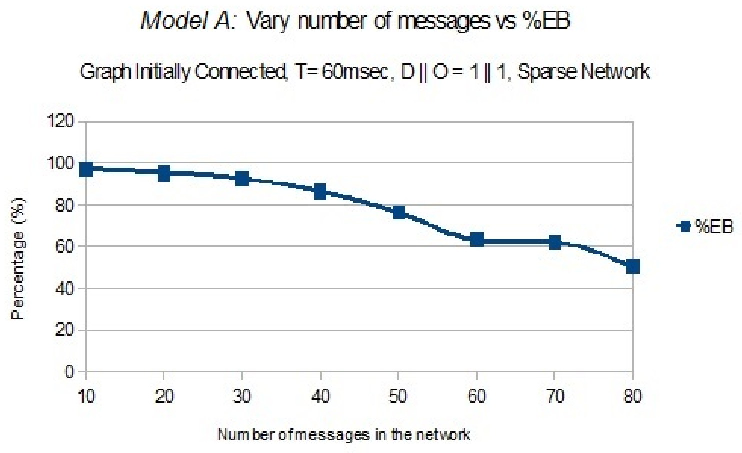


Figure 7.19: *Model A*: Number of messages in the network vs. %EB.

creased from 10 to 80 and the y-axes plot the properties of execution in which messages are eventually black. Figure 7.19 shows that when the number of messages is increasing in size, then the chance of all messages eventually being black decreased. This is because the number of messages affected the chance of the *Organizer* visiting all the messages. Figure 7.20 shows this trend.

Conclusion: The number of messages within the network influences the chance of the messages being visited by the *Organizer* process. Furthermore, it affects the number of messages that eventually turn black within the execution time.

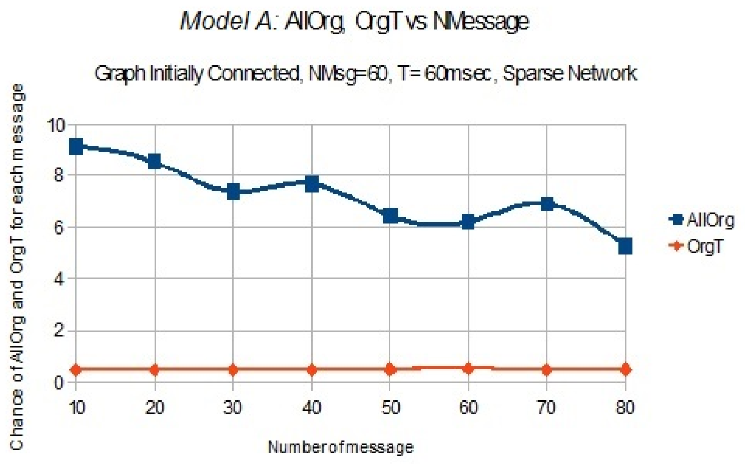


Figure 7.20: *Model A*: %EB vs. Number of messages in the network.

7.3.2.5 Experiment 5: connectivity of original graph

Aim: Experiment 5 aims to observe whether the traces satisfy properties Prop1, Prop2, Prop3, Fair1, and Fair2 when the programs run on different connectivities of the original graph. Furthermore, it aims to observe whether all messages are eventually delivered.

Setting: The following parameters are fixed. Using *Check model* for *Model A*, the number of messages that are being delivered is 60, and run on a sparse network. We consider 17 different settings of the probabilities of process access to the network from 256 || 1 (*Disrupter* sleeps 256 times longer than *Organizer*), through to 1 || 256 (*Organizer* sleeps 256 times longer than *Disrupter*). Each model instance has 10 runs of the program, and is run firstly on an initially connected graph, then on an arbitrarily graph. Events from the run of each model instance are recorded in the trace log files. The information from 10 runs are calculated as the average and become a point in the results chart.

Results: The x-axes in Figure 7.21 denotes the probabilities of two processes (*Disrupter* and *Organizer*) in accessing the network. On the y-axes, we plot two different properties of the executions (average over the 10 runs). These are: *%EB Init* (the number of messages $\Diamond$ black (m)) on the sparse network with initial graph connected and *%EB Arbit* on the sparse network with initial graph arbitrarily connected.

Figure 7.21 shows that when the *Disrupter* is less disruptive, then the number of %EB Init are nearly 100 % at point 256 || 1 and about 70 % at point 1 || 1. However, the number of %EB Arbit at the same points are always between the range 20 to 40 %,

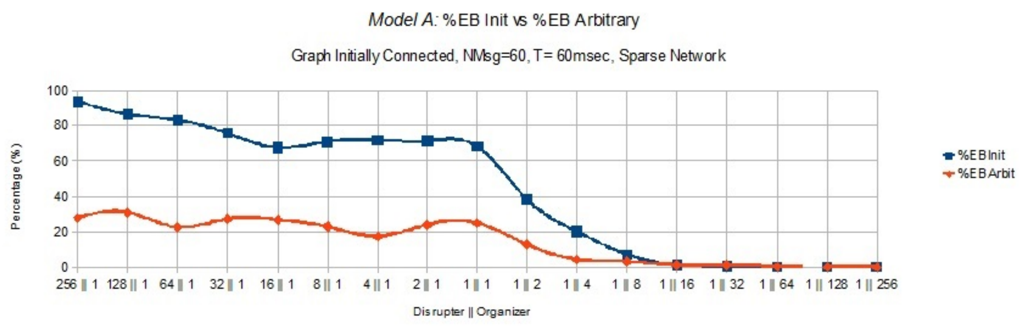


Figure 7.21: *Model A*: Number of messages in the network with initially connected graph vs. arbitrary connected.

and therefore lower than %EB Init.

The interesting fact in Figure 7.21 is the curve of %EBInit and %EB Arbit decreases when the *Organizer* is less active. However, at the point where *Disrupter* || *Organizer* = 16 || 1, the number of %EBInit slightly drops for a while then increase again after that point and follows the trend. Similar pattern occurs on the curve of %EBArbit, at the point where *Disrupter* || *Organizer* = 4 || 1. This occurrence related with the properties of Prop1 and Fair2 (captured by PathX and potential livelock respectively using RULER).

Conclusion: Using different initial graph (initial graph connected and initial graph arbitrarily connected), the results shows that, although there is a similar pattern (heading towards zero), the number of messages eventually black/delivered is affected by this initial condition (%EBInit higher than %EBArbit).

7.3.2.6 Experiment 6: number of runs

Aim: Experiment 6 aims to observe whether the traces satisfy properties Prop1, Prop2, Prop3, Fair1, and Fair2 when the programs run on different numbers of runs, in a model instance. Furthermore, it aims to observe whether all messages are eventually delivered.

Setting: The following parameters are fixed. Using *Check model* for *Model A*, the graph initially connected (i.e., at the start of programs, all edges have the status *on*) and run on a sparse network. The model instance that used here has a 64 || 1 probability, therefore the *Disrupter* and *Organizer* have the same chance to sleep and awake, with execution time = 180000 μsecs. This experiment runs with variations in the number of runs in the network, i.e. 3, 10, and 20 times. Events from each run are recorded in the trace log files. The information from various runs are calculated as the average and become a point in the result chart.

Results:

The x-axes in 7.22 denotes the probabilities of the three processes (*Disrupter,*

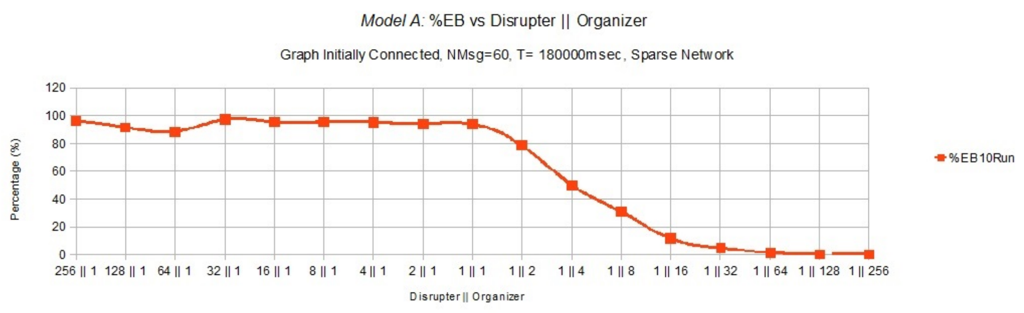


Figure 7.22: *Model A*: %EB vs. *Disrupter* || *Organizer* with Time = 180000 μsecs.

Organizer, and *Updater*) in accessing the network. On the y-axes, properties of the executions as the average over the 10 runs: %EB (the number of messages that are eventually black), being plotted. Notice that the %EB curve is slightly dropped from the point where *Disrupter* || *Organizer* = 128 || 1 to the point where *Disrupter* || *Organizer* = 64 || 1; afterwards the %EB increases again and then follows the trend towards zero. The decreasing shows that the results at that points do not follow the %EB curve trend and is suspected mainly as the effect from the number of runs.

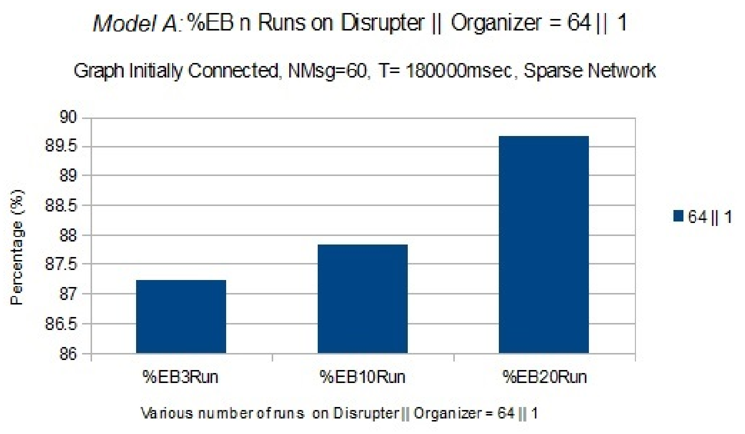


Figure 7.23: *Model A*: %EB vs. *Disrupter* || *Organizer* = 64 || 1, with N runs.

After three different numbers of runs performed, the results are shown as in Figure 7.23. The x-axes in Figure 7.23 denote various number of runs (3 runs, 10 runs, and 20

runs), while on the y-axes, the number of the messages that are eventually black(%EB) being plotted.
Conclusion: The figure shows that the more we run the program then we can obtain the more refined result.

7.4 Experiments of *Model B*

For the *Updater* Process model (*Model B*), to validate model means

$$\text{if } \mathcal{M} \models \text{Prop1} \wedge \text{Prop4} \wedge \text{Prop5} \wedge \text{Prop6} \wedge \text{Fair1} \wedge \text{Fair2} \text{ then } \mathcal{M} \models \forall \mathsf{m}. \Diamond \mathit{black}(\mathsf{m})$$

with the details of properties as follows.

1. Prop1 : Paths exist sufficiently often

   Prop1: $\forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\Diamond\ \mathit{pathX}(\mathsf{n}_1, \mathsf{n}_2)$

2. Prop4 : Messages get attention sufficiently often and paths for messages are obtained from the routing table.

   Prop4: $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ (((\Box\Diamond\ \exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}), \mathit{dest}(\mathsf{m})).\ \mathsf{rt}(\mathit{at}(\mathsf{m}), \mathit{dest}(\mathsf{m}), \mathsf{p}))$
   $\wedge\ \mathit{red}(\mathsf{m}))$
   $\Rightarrow\ \Diamond\ (\mathit{green}(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(\mathit{at}(\mathsf{m}), \mathit{dest}(\mathsf{m})).\ \mathit{path}(\mathsf{m}, \mathsf{p}))))$

3. Prop5 : Messages get attention sufficiently often and paths for messages are obtained from the routing table, and when they are accessed, they can be moved

at some time (modified Prop4)

Prop5: $\forall \mathsf{m} \in \mathsf{M}.\ \Box\ ((\Box\Diamond\ \exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ (\mathsf{rt}(at(\mathsf{m}), dest(\mathsf{m}), \mathsf{p}))) \Rightarrow$
$\Diamond\ (black(\mathsf{m}) \wedge$
$(red(\mathsf{m})\ \Rightarrow\ \bigcirc(green(\mathsf{m})\ \wedge\ (\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ path(\mathsf{m}, \mathsf{p})))) \wedge$
$(\exists \mathsf{p} \in \mathsf{P}(at(\mathsf{m}), dest(\mathsf{m})).\ ((green(\mathsf{m})\ \wedge\ at(\mathsf{m}) \neq dest(\mathsf{m})\ \wedge$
$pathX(at(\mathsf{m}), dest(\mathsf{m})\ \wedge\ path(\mathsf{m}, \mathsf{p})))\ \Rightarrow$
$(up(1st\_e(\mathsf{p}))\ \wedge\ \bigcirc(green(\mathsf{m})\ \wedge\ path(\mathsf{m}, tl(\mathsf{p}))$
$\wedge\ at(\mathsf{m}) = next\_node(\mathsf{p}))))) \wedge ((green(\mathsf{m})\ \wedge\ at(\mathsf{m}) = dest(\mathsf{m}))$
$\Rightarrow\ \bigcirc\ black(\mathsf{m}))))$

4. Prop6 : The routing table is populated sufficiently often (for *Model B*)

   Prop6:
   $\forall\ \mathsf{n}_1, \mathsf{n}_2 \in \mathsf{N}.\ \Box\ ((\Box\Diamond\ pathX(\mathsf{n}_1, \mathsf{n}_2))\ \Rightarrow\ \Diamond\ (\exists \mathsf{p} \in \mathsf{P}(\mathsf{n}_1, \mathsf{n}_2).\ \mathsf{rt}(\mathsf{n}_1, \mathsf{n}_2, \mathsf{p})))$

5. Fair1 : finiteness of path, as defined in Chapter 5.

   $$\text{Fair1: } \forall \mathsf{m} \in \mathsf{M}.\ ((green(\mathsf{m})\ \wedge\ \neg\Diamond\ red(\mathsf{m}))\ \Rightarrow\ (\Diamond\ black(\mathsf{m})))$$

6. Fair2 : no livelock (livelock-free), as defined in Chapter 5.

   $$\text{Fair2: } \forall \mathsf{m} \in \mathsf{M}.\ \neg\ \Box\Diamond\ (green(\mathsf{m})\ \Rightarrow\ \bigcirc red(\mathsf{m}))$$

7.4.1 RULER specification for *Model B*

Model B (the *Updater* Process model) uses the RULER specification as listed in Appendix B.2. In *Model B*, the *Updater* process performs update on the routing table by marking the path as *available* or *unavailable*. Therefore, there are the additional state rules in specification of *Model B*, as the replacement of the Routing Update rule. These state rules are as follows.

- UpdaV : records the number of existing paths during runtime, based on the action of UAv (which turns the path status to *available*). This variable has a similar role to PathT.

```
 state UpdaV(UpdCount:int) {
  UAv(h:obj, k:int) -> UpdaV(UpdCount+1), print("UpdaV = " + (UpdCount+1));
}
```

- UpdNAv : records the number of unavailable paths during runtime, based on the action of UNAv (which turns the path status to *unavailable*). Total number of UpdaV and UpdNAv indicates the number of *Updater* actions involved during runtime.

```
state UpdNAv(UpdCount2:int) {
  UNAv(h:obj, k:int) -> UpdNAv(UpdCount2+1), print("UpdNAv = " + (UpdCount2+1));
}
```

Another difference from *RuleR* specification of *Model A* is the definition of *Calculate* monitor, as follows.

```
state CountA(cct:int, cct2:int, cct2a:int, cct3:int, cct4:int, cct5:int, ttc:int) {
   pIn(ct:int), tIn(tc:int) -> CountA(ct, cct2, cct2a, cct3, cct4, cct5, tc),
      print("%EB = " + (ct/60.0)*100 + ", PathX = " + (cct2/55.0) +
      ", %Updater = " + (((cct2/55.0) + (cct2a/55.0))/((ct + cct3 + cct5 + (((cct2/55.0) +
                     (cct2a/55.0))))*1.0))*100 +
      ", %AllOrg = " + (cct3/((ct + cct3 + cct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 +
      ", %OrgT = " + (cct4/((ct + cct3 + cct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 +
      ", %DisT = " + (cct5/((ct + cct3 + cct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 + ",
                     tc = " + tc) ;
      ... (see Appendix B.2 for the details)
 }
```

The percentage number of messages that are eventually black, namely %EB, is obtained from the number of messages that are eventually black within the execution time relative to the number of messages within the network (in this case, 60). The number of available paths existing within the execution time is calculated in total during runtime as PathX, relative to the total number of possible paths in the routing table (in this case, 55). %Updater is the percentage of UpdAv and UpdNAv, relative to the number of the *Disrupter, Organizer* and *Updater* events during runtime. %AllOrg, %OrgT, and %DisT are the percentage of AllOrg, OrgT, and DisT, respectively, relative to the number of the *Disrupter, Organizer* and *Updater* events during runtime.

For checking the livelock, the RULER monitor specification is as listed in Appendix B.2. This checking is associated with properties Fair2 of *Model A*.

```
always Red{
    writeTheRoute(m:obj, idx:int, p:string) -> redG(idx, 1, p),
                                          print("message m " + idx + "path = " + p);
    RGr(m:obj, idx:int, p:string, ps:int) -> Black(m, idx);
  }
```

In *Model A* specification, the trigger event which activates the state *redG* is RoutingUpdate (which belongs to the *pathFinder* process). Here, at the occurrence of *writeTheRoute* event of the *Organizer* process (i.e., when the *Organizer* process pick the available path for a message), the state *redG* will be activated. This state will detect whether a particular message is in the position where livelock may occur (i.e., on a node within a circular path). The state *redG* then will be proceed the same as in *Model A*.

7.4.2 Results for *Model B*

The results of these experiments are presented based on the parameters mentioned in Section 7.1. Firstly, through the *Experiment 1*, we can check whether the traces of the program satisfy the properties of *Model B*. Moreover, if the properties of *Model B* are satisfied by the traces, then we expect that the results of the *Experiment 1* will show whether all the messages are eventually delivered (whether red(m) eventually black (m)).

After we know the possibilities of the messages are eventually delivered, now we suspect that if the length of the traces increased, then the chance for the messages eventually delivered will be higher. Therefore, the *Experiment 2* is set out to check the traces using various execution times. Furthermore, the impact of livelock potential occurrence to the number of messages eventually delivered is explored through *Experiment 1* and *Experiment 2*.

Both *Experiment 1* and *Experiment 2* use the same density of the graph. Here, the dynamic changes of dynamic network models are established from the changes of edge status over time. Therefore, the *Experiment 3* is performed to learn whether the density of the graph will affect the number of message becoming black/delivered. Particularly, we can know through the Experiment 3, whether the density of the graph affects the path availability within the network; and moreover, whether the density of the graph affects the percentage of messages that are delivered.

Properties of *Model B* show that the messages should get visited sufficiently often to become eventually green (moreover, eventually black). The *Experiment 4* is performed to know whether the chance of the messages to become eventually green/black also will be affected by the various numbers of the messages within the network.

The initial conditions for the program itself may affect the result. Therefore, the *Experiment 5* explore the possibility of initial conditions effect, particularly using the different initial graph. The number of runs also can affect the results of the program, therefore the *Experiment 6* is carried out to show this.

7.4.2.1 Experiment 1: probabilities between the three processes

Aim: The *Experiment 1* aims to observe whether the traces satisfy properties Prop1, Prop4, Prop5, Prop6, Fair1, and Fair2 when we set the frequency of the two processes, *Disrupter* and *Organizer* in each model instance. Furthermore, it aims to observe whether all messages are eventually delivered.
Setting: The following parameters are fixed. Using *Check model* for *Model B*, the number of messages in the network is 60, the execution time is 60000 μsecs, and run on a sparse network. The graph is initially connected, and at the start of the programs, all edges have the status *on*. We consider 25 different settings of the probabilities of process access to the network from 256 || 1 || 1 (*Disrupter* sleeps 256 times longer than the *Organizer* and the *Updater*), through to 1 || 256 || 1 (i.e the *Updater* rate is fixed, in which *Organizer* sleeps 256 times longer than the *Disrupter* and the *Updater*), also through to 1 || 1 || 256 (i.e the *Organizer* rate is fixed, in which *Updater* sleeps 256 times longer than the *Disrupter* and the *Organizer*). Each model instance has 10 runs of the program. Events from the run of each model instance are recorded in the trace log files. These information from 10 runs are calculated as the average and become a point in the result chart.
Results:

Figure 7.24 shows the trend of %EB in *Model B* with fixed *Updater* rate. The

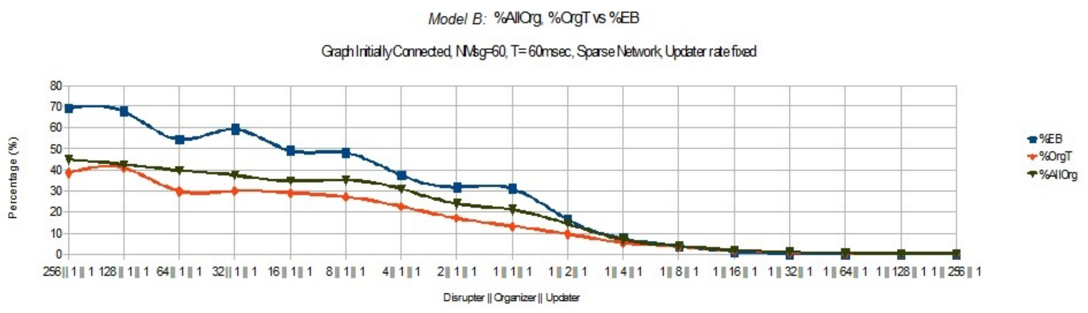


Figure 7.24: *Model B*: Number of messages eventually black on sparse network, with fixed *Updater* rate.

number of messages eventually black drop towards zero after point 1 || 2 || 1. This happens because although the *Updater* keeps updating the routing table (Prop6), but the *Organizer* is less active. Therefore, there is also a less chance of a message being visited by the *Organizer* (Prop4). The result shows that Prop6 is a sufficient condition for messages to eventually reach their destinations. If we have Prop6 and Prop4, then 'messages to eventually reach their destinations' must follow.

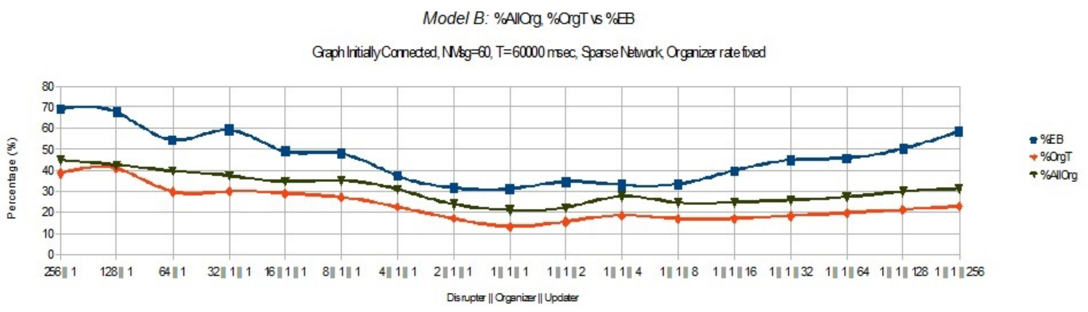


Figure 7.25: *Model B*: Number of messages eventually black on sparse network, with fixed *Organizer* rate

Figure 7.25 shows the trend of %EB in *Model B* with fixed *Organizer* rate. The number of messages eventually black increased again after point 1 || 1 || 8. This should not occur because logically, the *Updater* is less active and therefore the routing table might be wrong (¬Prop6). However, the *Organizer* rate is fixed therefore message get visited sufficiently often (Prop4 and Prop5). The result shows again that Prop6 is a sufficient but not necessary, because a number of messages still can get through to their destinations with this condition.

Moreover, as we know from Chapter 6 that a *Disrupter* process has two actions, DTC (to make the link connected) and CTD (to make the link disconnected). These two actions have equal probability to do their job within the *Disrupter* process. The disruption is not only disconnecting the link but also connecting the link. Therefore, there is always a chance of the message to be eventually black under this condition. Suppose we set the probabilities between the two actions DTC and CTD as 1 vs. 10 (i.e. less chance of connecting the link), then we get the different plot about %EB, as shown in Figure 7.26.

Conclusion: From the results above, we can conclude as follows.

- The result shows that: if path existence occurs infinitely often, the messages being looked at sufficiently often, and the routing table is populated sufficiently often then the number of messages eventually reaches their destinations increased. This result supports the proof of Lemma 1, 2 and 3, hence the Theorem that has been discussed in Chapter 5.

- The properties Prop1 (i.e. path exists infinitely often) is sufficient but not necessary for message eventually black, whether the path existences relate to the messages or not.

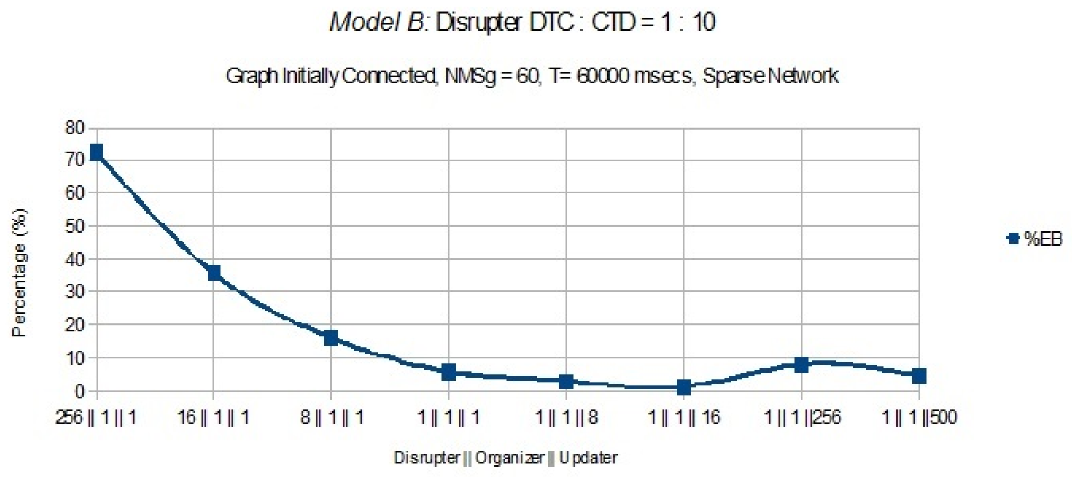


Figure 7.26: *Model B*: with probabilities between the action DTC and CTD = 1 : 10.

- The properties Prop4 and Prop5 are sufficient and necessary for message eventually black.
- The properties Prop6 is a sufficient condition but not necessary for message eventually black.
- In *Model B*, the intra-probabilities (probabilities between the actions within a process) also determines the number of messages eventually black. Furthermore, the intra-probabilities can be used to model how dynamic the network is.

7.4.2.2 Experiment 2: variation of execution time

Aim: Experiment 2 aims to observe whether the traces satisfy properties Prop1, Prop4, Prop5, Prop6, Fair1, and Fair2 when the programs run for various execution time, in a model instance. Furthermore, to observe whether all messages are eventually delivered.

Setting: The following parameters are fixed. Using *Check model* for *Model B*, the number of messages that being delivered is 60, and run on a sparse network. The graph initially connected, means at the start of programs, all edges have the status *on*. The model instance that used here is 1 || 1 combination, therefore the *Disrupter* and *Organizer* have the same chance to sleep and awake. This experiment runs on the variation of execution time for the model instance 1 || 1 || 1 (30000, 60000, 120000, 240000, and 300000 μsecs), and repeat for three times. Events from each run are recorded in the trace log files. The information from three runs are calculated as the average and become a point in the result chart.

Results:

Figure 7.27 shows the number of messages that eventually turn black relative to the

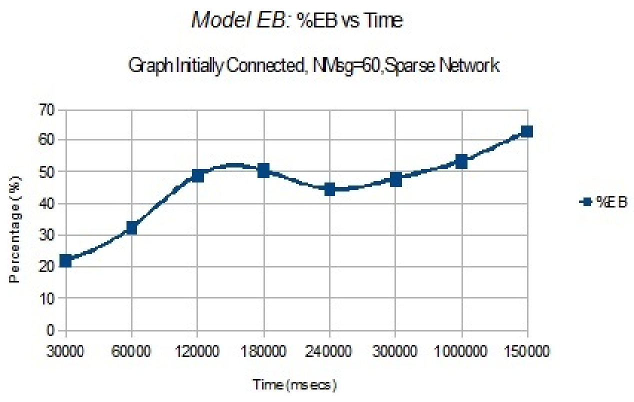


Figure 7.27: *Model B*: Percentage of messages eventually black vs. time in sparse network, with D || O || U = 1 || 1 || 1 .

different execution time. The x-axes denote the variation of execution time, while on the y-axes we plot the curve for %EB over time. The %EB increases sharply from the point where Time = 60000 μsecs to the point where Time = 120000 μsecs. Afterwards, the %EB only slight increases, which related to the *Organizer*'s activity and potential livelock over time.

Conclusion: Using *Model B*, no need for dynamic changes to cease to obtain the liveness property (i.e. messages eventually delivered). This is shown by the longer the traces, the number of messages being delivered more likely to happen in combination 1 || 1 || 1. Potential livelock occurrence makes the rate of message eventually black getting slower. The number of messages being caught and affected by potential livelock influence the number of messages eventually black within the execution time.

7.4.2.3 Experiment 3: density of initial graph

Aim: Experiment 3 aims to observe whether the traces satisfy properties Prop1, Prop4, Prop5, Prop6, Fair1, and Fair2 when the programs run on different density of the initial graph. Furthermore, to observe whether all messages are eventually delivered.

Setting: The following parameters are fixed. Using a *Check model* for *Model B*, the number of messages that being delivered is 60, and the graph is initially connected (i.e., the start of the programs, all edges have the status *on*). We consider 25 different settings of the probabilities of process access to the network from 256 || 1 || 1 (*Disrupter* sleeps 256 times longer than the *Organizer* and the *Updater*), through to 1 || 256 || 1 (i.e the *Updater* rate is fixed, in which *Organizer* sleeps 256 times longer than the *Disrupter* and the *Updater*), also through to 1 || 1 || 256 (i.e the *Organizer* rate is fixed, in which *Updater* sleeps 256 times longer than the *Disrupter* and the *Organizer*). Each model instance has 10 runs of program, and run first on a sparse network, then on a dense network. Events from the run of each model instance are recorded in the trace log files. The information from 10 runs are calculated as the average and become a point in the result chart.

Results:

In Figure 7.28 and Figure 7.29, x-axes denote the probabilities between the three

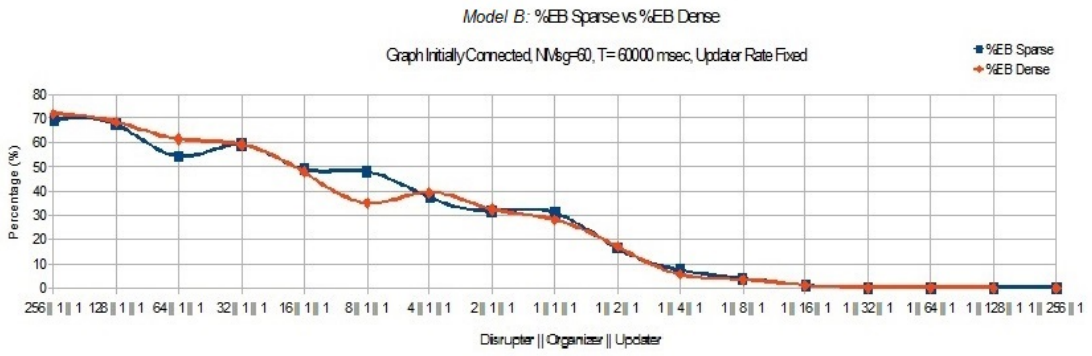


Figure 7.28: *Model B*: Number of messages eventually black on sparse network vs. dense network, with fixed *Updater* rate.

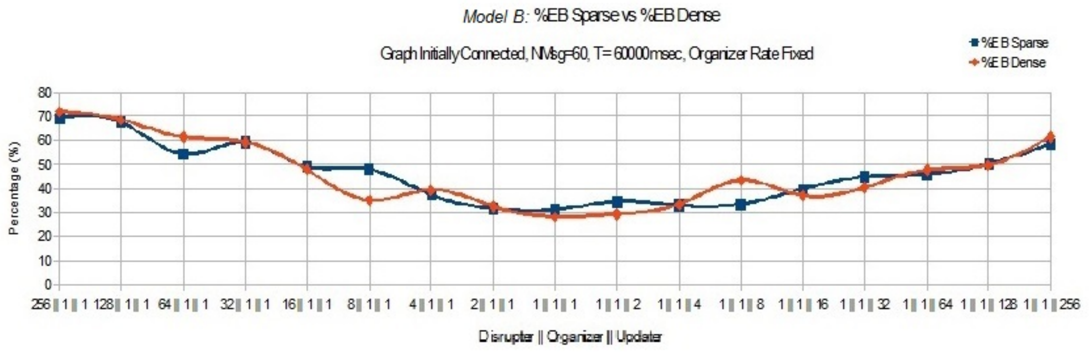


Figure 7.29: *Model B*: Number of messages eventually black on sparse network vs. dense network, with fixed *Organizer* rate.

processes and y-axes plot two properties of execution with variation of the density

of the graph. Figure 7.28 and Figure 7.29 show that there is no significant different number of %EB if we use different densities of the graph.
Conclusion: For *Model B*, there is no relation between the density of a graph with the number of messages eventually turning black. If we use different combination such as: a big and dense graph, then the difference of %EB might be significant.

7.4.2.4 Experiment 4: the number of messages in network

Aim: Experiment 4 aims to observe whether the traces satisfy properties Prop1, Prop4, Prop5, Prop6, Fair1, and Fair2 when the programs run for different number of messages in the network. Furthermore, it aims to observe whether all messages are eventually delivered.
Setting: The following parameters are fixed. Using the *Check model* for *Model B*, the graph is initially connected (i.e., at the start of programs, all edges have the status *on*) and run on a sparse network. The model instance that used here is a probability of 1 || 1 || 1, therefore the *Disrupter, the Organizer* and the *Updater* have the same chance of being asleep and awake, with execution time = 60000 μsecs. This experiment run with various numbers of messages in the network, i.e., 10, 20, 30, 40, 50, 60, 70, and 80 messages, and is repeated for three times. Events from each run are recorded in the trace log files. The information from three runs are calculated as the average and become a point in the result chart.
Results:

In Figure 7.30, the x-axes denote the number of messages in the networks, increased from 10 to 80 and the y-axes plot the properties of execution in which messages are eventually black. Figure 7.30 shows that when the number of messages is increasing in size, then the chance of all messages eventually being black is decreased. This is because the number of messages affected the chance of the *Organizer* visiting all the messages.
Conclusion: The number of messages within the network influences the chance of the messages being visited by the *Organizer* process. Furthermore, it affects the number of messages that eventually turn black within the execution time.

7.4.2.5 Experiment 5: connectivity of original graph

Aim: Experiment 5 aims to observe whether the traces satisfy properties Prop1, Prop4, Prop5, Prop6, Fair1, and Fair2 when the programs run on different connectivity of the

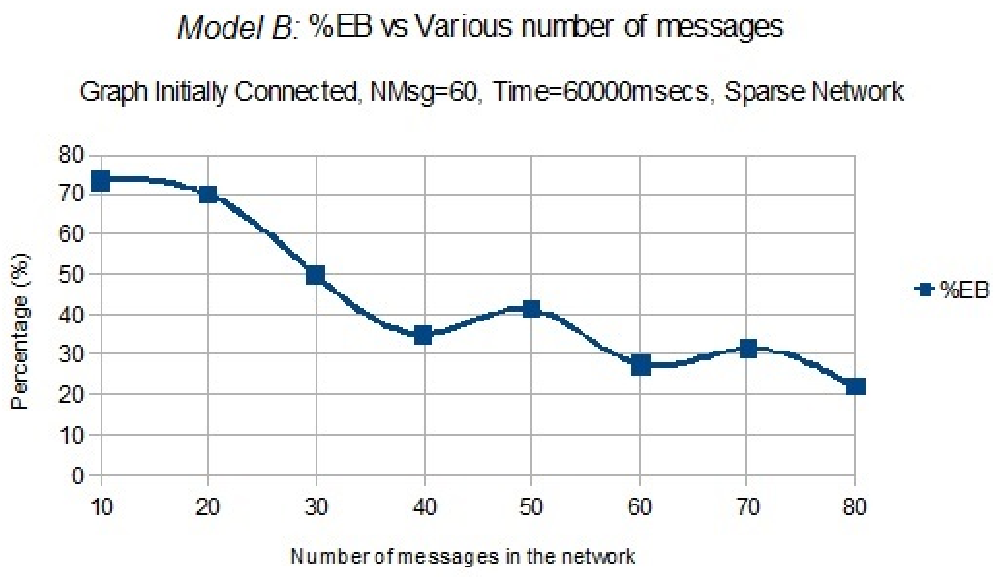


Figure 7.30: *Model B*: Percentage of messages eventually black vs. various numbers of messages, with D || O || U = 1 || 1 || 1.

original graph. Furthermore, it aims to observe whether all messages are eventually delivered.

Setting: The following parameters are fixed. Using a *Check model* for *Model B*, the number of messages that being delivered is 60, and run on a sparse network. We consider 25 different settings of the probabilities of process access to the network from 256 || 1 || 1 (*Disrupter* sleeps 256 times longer than the *Organizer* and the *Updater*), through to 1 || 256 || 1 (i.e the *Updater* rate is fixed, in which *Organizer* sleeps 256 times longer than the *Disrupter* and the *Updater*), also through to 1 || 1 || 256 (i.e. the *Organizer* rate is fixed, in which *Updater* sleeps 256 times longer than the *Disrupter* and the *Organizer*). Each model instance has 10 runs of the program, and run first on an initially connected graph, then on an arbitrarily graph. Events from the run of each model instance are recorded in the trace log files. These information from 10 runs are calculated as the average and become a point in the results chart.

Results:

The x-axes in Figure 7.31 and Figure 7.32 denotes the probabilities of the three processes (*Disrupter, Organizer* and *Updater*) in accessing the network. On the y-axes, we plot two different properties of the executions (average over the 10 runs). These are: *%EB Init* (the number of messages $\Diamond$ black (m)) on the sparse network with initial graph connected and *%EB Arbit* on the sparse network with initial graph arbitrarily

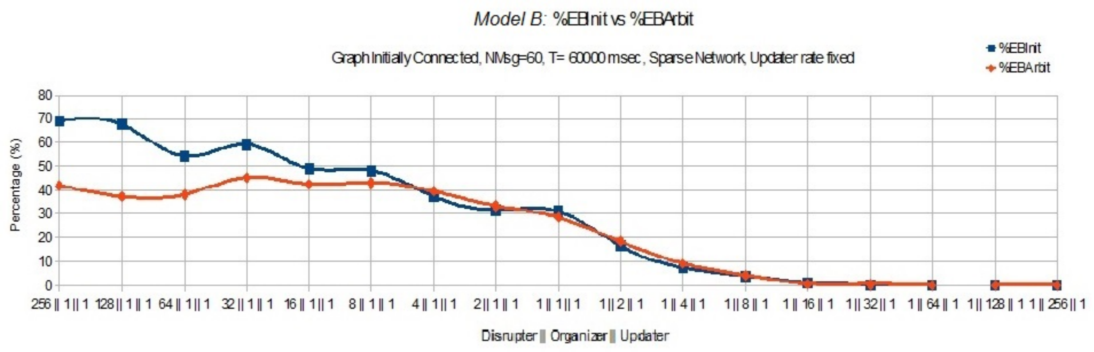


Figure 7.31: *Model B*: Percentage of messages on the graph initially connected vs. graph arbitrarily connected, with *Updater* rate fixed.

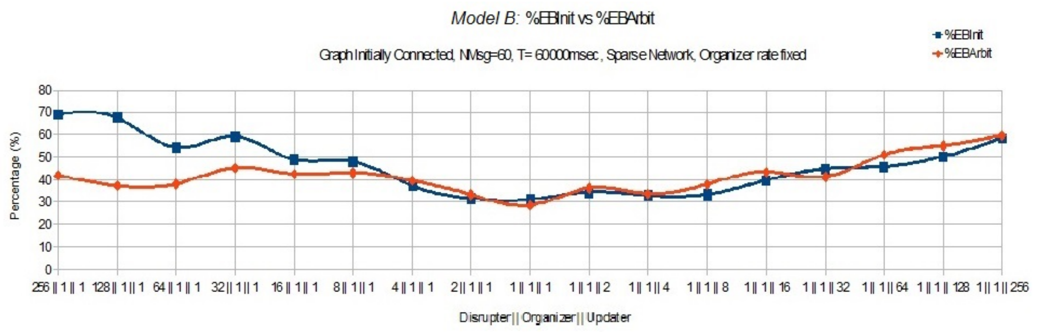


Figure 7.32: *Model B*: Percentage of messages on the graph initially connected vs. graph arbitrarily connected, with *Organizer* rate fixed.

connected.

Figure 7.31 and Figure 7.32 show that when the *Disrupter* is less disruptive, then the number of %EB Init are nearly 80 % at point 256 || 1 || 1 and about 30 % at point 1 || 1 || 1. The interesting fact in Figure 7.31 and Figure 7.32 is that the number of %EB Arbit almost similar to %EB Init, with no significant difference.

Conclusion: Using different initial graph (initial graph connected and initial graph arbitrarily connected), the results shows that, although there is a similar pattern (heading towards zero) and a similar number of messages eventually black/delivered. Therefore, for *Model B*, the %EB results do not affected by the initial condition of the graph.

7.4.2.6 Experiment 6: number of runs

Aim: Experiment 6 aims to observe whether the traces satisfies properties Prop1, Prop4, Prop5, Prop6, Fair1, and Fair2 when the programs run on different number of runs, in a model instance. Furthermore, it aims to observe whether all messages are eventually delivered.

Setting: The following parameters are fixed. Using *Check model* for *Model B*, the graph initially connected (i.e., at the start of programs, all edges have the status *on*) and run on a sparse network. The model instance that used here has a 64 || 1 || 1 probability, therefore the *Disrupter* and *Organizer* have the same chance to sleep and awake, with execution time = 180000 μsecs. This experiment runs with variations in the number of runs in the network, i.e. 3, 10, and 20 runs. Events from each run are recorded in the trace log files. The information from various of the runs are calculated as the average and become a point in the result chart.

Results:

The x-axes in 7.33 denotes the probabilities of the three processes (*Disrupter,*

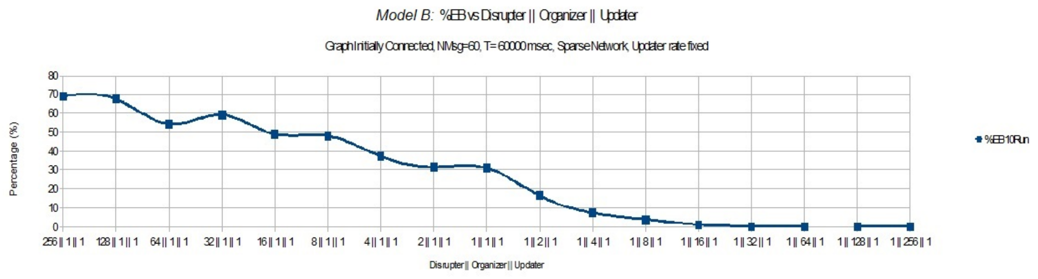


Figure 7.33: *Model B*: Percentage of messages vs. *Disrupter* || *Organizer* || *Updater*, *Updater* rate fixed.

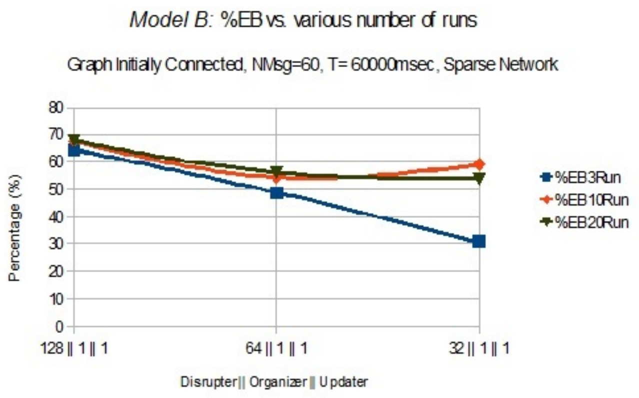


Figure 7.34: *Model B*: Percentage of messages vs. number of runs, with *Updater* rate fixed and D || O || U = 64 || 1 || 1.

Organizer, and *Updater*) in accessing the network. On the y-axes, a properties of the executions as the average over the 10 runs: %EB (the number of messages that are eventually black), is plotted. Notice that the %EB curve is slightly dropped from the point where *Disrupter* || *Organizer* || *Updater* = 128 || 1 || 1 to the point where *Disrupter* || *Organizer* || *Updater* = 64 || 1 || 1; afterwards the %EB increases again and then follows the trend towards zero. The decreasing shows that the results at that point do not follow the %EB curve trend. These results mainly caused by the effect from the number of runs.

After the execution of three different numbers of runs, the result is shown as in Figure 7.34. The x-axes in Figure 7.34 denote various the probabilities of the three processes from 128 || 1 || 1 to 32 || 1 || 1, while on the y-axes, we plot three properties of executions. These are: %EB 3runs, %EB10 runs, and %EB 20 runs.
Conclusion: The figure shows that the more we run the program then we can obtain the more refined result.

7.5 General conclusions

Now we summarise what is achieved by the experimental work. Using different initial graph (initial graph connected and initial graph arbitrarily connected) gives a different result for each model. In *Model A*, the number of messages eventually black/delivered is affected by this initial condition (%EBInit higher than %EBArbit). On the contrary, for *Model B*, the %EB results do not affected by the initial condition of the graph. Furthermore, result of *Model B* experiment shows that the intra-probabilities (probabilities between the actions within a process) also determines the number of messages eventually black, as well as how dynamic the network is.

Generally, the results show that properties Prop2 (messages get attention sufficiently often) is a necessary condition for messages to eventually reach their destinations (denoted by "*G*"), either for *Model A* or *Model B*. If the models do not have properties Prop2, then they will not have *G*. The results also show that this properties Prop2 is also a sufficient condition for *G*. This means that if we have Prop2, then *G* must follow (Prop2 guarantees *G*). The properties Prop2 and Prop3 (see Section 7.3) are sufficient and necessary for message eventually black.

By contrast, Prop6 (the routing table is populated sufficiently often) for *Model B* is a sufficient condition for *G*. If we have Prop6 **and** Prop4 **and** Prop5, then *G* must follow. It is sufficient but not necessary, because the results show that a number of messages still can get through to their destinations with the following conditions : $\neg Prop6 \wedge Prop5 \rightarrow G$. The properties Prop4 and Prop5 (see Section 7.4 for properties definition) are sufficient and necessary for message eventually black.

The experiments of *Model A* show that: if path existence occurs infinitely often and the messages are looked at sufficiently often, then the number of messages that eventually reach their destinations increased. Moreover, experiments of *Model B* show that: if path existence occurs infinitely often; the messages being looked at sufficiently often; and the routing table is populated sufficiently often; then the number of messages eventually reaches their destinations increased. Therefore, the result of both experiments (*Model A* and *Model B*) supports the proof of Lemma 1, 2 and 3, hence the Theorem (see Chapter 5). In summary, the RV techniques using RULER enable us to determine whether an implementation of a model satisfies the conditions to ensure that all messages are delivered.

7.6 Discussion

PathX is the number of the available paths being found by the *pathFinder* process in *Model A*. These paths relate to the path that is needed by the messages. This form is an extension of Prop1 mentioned above for *Model A*. Suppose we have a path available from source = A to destination B, namely ACDB. We also have other paths available that are not related or needed by the messages in the network. If we have a message m that needs to go from A to B, then at some point message m eventually move to C. Properties Prop1 means that we have path ACDB infinitely often. However, after this move, the system does not need this information anymore. The information about available path being needed now is CDB. Therefore, properties Prop1 is sufficient for the messages to eventually move and become black, but is not necessary.

Chapter 8

Conclusions

The research in dynamic network systems offers some challenges such as: message-passing mechanisms in unreliable networks; and computations attempting to reach a consensus or compute global properties. The majority of literature has not provided a systematic approach to overcome these challenges without the requirement of dynamic changes to cease, to obtain the correctness. Therefore, this thesis addressed this gap by developing a systematic approach to describe and reason about dynamic networks using concurrent systems techniques.

8.1 Contributions

The dynamic networks research presented in this thesis clearly constitutes an original set of contributions answering the questions left by the majority of literature. The systematic approach articulated a number of important findings:

1. At its simplest level, a network consists of a collection of nodes connecting to each other through edges. In a message-passing network, each node communicates by exchanging messages in an attempt to deliver them to their destinations. In a dynamic network, nodes and/or edges may become inoperative or operative. This representation of the dynamics of a network clearly models unreliable networks. It also models mobile and wireless networks by considering edges as possible communication links and operative edges as the links established at a particular time.

2. The correctness of dynamic networks can be established without the termination requirement. This correctness has been developed in terms of ensuring that:

even when routing tables do not reflect the actual network connections, the routing information is correct sufficiently often; messages eventually get delivered; the network is sufficiently connected for sufficiently often; and there is no persistent livelock. The main contributions are: modelling dynamic networks using concurrent systems; factorisation of proof; and runtime verification of the implementation of dynamic network models. This correctness of dynamic networks as an answer for the question in Chapter 1: "how does one model and reason about dynamic network systems in an appropriate formal and modelling framework, with an assumption that the graph is always connected, but as weakly as possible?"

3. A series of dynamic network models as concurrent systems has been developed. This series of models represents dynamic networks start from a simple network behaviour (in which the routing update occurs instantaneously within the network) to more realistic behaviour (in which the routing update is executed by a process concurrently). Furthermore, this series of models can be extended to represent the more realistic and actual dynamic networks, by formulating the properties of network according to the actual systems.

4. A new approach to proof techniques for dynamic networks in which using ideas from concurrent systems [49] to analyse message-passing has been introduced. A modal logic is used and concepts of fairness which capture network properties are formulated. To express dynamic networks as concurrent systems [42], the dynamic changes are considered to be the result of a "demonic" process which runs concurrently with routing updates and message-passing. Using such description, correctness of networks that never cease to change can be established. The correctness of dynamic networks means that under certain conditions, all messages will eventually be delivered. Such conditions are defined using the properties of networks which are formulated independently of the mechanisms in actual networks. By showing the correctness can be proved, and therefore it also provides "a factorisation" of proofs of correctness for actual dynamic networks. This definition of correctness as an answer to the question in Chapter 1: "how can we describe about the correctness criteria for re-routing algorithms in dynamic network systems?".

5. Two abstract models as concurrent systems are implemented and then a runtime verification systems RULER [5] is adopted, to analyse execution traces to

test whether model instances satisfy the modal correctness for message delivery. These ways can be regarded as an answer for this following question in Section 1.1: “The formal framework should be useful to obtain the desired correctness criteria for dynamic network systems. How does one obtain **realistic** implementations, proof of correctness, and models of actual systems?”

8.2 Future Works

There are a number of areas that did not address and left as an open various issues. First, this thesis did not explored about probabilistic and timing in dynamic network models. The probabilistic in dynamic network models enable us to formulate a stochastic model using the probability number, for instance, to estimate: the connection of the graph, the existence of an edge, and the chance of a message being visited and picked. The timing dynamic network model has the assumptions about delivery time and approximate real time.

Secondly, in the future work, Hoare logic will be needed for proving the particular routing algorithms. This step is needed particularly when we want to extend the correctness of dynamic network models into more specific mechanisms and routing algorithms. This means we need to get into the lower level of model abstraction, with more detail properties definitions.

Thirdly, there is still an open question for dynamic networks as concurrent systems. In terms of abstraction level, can we still use the concurrent systems for the lower level? There will be a branch decision to use broadcast or point-to-point mechanism. Furthermore, the issue of communication between the processes could be one of the main focuses.

Bibliography

[1] Alotaibi, E. & Mukherjee, B. *A survey on routing algorithms for wireless Ad-Hoc and mesh networks*. Computer Networks, Elsevier, 2012, Vol. 56(2), pp. 940-965.

[2] Ashcroft, E.A. *Proving assertions about parallel programs*. Journal of Computer and System Sciences, Elsevier, 1975, Vol. 10(1), pp. 110-135.

[3] Baier, C., Katoen, J.-P. & others. *Principles of model checking*. MIT press Cambridge, 2008, Vol. 26202649.

[4] Baldoni, R., Bertier, M., Raynal, M. & Tucci-Piergiovanni, S. *Looking for a definition of dynamic distributed systems*. Malyshkin, V. (ed.) Parallel Computing Technologies. Springer, 2007, pp. 1-14.

[5] Barringer, H., Havelund, K., Rydeheard, D. & Groce, A. *Rule Systems for Runtime Verification: A Short Tutorial*. Bensalem, S. and Peled, D. (ed.), Runtime Verification, Springer Berlin Heidelberg, 2009, Vol. 5779, pp. 1-24.

[6] Basu, A. Ong, C.-H.L., Rasala, A., Shepherd, F.B. & Wilfong, G. *Route oscillations in I-BGP with route reflection*. Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications. ACM, 2002, pp. 235-247.

[7] Ben-Ari, M. *Algorithms for on-the-fly garbage collection*. ACM Transactions on Programming Languages and Systems (TOPLAS), ACM, 1984, Vol. 6(3), pp. 333-344.

[8] Ben-Ari, M. *Principles of concurrent and distributed programming*. Addison-Wesley Longman, 2006.

[9] Casteigts, A., Flocchini, P., Quattrociocchi, W. & Santoro, N. *Time-varying graphs and dynamic networks*. Ad-hoc, Mobile, and Wireless Networks. Springer, 2011, pp. 346-359.

[10] Chen, Y. & Welch, J.L. *Self-stabilizing mutual exclusion using tokens in mobile ad hoc networks*. Proceedings of the 6th international workshop on Discrete algorithms and methods for mobile computing and communications. ACM, 2002, pp. 34-42.

[11] Clementi, A. & Pasquale, F. *Information Spreading in Dynamic Networks: An Analytical Approach*. Nikoletseas, S. and Rolim, J. D. (eds.) Theoretical Aspects of Distributed Computing in Sensor Networks. Springer Berlin Heidelberg, 2010, pp. 591-619.

[12] Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S. & Steffens, E.F. *On-the-fly garbage collection: an exercise in cooperation*. Communications of the ACM, ACM, 1978, Vol. 21(11), pp. 966-975.

[13] Dijkstra, E.W. *Self-stabilizing systems in spite of distributed control*. Communications of the ACM, ACM, 1974, Vol. 17(11), pp. 643-644.

[14] Elmokashfi, A., Kvalbein, A. & Dovrolis, C. *BGP churn evolution: a perspective from the core*. Networking, IEEE/ACM Transactions on, IEEE, 2012, Vol. 20(2), pp. 571-584.

[15] Emerson, E.A. *Temporal and modal logic*. Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), 1990, Vol. 995, pp. 1072.

[16] Fabrikant, A. & Papadimitriou, C.H. *The complexity of game dynamics: BGP oscillations, sink equlibria, and beyond*. Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2008, pp. 844-853.

[17] Feamster, N. & Balakrishnan, H. *Detecting BGP configuration faults with static analysis*. Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation - Volume 2 USENIX Association, 2005, pp. 43-56.

[18] Feamster, N. & Balakrishnan, H. *Correctness properties for Internet routing*. Allerton Conference on Communication, Control, and Computing, 2005.

[19] Feamster, N. & Balakrishnan, H. *Towards a logic for wide-area Internet routing*. ACM SIGCOMM Computer Communication Review, 2003, Vol. 33(4), pp. 289-300.

[20] Fisher, M. *An Introduction to Practical Formal Methods Using Temporal Logic*. John Wiley & Sons, 2011.

[21] Gries, D. *An exercise in proving parallel programs correct*. Communications of the ACM, ACM, 1977, Vol. 20(12), pp. 921-930.

[22] Griffin, T.G. & Wilfong, G. *An analysis of BGP convergence properties*. ACM SIGCOMM Computer Communication Review 1999, Vol. 29(4), pp. 277-288.

[23] Griffin, T.G., Shepherd, F.B. & Wilfong, G. *The stable paths problem and interdomain routing*. IEEE/ACM Transactions on Networking (ToN), IEEE Press, 2002, Vol. 10(2), pp. 232-243.

[24] Griffin, T. & Wilfong, G. *A safe path vector protocol*. INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE 2000, Vol. 2, pp. 490-499.

[25] Griffin, T.G. & Wilfong, G. *On the correctness of IBGP configuration*. ACM SIGCOMM Computer Communication Review 2002, Vol. 32(4), pp. 17-29.

[26] Griffin, T.G. & Sobrinho, J.L. *Metarouting*. ACM SIGCOMM Computer Communication Review 2005, Vol. 35(4), pp. 1-12.

[27] Grindrod, P. & Higham, D.J. *Evolving graphs: dynamical models, inverse problems and propagation*. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science, The Royal Society, 2010, Vol. 466(2115), pp. 753-770.

[28] Ho, A., Smith, S. & Hand, S. *On deadlock, livelock, and forward progress*. Technical Report, University of Cambridge, Computer Laboratory (May 2005), 2005.

[29] Hassan, H., Eltarras, R. & Eltoweissy, M. *Towards a Framework for Evolvable Network Design*. Akan, O., Bellavista, P., Cao, J., Dressler, F., Ferrari, D., Gerla, M., Kobayashi, H., Palazzo, S., Sahni, S., Shen, X. (S., Stan, M., Xiaohua, J., Zomaya, A., Coulson, G., Bertino, E. and Joshi, J. B. D. (ed.) Collaborative Computing: Networking, Applications and Worksharing Springer Berlin Heidelberg, 2009, Vol. 10, pp. 390-401.

[30] Hoare, C. A. R. *An axiomatic basis for computer programming*. Communications of the ACM, ACM, 1969, Vol. 12(10), pp. 576-580.

[31] Baeten, J., Bergstra, J., Hoare, C., Milner, R., Parrow, J. & de Simone, R. *The variety of process algebra*. Deliverable ESPRIT Basic Research Action, 1992, Vol. 3006

[32] Fiedor, J., Krena, B., Letko, Z. & Vojnar, T. *A uniform classification of common concurrency errors*. Computer Aided Systems Theory–EUROCAST 2011 Springer, 2012, pp. 519-526.

[33] Karsten, M., Keshav, S., Prasad, S. & Beg, M. *An axiomatic basis for communication*. ACM SIGCOMM Computer Communication Review 2007, Vol. 37(4), pp. 217-228.

[34] Kuhn, F., Lynch, N. & Oshman, R. *Distributed computation in dynamic networks*. Proceedings of the 42nd ACM symposium on Theory of computing, 2010, pp. 513-522.

[35] Kuhn, F. & Oshman, R. *Dynamic networks: models and algorithms*. ACM SIGACT News, ACM, 2011, Vol. 42(1), pp. 82-96.

[36] Kwiatkowska, M.Z. *Survey of fairness notions*. Information and Software Technology, Elsevier, 1989, Vol. 31(7), pp. 371-386.

[37] Labovitz, C., Malan, G.R. & Jahanian, F. *Internet routing instability*. Networking, IEEE/ACM Transactions on, IEEE, 1998, Vol. 6(5), pp. 515-528

[38] Lad, M., Nanavati, A., Massey, D. & Zhang, L. *An algorithmic approach to identifying link failures*. Dependable Computing, 2004. Proceedings. 10th IEEE Pacific Rim International Symposium on 2004, pp. 25-34.

[39] Leucker, M. & Schallhart, C. *A brief account of runtime verification*. The Journal of Logic and Algebraic Programming, Elsevier, 2009, Vol. 78(5), pp. 293-303.

[40] Lo, D., Cheng, K. & Han, J. *Mining Software Specifications: Methodologies and Applications*. Taylor and Francis Group, 2011.

[41] Lynch, N.A. *Distributed algorithms*. Morgan Kaufmann, 1996.

[42] Magee, J. & Kramer, J. *Concurrency: State Models and Java Programs*. Wiley, 2006.

[43] Manna, Z. & Pnueli, A. *Verification of concurrent programs: Temporal proof principles*. Kozen, D. (ed.) Logics of Programs Springer Berlin Heidelberg, 1982, Vol. 131, pp. 200-252.

[44] J. McCarthy & P.J. Hayes. *Some philosophical problems from the standpoint of artificial intelligence*. In B. Meltzer and D. Michie, editors, Machine Intelligence, Edinburgh Univ. Press, Edinburgh, Scotland, 1969, volume 4, pp. 463-502.

[45] Medhi, D. & Ramasamy, K. *Network Routing: Algorithms, Protocols, and Architectures*. Adams, R. (ed.) Morgan Kaufmann, 2007.

[46] Fitting, M. & Mendelsohn, R.L. *First-order modal logic*. Springer, 1998, Vol. 277.

[47] Nakano, T. & Suda, T. *Adaptive and Evolvable Network Services*. et al., K. D. (ed.) GECCO 2004 LCNS 3102 Springer-Verlag Berlin Heidelberg, 2004, pp. 151-162.

[48] O'Dell, R. & Wattenhofer, R. *Information dissemination in highly dynamic graphs*. Proceedings of the 2005 joint workshop on Foundations of mobile computing ACM, 2005, pp. 104-110.

[49] Owicki, S. & Gries, D. *An axiomatic proof technique for parallel programs I*. Acta informatica, Springer, 1976, Vol. 6(4), pp. 319-340.

[50] Owicki, S. & Lamport, L. *Proving liveness properties of concurrent programs*. ACM Transactions on Programming Languages and Systems (TOPLAS), ACM, 1982, Vol. 4(3), pp. 455-495.

[51] Ramanathan, R., Basu, P. & Krishnan, R. *Towards a formalism for routing in challenged networks*. Proceedings of the second ACM workshop on Challenged networks, ACM 2007, pp. 3-10.

[52] Ramamoorthy, C. & Ho, G.*Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets*. Software Engineering, IEEE Transactions on, 1980, Vol. SE-6(5), pp. 440-449.

[53] Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M. & Ratchford, T. *Automated API Property Inference Techniques*. IEEE Transactions on Software Engineering, IEEE Computer Society, 2013, Vol. 39(5), pp. 613-637.

[54] Aspnes, J. & Ruppert, E. *An Introduction to Population Protocols*. Garbinato, B., Miranda, H. and Rodrigues, L. (ed.) Middleware for Network Eccentric and Mobile Applications. Springer Berlin Heidelberg, 2009, pp. 97-120.

[55] Sami, R., Schapira, M. & Zohar, A. *Searching for Stability in Interdomain Routing*. INFOCOM 2009, IEEE 2009, pp. 549-557. Sami, R., Schapira, M. & Zohar, A.

[56] Schubert, L.K. *Monotonic solution of the frame problem in the situation calculus*, in: H.E. Kyburg and R. Loui, eds., Selected Papers, from the 1988 Society, for Exact Philosophy Conference, 1989.

[57] Sobrinho, J.L. *Network routing with path vector protocols: theory and applications*. Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications. ACM, 2003, pp. 49-60.

[58] Soedarmadji, E. & McEliece, R.J. *A dynamic graph algorithm for the highly dynamic network problem*. Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on 2006, pp. 5-pp.

[59] Tanenbaum, A.S. *Computer networks*, 4th edition. Prentice Hall PTR, 2003.

[60] Valmari, A. *A stubborn attack on state explosion*. Computer-Aided Verification 1991, pp. 156-165.

[61] Varadhan, K., Govindan, R. & Estrin, D. *Persistent route oscillations in interdomain routing*. Computer Networks, Elsevier, 2000, Vol. 32(1), pp. 1-16.

[62] Voellmy, A. *Proof of an interdomain policy: a load balancing multi-homed network*. Proceedings of the 2nd ACM workshop on Assurable and usable security configuration, 2009, pp. 37-44.

[63] Wang, A. *Formal Analysis of Network Protocols*. WPE-II Written Report. University of Pennsylvania Department of Computer and Information Science, 2010(MS-CIS-10-16).

[64] Wu, J. & Li, H. *A Dominating-Set-Based Routing Scheme in Ad Hoc Wireless Networks*. Telecommunication Systems. Springer Netherlands, 2001, Vol. 18, pp. 13-36.

[65] P. Zave. *A formal model of addressing for interoperating networks*. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, (eds.), FM, volume 3582 of Lecture Notes in Computer Science. Springer, 2005, pp. 318333.

[66] Zhao, X., Zhang, B., Massey, D., Terzis, A. & Zhang, L. *The Impacts of Link Failure Location on Routing Dynamics: A Formal Analysis*. SIGCOMM Asia Workshop 05, Apr. 12-14, 2005, Beijing, China. ACM, 2005.

Appendix A

Algorithms

A.1 OTF sorting algorithm

The ***sorter*** will do this execution below, with $0 \leq i \leq N-1$:

```
while (true)
  for (0 <= i < numArrayItem)
  /* or use random i to pick element randomly */
    if (A[i].colour = Red)
      lock A[i];
      if A[i] = N-1 /* last element in array */
        lock A[i-1];
        if A[i].value < A[i-1].value
          /* SwapXY action */
          Swap(A[i], A[i-1]);
          /* A[i].value > A[i-1].value */
          lock A[i-2];
          if A[i-2].value < A[i-1].value
          and ((A[i-2].colour = Blue) or (A[i].colour = Blue))
            release A[i-2];
            /* rb1 condition */
            A[i-1].colour = Blue;
            /* A[i-2] < A[i-1] < A[i] and A[i-1].colour = Blue */
            release A[i-1];
        else
          lock A[i-2];
          if A[i-2].value < A[i-1].value
          and ((A[i-2].colour = Blue) or (A[i-1].colour = Blue))
              release A[i-2];
              release A[i-1];
              /* rb2 condition */
              A[i].colour = Blue;
              /* A[i-2] < A[i-1] < A[i] and A[i].colour = Blue */
        else if all j != i are red, then
          release A[i-2];
          /* rb8 condition */
          A[i].colour = Blue;
          /* A[i].colour = Blue and A[j].colour = Red for j != i */
          release A[i-1];
    else
      if A[i] = 0 { the first element in Array }
        lock A[i+1];
        if A[i].value > A[i+1].value
          /* SwapXY action */
          Swap(A[i], A[i+1]);
          /* A[i].value < A[i+1].value */
          lock A[i+2];
          if A[i].value < A[i+1].value < A[i+2].value
          and ((A[i].colour = Blue) or (A[i+2].colour = Blue))
            release A[i+2];
            /* rb3 condition */
            A[i+1].colour = Blue;
            /* A[i+1].colour = Blue */
            release A[i+1];
          else
            if A[i].value < A[i+1].value < A[i+2].value
            and ((A[i+1].colour = Blue) or (A[i+2].colour = Blue))
              release A[i+2];
              release A[i+1];
              /* rb4 condition */
              A[i].colour = Blue;
              /* A[i].colour = Blue */
      else /* other elements in Array */
        lock A[i-1];
        lock A[i+1];
          if A[i-1].value <= A[i].value <= A[i+1].value
          and (A[i+1].colour = Blue and A[i-1].colour = Blue)
            release A[i-1];
            release A[i+1];
            /* rb5 condition */
            A[i].colour = Blue;
            /* A[i].colour = Blue */
          else
            if A[i-1].value > A[i].value < A[i+1].value
            and A[i-1].colour = Blue and A[i+1].colour = Blue
              release A[i+1];
              /* SwapXY action */
              Swap(A[i], A[i-1]);
              /* A[i-1].value < A[i].value < A[i+1].value */
              if A[i-1] != 0
                lock A[i-2];
                if A[i-2].value < A[i-1].value < A[i].value
                and A[i-2].colour = Blue and A[i].colour = Blue
                  release A[i-2];
                  /* rb6 condition */
                  A[i-1].colour = Blue;
                  /* A[i-1].colour = Blue */
                  release A[i-1];
                else /* element is at an appropriate position */
                  /* rb7 condition */
                  A[i-1].colour = Blue;
                  /* A[i-1].colour = Blue */
                  release A[i-1];
            else /* if p or s not blue then swap only */
               if (A[i].value < A[i+1].value )
               and (A[i-1].value > A[i].value) and
               (A[i-1].colour = Red or A[i+1].colour = Red)
                 release A[i+1];
                 /* SwapXY action */
                 Swap(A[i], A[i-1]);
                 /* A[i-1].value < A[i].value */
                 release A[i-1];
               else
                 if A[i-1].value < A[i].value
                 and A[i].value > A[i+1].value and
                   release A[i-1];
                   /* SwapXY action */
                   Swap(A[i], A[i+1]);
                   /* A[i].value < A[i+1].value */
                   release A[i+1];
      release A[i];
```

A.2 *Disrupter* algorithm

```
Process Disrupter
begin
  while true
    pick one edge (i,k) within the network randomly
    if (i,k) is connected then
       /* remove(i,k) action */
       disconnect(i,k);
       /* G[i,k] = 0 */
       /* for Model1 : recalculate route of message m which is affected
          by disrupter */

        if At(m,i) and i!= destination then
          /* GR1 condition */
          GR(m, i);
          /* m[i].colour = ''red'' */
        endif
        if At(m,k) and k!= destination then
          /* GR1 condition */
         GR(m, k);
         /* m[k].colour = ''red'' */
        endif
    else
      /* insert(i,k) action */
      connect(i, k);
      /* G[i,k] = 1 */
      /* for Model1 : recalculate route of message m which is affected
      by disrupter*/
    endif
end
```

A.3 *Organizer* algorithm

```
Process Organizer
begin
  while true
    // pick a message m
    choose message m at node j (randomly or systematically)
    case
      m.s==j and m.d==j :
        /* empty path */
        /* no move action, message m is already at destination */
        /* RG1 condition */
        RG(m, j);
        /* m[j].colour = ''green'' */
        /* GB condition */
        GB(m, j);
        /* m[j].colour = ''black'' */
      m.d==j : // message m reach the destination
        /* GB condition */
        GB(m, j)
        /* m[j].colour = ''black'' */
      m.s==j : // message m at source
        if there is a possible path p ({s, .... , d}) then
        /* Move(m, succ(j) ) action */
          move m to succ(j);
          /* m[j].pos = succ(j) */
          /* RG2 condition */
          RG(m, succ(j));
          /* m[succ(j)].colour = ''green'' */
        endif
```

```
     j != m.s and j != m.d :
      // message m at intermediate node and the given route is available
      if there is a possible path p' ({j, .... , d} is a suffix of p) then
      /* Move(m, succ(j) ) action */
        move m to succ(j);
        /* m[j].pos = succ(j) */
        /* RG2 condition */
        RG(m, succ(j));
        /* m[succ(j)].colour = ''green'' */
      else
          // given route is blocked
          /* GR2 condition */
          GR(m, j);
          /* m[j].colour = ''red'' */
          if there is a possible path p' ({j, .... , d} is not a
          suffix of p) then
            /* RG3 condition */
            RG(m, j);
            /* m[j].colour = ''green'' */
            /* Move(m, succ(j) ) action */
            move m to succ(j);
            /* m[j].pos = succ(j) */
            /* RG2 condition */
            RG(m, succ(j));
            /* m[succ(j)].colour = ''green'' */
          endif
      endif
    end case
end
```

A.4 *Updater* algorithm

```
Process Updater
  begin
    while true
    pick an edge e
                /* check affected route */
                for all route in routing table myRT
                  if myRT[iUpd].contains(e) then
                    if pathOK(e) then
                      /* route is available */
                      mark the route as "availP";
                    else
                      /* route is unavailable */
                      mark the route as "unAvailP";
                    endif
                  endif
  endif
```

A.5 Proof of interference: *Disrupter*, *Organizer*, and *Updater*

```
cobegin
  {n = numOfEdge}, {m = numOfMessage}
  {L[(edge, state)]}, {RT[(node, destination, path)]}
  disrupter:
  /* change state of edge into connect or disconnect */
    begin
      while (true)
       /* use random x to pick edge randomly */
        x := generateRandom(n);
        /* connected(x) */
        if L(x)='On' then
          /* CTD action */
          L(x) := 'Off'
          /* disconnected(x)*/
        else
          /* disconnected(x) */
          /* DTC action */
          L(x) := 'On'
          /* connected(x) */
    end
    {L(x)='On' || L(x)='Off'}
  //
  organizer:
  /* deliver a message to a destination */
  {(avail(p) || !avail(p)) && (m.color==color) && (m.pos==pos)}
  begin
  while true
    // pick a message m randomly
    /* based on current position, state and path p availability
        change the color of m and or move m
          m.color := newColor;
          m.pos := newPos */
    case
      m.s==m.pos and m.d==m.pos :
        /* empty path */
        /* no move action, message m is already at destination */
        /* RG action */
        RG(m, j);
        /* m.color = ''green'' */
        /* GB action */
        GB(m);
        /* m.color = ''black'' */
      m.d==m.pos : // message m reach the destination
        /* GB action */
        GB(m)
        /* m.color = ''black'' */
    m.s==m.pos : // message m at source
      /* avail(p) */
     if there is a possible path p ({s, .... , d}) then
       /* move(m, newpos ) action */
       move m to newpos;
       /* m.pos = newpos */
       /* RG action */
       RG(m);
       /* m.color = ''green'' */
     endif
    m.pos != m.s and m.pos != m.d :
     // message m at intermediate node and the given route is available
     /* avail(p') */
     if there is a possible path p' ({j, .... , d} is a suffix of p) then
       /* Move(m, newpos ) action */
       move m to newpos;
       /* m.pos = newpos */
       /* RG action */
       RG(m);
       /* m.colour = ''green'' */
     else
         // given route is blocked
         /* !avail(p) */
         /* GR action */
         GR(m);
         /* m.colour = ''red'' */
         /* avail(p') */
         if there is a possible path p' ({j, .... , d} is not a
         suffix of p) then
           /* RG action */
           RG(m);
           /* m.colour = ''green'' */
           /* Move(m, newpos ) action */
           move m to newpos;
           /* m.pos = newpos */
           /* RG action */
           RG(m);
           /* m.colour = ''green'' */
         endif
     endif
  end case
end
  {(avail(p) || !avail(p)) && (m.color ==newColor) && (m.pos==newPos)}
 //
 updater:
 /* update the available path based on state of edges */
  begin
    while (true)
     z := generateRandom(n);
     eState := L(z);
     while not end of RT
       if eState in p then
         check p;
```

```
          if all edge in p connected then
            /* UAv(p) action */
            avail(p) := true
            /* avail(p) */
          else
            /* UNAv(p) action */
            avail(p) := false
            /* !avail(p) */
   end
   {(L(x)='On' || L(x)='Off') && (avail(p) || !avail(p))}
coend
{(L(x)='On' || L(x)='Off') && (avail(p) || !avail(p))}
```

Appendix B

RULER specifications

B.1 *Model A* (Check model)

```
ruler Calculate(tIn:obs, pIn:obs, pIn2:obs, pIn3:obs, pIn4:obs, pIn5:obs, pTotOut: obs,
              pTotOut2: obs, pTotOut3:obs, pTotOut4:obs, pTotOut5:obs) {
 always startP {
 //message = 60
     pIn(ct:int), tIn(tc:int) -> Ok;
     pIn2(ct2:int), tIn(tc:int) -> Ok;
     pIn3(ct3:int), pIn4(ct4:int), tIn(tc:int) -> Ok;
     pIn5(ct5:int), tIn(tc:int) -> Ok;
  }
  state CountA(cct:int, cct2:int, cct3:int, cct4:int, cct5:int, ttc:int) {
    pIn(ct:int), tIn(tc:int) -> CountA(ct, cct2, cct3, cct4, cct5, tc),
      print("%EB = " + (ct/60.0)*100 + ", PathX = " + cct2 + ", %AllOrg = "
            + (cct3/(tc*1.0))*100 + ", %OrgT = " + (cct4/(tc*1.0))*100 + ", %DisT = "
            + (cct5/(tc*1.0))*100 + ", tc = " + tc) ;
    pIn2(ct2:int), tIn(tc:int) -> CountA(cct, ct2, cct3, cct4, cct5, tc),
      print("%EB = " + (cct/60.0)*100 + ", PathX = " + ct2 + ", %AllOrg = "
            + (cct3/(tc*1.0))*100 + ", %OrgT = " + (cct4/(tc*1.0))*100 + ", %DisT = "
            + (cct5/(tc*1.0))*100 + ", tc = " + tc) ;
    pIn3(ct3:int), pIn4(ct4:int), tIn(tc:int) -> CountA(cct, cct2, ct3, ct4, cct5, tc),
      print("%EB = " + (cct/60.0)*100 + ", PathX = " + cct2 + ", %AllOrg = "
            + (ct3/(tc*1.0))*100 + ", %OrgT = " + (ct4/(tc*1.0))*100 + ", %DisT = "
            + (cct5/(tc*1.0))*100 + ", tc = " + tc) ;
    pIn5(ct5:int), tIn(tc:int) -> CountA(cct, cct2, cct3, cct4, ct5, tc),
      print("%EB = " + (cct/60.0)*100 + ", PathX = " + cct2 + ", %AllOrg = "
            + (cct3/(tc*1.0))*100 + ", %OrgT = " + (cct4/(tc*1.0))*100 + ", %DisT = "
            + (ct5/(tc*1.0))*100 + ", tc = " + tc) ;
  }
  initials startP, CountA(0, 0, 0, 0, 0, 0);
}
ruler PathXCountTrace(totEv:obs, pOut:obs, pOut2:obs, pOut3:obs, pOut4:obs, pOut5:obs){
  observes RGr(obj, int, string, int), GB(obj, int), DTC(obj, int, int),
  CTD(obj, int, int), move(obj, int, int), write_Path(obj),
  GR(obj, int), waitM(obj, int), RoutingUpdate(int, int, string, int, obj, obj, string);
```

```
always Red{
   RoutingUpdate(id:int, s:int, x:string, d:int, y:obj, z:obj, p:string) -> redG(id, 1, p),
                                               print("message m " + id + "path = " + p);
   RGr(m:obj, idx:int, p:string, ps:int) -> Black(m, idx);
}
state redG(id:int, ctp:int, pm:string) {
  RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 <6 -> redG(id, ctp+1, pm);
  RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 >=6, pos==2 -> redG(id, ctp+1, pm),
      print("m" + idm + " potential livelock after using path " + rp + " " + (ctp+1) + " times");
  RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 >=6, pos==3 -> redG(id, ctp+1, pm),
      print("m" + idm + " potential livelock after using path " + rp + " " + (ctp+1) + " times");
  RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 >=6, pos==4 -> redG(id, ctp+1, pm),
      print("m" + idm + " potential livelock after using path " + rp + " " + (ctp+1) + " times");
  RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 >=6, pos!=2, pos!=3, pos!=4
                                                                  -> redG(id, ctp+1, pm);
}
 state Black(m:obj, idx:int) {
  GB(m:obj, idx:int) -> Ok;
}
  state GBt(GBcount:int) {
   GB(m:obj, idx:int) -> GBt(GBcount+1), print("GB = " + (GBcount+1)), pOut(GBcount+1);
}
state DisT(DisCount:int) {
  DTC(g:obj, a:int, b:int) -> DisT(DisCount+1), print("DisT = " + (DisCount+1)), pOut5(DisCount+1);
  CTD(g:obj, a:int, b:int) -> DisT(DisCount+1), print("DisT = " + (DisCount+1)), pOut5(DisCount+1);
}
state OrgT(OrgTCount:int) {
  RGr(m:obj, idx:int, p:string, ps:int) -> OrgT(OrgTCount+1),
                                     print("OrgTCount = " + (OrgTCount+1)), pOut4(OrgTCount+1);
  move(m:obj, idx:int, p:int) -> OrgT(OrgTCount+1), print("OrgTCount = " + (OrgTCount+1)),
                             pOut4(OrgTCount+1);
}
state AllOrT(AllTCount:int) {
  RGr(m:obj, idx:int, p:string, ps:int) -> AllOrT(AllTCount+1),
                                      print("AllTCount = " + (AllTCount+1)), pOut3(AllTCount+1);
  move(m:obj, idx:int, p:int) -> AllOrT(AllTCount+1), print("AllTCount = " + (AllTCount+1)),
                             pOut3(AllTCount+1);
  GR(m:obj, idx:int) -> AllOrT(AllTCount+1), print("AllTCount = " + (AllTCount+1)),
                     pOut3(AllTCount+1);
  waitM(m:obj, idx:int) -> AllOrT(AllTCount+1), print("AllTCount = " + (AllTCount+1)),
                        pOut3(AllTCount+1);
}
  state PathT(PathXCount:int) {
  write_Path(p:obj) -> PathT(PathXCount+1), print("PathXCount = " + (PathXCount+1)),
                    pOut2(PathXCount+1);
}

  state GRt(GRCount:int){
  GR(m:obj, idx:int) -> GRt(GRCount+1), print("GRt = " + (GRCount+1));
}
    state RedGt(RedGCount:int){
  RGr(m:obj, idx:int, p:string, ps:int) -> RedGt(RedGCount+1), print("RedGt = " + (RedGCount+1));
}
```

```
   state TotC(totCount:int){
    GR(m:obj, idx:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
    RGr(m:obj, idx:int, p:string, ps:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)),
                                        totEv(totCount+1);
    write_Path(p:obj) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
    waitM(m:obj, idx:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
    move(m:obj, idx:int, p:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)),
                               totEv(totCount+1);
    CTD(g:obj, a:int, b:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
    DTC(g:obj, a:int, b:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
    GB(m:obj, idx:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
    RoutingUpdate(id:int, s:int, x:string, d:int, y:obj, z:obj, p:string) -> TotC(totCount+1),
                                            print("TotC = " + (totCount+1)), totEv(totCount+1);
  }
    initials Red, GBt(0), DisT(0), PathT(0), OrgT(0), AllOrT(0), GRt(0), RedGt(0), TotC(0);
    forbidden Black;
    outputs totEv, pOut, pOut2, pOut3, pOut4, pOut5;
}

monitor{
  uses PXC: PathXCountTrace, C: Calculate;
  locals tEv(int), pgb(int), ppath(int), pall(int), porg(int), pdis(int), c0(int), c02(int),
                 c03(int), c04(int), c05(int);
  run (PXC(tEv, pgb, ppath, pall, porg, pdis) >> C(tEv, pgb, ppath, pall, porg, pdis, c0, c02, c03,
              c04, c05)) .
}
```

B.2 *Model B* (Check model)

```
ruler Calculate(tIn:obs, pIn:obs, pIn2:obs, pIn2a:obs, pIn3:obs, pIn4:obs, pIn5:obs, pTotOut: obs,
              pTotOut2: obs, pTotOut2a: obs, pTotOut3:obs, pTotOut4:obs, pTotOut5:obs) {
 always startP {
     pIn(ct:int), tIn(tc:int) -> Ok;
     pIn2(ct2:int), tIn(tc:int) -> Ok;
     pIn2a(ct2a:int), tIn(tc:int) -> Ok;
     pIn3(ct3:int), pIn4(ct4:int), tIn(tc:int) -> Ok;
     pIn5(ct5:int), tIn(tc:int) -> Ok;
  }
  state CountA(cct:int, cct2:int, cct2a:int, cct3:int, cct4:int, cct5:int, ttc:int) {
    pIn(ct:int), tIn(tc:int) -> CountA(ct, cct2, cct2a, cct3, cct4, cct5, tc),
      print("%EB = " + (ct/60.0)*100 + ", PathX = " + (cct2/55.0) +
      ", %Updater = " + (((cct2/55.0) + (cct2a/55.0))/((ct + cct3 + cct5 + (((cct2/55.0)
                    + (cct2a/55.0))))*1.0))*100 +
      ", %AllOrg = " + (cct3/((ct + cct3 + cct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 +
      ", %OrgT = " + (cct4/((ct + cct3 + cct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 +
      ", %DisT = " + (cct5/((ct + cct3 + cct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 + ",
                  tc = " + tc) ;
    pIn2(ct2:int), tIn(tc:int) -> CountA(cct, ct2, cct2a, cct3, cct4, cct5, tc),
      print("%EB = " + (cct/60.0)*100 +
      ", PathX = " + (ct2/55.0) +
       ", %Updater = " + (((ct2/55.0) + (cct2a/55.0))/((cct + cct3 + cct5 + (((ct2/55.0)
                     + (cct2a/55.0))))*1.0))*100 +
       ", %AllOrg = " + (cct3/((cct + cct3 + cct5 + (((ct2/55.0) + (cct2a/55.0))))*1.0))*100 +
       ", %OrgT = " + (cct4/((cct + cct3 + cct5 + (((ct2/55.0) + (cct2a/55.0))))*1.0))*100 +
       ", %DisT = " + (cct5/((cct + cct3 + cct5 + (((ct2/55.0) + (cct2a/55.0))))*1.0))*100 + ",
                   tc = " + tc) ;
    pIn2a(ct2a:int), tIn(tc:int) -> CountA(cct, cct2, ct2a, cct3, cct4, cct5, tc),
       print("%EB = " + (cct/60.0)*100 +
       ", PathX = " + (cct2/55.0) +
       ", %Updater = " + (((cct2/55.0) + (ct2a/55.0))/((cct + cct3 + cct5 + (((cct2/55.0)
                     + (ct2a/55.0))))*1.0))*100 +
       ", %AllOrg = " + (cct3/((cct + cct3 + cct5 + (((cct2/55.0) + (ct2a/55.0))))*1.0))*100 +
       ", %OrgT = " + (cct4/((cct + cct3 + cct5 + (((cct2/55.0) + (ct2a/55.0))))*1.0))*100 +
       ", %DisT = " + (cct5/((cct + cct3 + cct5 + (((cct2/55.0) + (ct2a/55.0))))*1.0))*100 + ",
                   tc = " + tc) ;
    pIn3(ct3:int), pIn4(ct4:int), tIn(tc:int) -> CountA(cct, cct2, cct2a, ct3, ct4, cct5, tc),
       print("%EB = " + (cct/60.0)*100 +
       ", PathX = " + (cct2/55.0) +
       ", %Updater = " + (((cct2/55.0) + (cct2a/55.0))/((cct + ct3 + cct5 + (((cct2/55.0)
                     + (cct2a/55.0))))*1.0))*100 +
       ", %AllOrg = " + (ct3/((cct + ct3 + cct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 +
       ", %OrgT = " + (ct4/((cct + ct3 + cct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 +
       ", %DisT = " + (cct5/((cct + ct3 + cct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 + ",
                   tc = " + tc) ;
    pIn5(ct5:int), tIn(tc:int) -> CountA(cct, cct2, cct2a, cct3, cct4, ct5, tc),
       print("%EB = " + (cct/60.0)*100 +
       ", PathX = " + (cct2/55.0) +
       ", %Updater = " + (((cct2/55.0) + (cct2a/55.0))/((cct + cct3 + ct5 + (((cct2/55.0)
                     + (cct2a/55.0))))*1.0))*100 +
       ", %AllOrg = " + (cct3/((cct + cct3 + ct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 +
       ", %OrgT = " + (cct4/((cct + cct3 + ct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 +
       ", %DisT = " + (ct5/((cct + cct3 + ct5 + (((cct2/55.0) + (cct2a/55.0))))*1.0))*100 + ",
                   tc = " + tc) ;
   }
  initials startP, CountA(0, 0, 0, 0, 0, 0, 0);
}

ruler PathXCountTrace(totEv:obs, pOut:obs, pOut2:obs, pOut2a:obs, pOut3:obs, pOut4:obs, pOut5:obs){
  observes RGr(obj, int, string, int), GB(obj, int), DTC(obj, int, int),
  CTD(obj, int, int), move(obj, int, int), UAv(obj, int), UNAv(obj, int), GR(obj, int),
  GR(obj, int), waitM(obj, int), writeTheRoute(obj, int, string);

  always Red{
    writeTheRoute(m:obj, idx:int, p:string) -> redG(idx, 1, p), print("message m " + idx
                                                                + "path = " + p);
    RGr(m:obj, idx:int, p:string, ps:int) -> Black(m, idx);
  }
  state redG(id:int, ctp:int, pm:string) {
   RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 <6 -> redG(id, ctp+1, pm);
   RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 >=6, pos==2 -> redG(id, ctp+1, pm),
      print("m" + idm + " potential livelock after using path " + rp + " " + (ctp+1) + " times");
   RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 >=6, pos==3 -> redG(id, ctp+1, pm),
      print("m" + idm + " potential livelock after using path " + rp + " " + (ctp+1) + " times");
   RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 >=6, pos==4 -> redG(id, ctp+1, pm),
      print("m" + idm + " potential livelock after using path " + rp + " " + (ctp+1) + " times");
   RGr(mp:obj, idm:int, rp:string, pos:int), idm == id, ctp+1 >=6, pos!=2, pos!=3, pos!=4
                                                             -> redG(id, ctp+1, pm);
  }
   state Black(m:obj, idx:int) {
   GB(m:obj, idx:int) -> Ok;
  }
   state GBt(GBcount:int) {
    GB(m:obj, idx:int) -> GBt(GBcount+1), print("GB = " + (GBcount+1)), pOut(GBcount+1);
  }
   state DisT(DisCount:int) {
   DTC(g:obj, a:int, b:int) -> DisT(DisCount+1), print("DisT = " + (DisCount+1)), pOut5(DisCount+1);
   CTD(g:obj, a:int, b:int) -> DisT(DisCount+1), print("DisT = " + (DisCount+1)), pOut5(DisCount+1);
  }
   state UpdaV(UpdCount:int) {
   UAv(h:obj, k:int) -> UpdaV(UpdCount+1), print("UpdaV = " + (UpdCount+1)), pOut2(UpdCount+1);
  }
  state UpdNAv(UNAvCount:int) {
   UNAv(h:obj, k:int) -> UpdNAv(UNAvCount+1), print("UpdNAv = " + (UNAvCount+1)),
                      pOut2a(UNAvCount+1);
  }
  state OrgT(OrgTCount:int) {
   RGr(m:obj, idx:int, p:string, ps:int) -> OrgT(OrgTCount+1),
                                   print("OrgTCount = " + (OrgTCount+1)), pOut4(OrgTCount+1);
   move(m:obj, idx:int, p:int) -> OrgT(OrgTCount+1), print("OrgTCount = " + (OrgTCount+1)),
                              pOut4(OrgTCount+1);
  }
  state AllOrT(AllTCount:int) {
   RGr(m:obj, idx:int, p:string, ps:int) -> AllOrT(AllTCount+1),
                                     print("AllTCount = " + (AllTCount+1)), pOut3(AllTCount+1);
   move(m:obj, idx:int, p:int) -> AllOrT(AllTCount+1),
                              print("AllTCount = " + (AllTCount+1)), pOut3(AllTCount+1);
   GR(m:obj, idx:int) -> AllOrT(AllTCount+1), print("AllTCount = " + (AllTCount+1)),
                      pOut3(AllTCount+1);
   waitM(m:obj, idx:int) -> AllOrT(AllTCount+1), print("AllTCount = " + (AllTCount+1)),
                      pOut3(AllTCount+1);
  }

   state GRt(GRCount:int){
   GR(m:obj, idx:int) -> GRt(GRCount+1), print("GRt = " + (GRCount+1));
  }
      state RedGt(RedGCount:int){
   RGr(m:obj, idx:int, p:string, ps:int) -> RedGt(RedGCount+1), print("RedGt = " + (RedGCount+1));
  }
   state TotC(totCount:int){
   GR(m:obj, idx:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
   RGr(m:obj, idx:int, p:string, ps:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)),
                                     totEv(totCount+1);
   UAv(h:obj, k:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
   UNAv(h:obj, k:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
   waitM(m:obj, idx:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
```

```
    move(m:obj, idx:int, p:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)),
                                totEv(totCount+1);
    CTD(g:obj, a:int, b:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
    DTC(g:obj, a:int, b:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
    GB(m:obj, idx:int) -> TotC(totCount+1), print("TotC = " + (totCount+1)), totEv(totCount+1);
 totEv(totCount+1);
  }
    initials Red, GBt(0), DisT(0), UpdaV(0), UpdNAv(0), OrgT(0), AllOrT(0), GRt(0), RedGt(0),
            TotC(0);
    forbidden Black;
    outputs totEv, pOut, pOut2, pOut2a, pOut3, pOut4, pOut5;
}
monitor{
  uses PXC: PathXCountTrace, C: Calculate;
  locals tEv(int), pgb(int), puav(int), punav(int), pall(int), porg(int), pdis(int), cO(int),
                 cO2(int), cO2a(int), cO3(int), cO4(int), cO5(int);
  run (PXC(tEv, pgb, puav, punav, pall, porg, pdis) >> C(tEv, pgb, puav, punav, pall, porg, pdis, cO,
                 cO2, cO2a, cO3, cO4, cO5)) .
}
```

B.3 *Model A*: Lemma 1 (eventually green)

```
ruler pathXE(pIn:obs, pTotOut: obs) {
  always startP {
    pIn(id:int, pt:string, ct:int), !pIn(id, " ", ct) ->
      pTotOut(id, pt, ct+1), print("this path " + pt + " is up for " + (ct + 1)
         + " times to be used by message m" + id);
  }
  initials startP;
  outputs pTotOut;
}

ruler disrupter(pOut:obs) {
  observes RoutingUpdate(int, int, string, int, obj, obj, string),
         RGr(obj, int, string), move();
  always ru {
   RoutingUpdate(id:int, s:int, x:string, d:int, y:obj, z:obj, p:string)
     {: !redG(id, 1, p) ->
        pOut(id, p, 0), print("id message = " + id + " path = " + p), redG(id, 1, p);
        redG(id, ct:int, p) -> pOut(id, p, ct), redG(id, ct+1, p); :}
  }
  state redG(id:int, cm:int, pm:string) {
    RGr(mp:obj, idm:int, rp:string), idm == id ->
       print("m" + idm + " eventually green after using path " + rp);
  }

  initials ru;
  forbidden redG;
  outputs pOut;
  }
  monitor {
  uses D : disrupter, P : pathXE;
  locals path(int, string, int), r0(int, string, int);
  run (D(path) >> P(path, r0)) .
}
```