Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2018

Project Title:	**Formal verification of POETS applications using TLA+**
Student:	**Muhammad Affan Qureshi**
CID:	**00985230**
Course:	**EEE4**
Project Supervisor:	**Dr. David B. Thomas**
Second Marker:	**Dr Thomas J. W. Clarke**

Abstract

POETS (Partially-Ordered Event-Triggered Systems) is a novel, massively parallel, computer architecture that can provide significantly better performance for certain applications compared to existing implementations on conventional architectures. Its concurrent and asynchronous characteristics make it difficult to develop and debug applications for it. This project explores TLA+ and the TLC model checker as suitable tools for formally specifying and verifying POETS applications. Two existing applications which had questions about their correctness were specified in TLA+. The bugs discovered while specifying and model checking them are discussed and alternative algorithms with the correct behaviour are presented. The usefulness of the specifications and their verification with respect to the application problem size is evaluated. The conclusion drawn is that while the verification of real world problem sizes is not feasible, bugs can still be detected using smaller problem sizes. Furthermore, the process of specifying the algorithms in TLA+ itself greatly helps in understanding them better. Lastly, the process of specifying applications in TLA+ is highlighted and a potential method of automatically converting existing applications to TLA+ specifications is suggested.

Acknowledgements

I would sincerely like to thank my project supervisor, Dr. David Thomas, for his continuous help and support throughout the year. I would like to thank all my teachers at the university greatly helped me in acquiring the necessary skills. I would also like to thank my family for supporting me through my years at university. And finally I would like to thank my friends for making my time here at Imperial very enjoyable.

Contents

List of Figures

List of Tables

1 Introduction

While the performance upgrades achieved due to Moore's Law[1] have slowed down in recent years, they have been offset by the development of multi-core processors and the adoption of concurrent computing methodologies. Due to the significant reduction in the cost of processor-cores new massively parallel architectures are emerging for an assortment of public domains. One such architecture is POETS (Partially-Ordered Event-Triggered Systems). The idea behind POETS is to have an extremely large number of asynchronous cores that communicate with each other through messages of a few bytes using a hardware based parallel communications infrastructure. It is an event triggered architecture, meaning that each core only performs calculations in response to a trigger, which in this case is the receipt of a message, and then goes back into a low powered state. This makes the system very energy efficient. While not being a general purpose architecture it can be used for a diverse set applications which can exploit its massively parallel nature. [1]

Being a new architecture, POETS does not have any mature development tools or techniques. Hence, new concurrent, event-based algorithms need to be developed that can run on and benefit from the architecture's main characteristic: its intrinsic parallelism. Concurrent algorithms are notoriously hard to reason about and it is easy to miss obscure bugs that might result in race conditions or other similar problems due to concurrency. It is also not possible to generate test cases for all possible scenarios while checking for code correctness. Using formal methods for specifying and verifying the algorithms mitigates this to a large extent by finding and fixing complicated bugs at an earlier development stage. TLA+ is one such formal method that has been successfully used for verifying distributed systems [2]. Since POETS has a lot of similarities to distributed systems, it is believed that TLA+ will be suitable for specifying and verifying POETS applications. This reports investigates this claim by specifying and verifying a few POETS applications using TLA+ and exploring how it can be used for other applications as well.

[1]The number of transistors on a chip doubles roughly every two years.

1.1 Report Structure

The structure of the report is as follows:

- The **Background** chapter gives an overview of the concepts that are relevant to this project and discusses work done in related fields.
- The **Aims and Objectives** chapter defines the hypothesis that will be investigated in this report and provides a roadmap for tasks that will help in the investigation.
- The **Analysis and Design** chapter analyses different criteria that would make the process of specifying and verifying POETS applications clearer and justifies the design choices made in this project.
- The **Implementation** chapter describes the process of specifying and verifying three different POETS applications.
- The **Evaluation** chapter provides evidence in support of the hypothesis defined in **Aims and Objective**.
- The **Further Works** chapter elaborates on how the research done in this project can be extended upon.
- The **Conclusion** chapter gives an overview of the achievements of this project.

2 Background

2.1 Distributed Systems

Any system comprising of independent components, located on multiple networked computers that communicate and interact with each other through messages only to achieve a common goal, can be considered a distributed system [3]. The fundamental properties of such systems, described in detail in the book Distributed Systems by G. Coulouris *et al.* [3], are summarised below.

- **Concurrency**: Different component in the system will be doing different tasks at the same time. This also allow the addition of new components to further scale the system.
- **No global clock**: Since different components can only communicate through messages there are limitations to how accurately different components can synchronize their clocks.
- **Independent failures**: Due to the increase in the complexity of the systems the chances of error are also increased. A component within the system can fail or a message can be dropped due to problems in the communication network. Therefore the overall system must be designed in such a way as to keep on working correctly even though there are some independent points of failure.

Although no global clock exists in a distributed system as mentioned above, it can still be categorized as a synchronous distributed system if there are known bounds on the execution time of a process step, the time delay between sending and receiving a message, and the rate of drift between the local clock of each process and the real time [4]. In contrast, a distributed system is considered asynchronous if there are absolutely no timing constraints on the properties mentioned above.

2.2 POETS

POETS, Partially Ordered Event-Triggered Systems, is a new massively parallel architecture currently being developed mainly as an alternative platform for use in a wide variety of industrial and scientific applications which are traditionally run on supercomputers. The system is supposed to have an extremely large number of cores ($> 10^7$) that are interconnected through a hardware based communication infrastructure. The only communication between individual cores is through small sized messages of a few bytes [1]. Being an event-triggered systems, each core only performs calculations in response to the receipt of messages, sends any new messages if required and then goes back into a low powered state [1].

The POETS architecture is built upon the concepts of Neuromorphic computing [5] [6] which is significantly different to architectures like Von Neumann [7] on which most of the present general purpose computer are based [8]. POETS is very well suited for applications where the underlying mathematics can be represented as a graph with asynchronous interactions between nodes [1]. However it is not possible to port algorithms of such applications developed for conventional architectures to POETS. New algorithms need to be developed using the underlying mathematics of the application that can exploit the structural capabilities of POETS [8].

This project explores the formal verification of POETS applications. The properties of POETS described in [1] and [6] that influence the structure and semantics of its applications are summarised below. The detailed hardware and software design of the architecture of POETS and advantages of the architecture can also be found in the above mentioned papers and more available at [9]. However these are not discussed here as they are not needed to follow the work done in this project.

- **Asynchronous**: The system has no central clock to use for synchronization. Cores only perform computations in response to triggers. hence there is no need of synchronization with other cores.
- **No Global State**: The system memory and hence the system state is distributed among cores. Each core only has access to a partial state that is available in its local memory
- **Communication via short messages**: Cores communicate with each other through hardware brokered packets. These packets, although guaranteed to reach their destination, can take arbitrarily long to do so.

2.3 TLA+

TLA+ is a general purpose, high level, formal specification language mainly used to specify and verify concurrent and distributed hardware and software systems [10]. TLA+ is based on the principles of set theory, first-order logic and Temporal Logic of Actions [11]. Temporal Logic of Actions [12] extends temporal logic [13] by introducing Invariance under stuttering, Temporal existential quantifiers and taking both the state predicate and action formulas as atomic formulas [14].

The book *Specifying Systems* by Leslie Lamport [15] gives a great introduction for writing TLA+ specifications for computer systems. The basic principles of TLA+ are described here.

A typical TLA+ specification defines an initial predicate that specifies the initial state of the system, a next state actions which specifies all the possible steps in the behaviour of the system and the liveness properties of the systems. The next state actions are specified as a disjunction of all the possible next state actions. The syntax would be of the form:

$$\begin{aligned} NextState \quad == \quad & \lor\ Action1 \\ & \lor\ Action2 \\ & \lor\ Action3 \end{aligned}$$

And the possible atomic actions are specified as the conjunction of conditions required to perform the certain action and the changes to variables representing the state. Being a formal method there no chance of ambiguity hence all the variables of the state need to be present in the action step. If a particular variable is to be left unchanged it should be explicitly mentions. The syntax for a possible action would be of the form:

$$\begin{aligned} Action \quad == \quad & \land\ Condition1 \\ & \land\ Condition2 \\ & \land\ var1' \ = \ var1 * 2 \\ & \land\ var2' \ = \ IF\ a > b\ THEN\ 1\ ELSE\ 2 \\ & \land\ UNCHANGES\ << var3, var4 >> \end{aligned}$$

Many different data structures are available for defining variables and constants with the primary ones being sets, functions, records and tuples. Sets are defined by comma separated items enclosed in curly braces ($var == a, b, b$). Tuples are defined as comma separated items enclosed in "<<" and ">>" ($var\ == << a, b, c >>$). The elements can be accesses individually using indexes which start from 1 ($var[2] = b$). For further details on specifying systems using TLA+ see [15].

2.3.1 TLC Model Checker

TLC [16] is the model checker used for checking TLA+ specifications. TLC can be used to check for both safety and liveness properties. It verifies the specification by generating all possible reachable states in the system. While the model checker can perform depth first search using the `- simulation` options it uses breath-first technique by default to search the state space. Hence the error-trace generated is always of minimum length if a safety property in violated [10].

2.4 Related Works

While specific work for verifying POETS applications using TLA+ has not been done before, TLA+ has been used for the verification of several concurrent and distributed systems. The most notable of these works is [2].

3 Aims and Objectives

The scope of this project can be broken down into investigations of three hypotheses. These hypotheses are:

1. TLA+ is suitable for formally specifying POETS applications.
2. The specified applications can be verified using the TLC model checker.
3. Existing POETS applications, written in XML and C++, can be mechanically converted to TLA+ specifications.

The syntax of TLA+ is based on formal logic and mathematics, making it extremely expressive. It can be used to formally specify almost any discrete system [15]. Therefore, it is entirely possible to describe POETS applications in TLA+. What we need to investigate in the first hypothesis is whether the TLA+ specification will be useful, which includes questions like: can the specification be actually converted in a real world application under the constraints of the provided DSLs (Domain Specific Languages discussed in the background for POETS), does the specification help in detecting bugs, etc.

The TLC model checker cannot be used on every TLA+ specification. The specifications need to have a certain form; for example, they should not include any infinite sets. Furthermore, it is also possible that the model checker does run but does not provide any useful results. This can be due to some properties of the application not allowing it to terminate (for example, being stuck in a loop) or due to the problem being just too big for the model checker to search the entire search space within a reasonable time. These problems and several others result in the model checkers continuing to run without providing any output. This will be explored in the second hypothesis.

While the benefits gained from formally specifying POETS applications can justify its use, there will still be some barriers for developers to actually use them. They would need to learn a new language which is significantly different in its syntax and semantics to common programming languages and paradigms. Moreover, they would also need to specify existing applications in TLA+ to formally verify them. The third hypothesis investigates mechanical means of converting existing applications to TLA+ specifications. If such means exist, they would have the potential of being automated. This would significantly bridge the gap between application specifications and formal TLA+ descriptions, further encouraging the use of formal methods.

3.1 Roadmap

Since there are no required deliverables for the project, a roadmap needs to be defined with clear tasks that when completed would bring us closer to proving or disproving the hypotheses stated above.

To start with, an existing application that is known to have a correct POETS implementation will be specified in TLA+. The specification will be model checked to verify that it has the same behaviour as the original implementation. As all POETS applications have a similar structure in the sense that each of them has several asynchronous processes working concurrently and communicating through messages, the specification will provide us with a general structure for specifying other similar applications.

Once the application specification is verified to be correct we will move on to more complex applications which have not been proven to be correct yet. The plan is to specify these applications in TLA+ and model check them. If any bugs corresponding to the algorithm are discovered in the specification, alternate algorithms will be considered. Alternate algorithms that have been verified to work correctly will be provided.

If everything goes as planned the tasks mentioned above should be enough to prove the first and second hypotheses. As such, we will be able to present a case where we initially had questions about an application's correctness, we specified the application in TLA+, discovered bugs through model checking, and then provided an alternate, correct implementation. If time allows, this process will be repeated for another application to strengthen our case.

For proving the third hypothesis, the plan is to get familiar with the algorithm language PlusCal which can be translated to a TLA+ specification, specifying an existing application in PlusCal, and then exploring if the process can be mechanized. PlusCal is chosen because it has a syntax similar to common imperative languages like C/C++, as opposed to TLA+, which is completely declarative.

4 Analysis and Design

Before we can start specifying POETS applications in TLA+ several design choices need to be considered. The choices greatly influence the complexity of the specifications, feasibility of model checking and the properties that can be verified.

4.1 Semantics of POETS

The fundamental characteristics of POETS as discussed in Section 2.2 are:

- Concurrent and asynchronous event-triggered processes, corresponding to individual cores, that only have access to a small local memory.
- Non deterministic order of messages resulting from arbitrary time delay in message delivery.
- Fault tolerance in the event of processes prematurely terminating or computing wrong results due to faulty cores or corrupted local memory. [2]

A complete formal specification must capture all the above characteristics. Since the algorithms developed for POETS applications need to be concurrent and asynchronous by design, these characteristics are intrinsically present in those applications' TLA+ specifications. The 'random order of messages' property will be explicitly captured in the specifications. The plan is to have several messaging protocols specified with the same interface so that they can be used interchangeably to test the application under different scenarios. Furthermore, this would also allow others to include these protocols in their own application specifications without much effort. The fault tolerance characteristic of the system was not explored here due to several reasons. Firstly, it would have significantly increased the complexity of the specifications. Secondly, the underlying algorithm needs to be correct before any benefits from fault tolerance can be achieved and this is what we are trying to verify here. Thirdly, while the fault tolerance characteristic would certainly have made the TLA+ specifications more complete, it is not needed for proving the hypotheses stated in Section 3.

[2] As message delivery is guaranteed by the architecture, dropped messages are not considered.

4.2 Levels of Abstraction

Another aspect to consider while specifying an application in TLA+ is its level of abstraction. The abstraction level of a system is very hard to quantify and the criteria for defining it differs from one problem domain to another. For the purposes of POETS and its applications, two criteria can be used to define the level of abstraction of the system.

1. Defining the abstraction level in terms of its complete implementation. A specification can be considered high level if it only captures the semantics of the application without any representation of the underlying system on which it is implemented. Conversely, a low level specification will correspond to the application's implementation on a certain architecture or system. There is a whole spectrum of levels in between; for instance, a specification could correspond to the application's implementation on a certain class of systems. The characteristics defined in a higher level specification must always be satisfied by the lower level one while the opposite does not need to be true.

2. Defining the abstraction level in terms of the application properties specified. A high level specification will be one where the complete application is specified while a low level one will only have certain properties of the application specified. Similar to 1., there is whole spectrum of levels in between; for instance, a specification could have a certain group or category of properties specified.

Choosing a suitable level of abstract is extremely important and often a difficult decision. A low level specification will significantly increase the problem size, making it infeasible to model check while a high level specification might mask some underlying problems in the application specific to the system implementation. A possible solution to mitigate these difficulties can be to specify separate parts of the application at a low level to verify them individually, and also have a higher level specification of the complete application that verifies that all the parts work together as expected. The abstraction levels of applications specified in this report will be discussed in Section 5.

4.3 State Space of Model

The state space of a specification model is highly dependent on the problem size. POETS consists of a massive number ($> 10^7$) of cores that correspond to different processes working and communicating concurrently. Trying to model check a specification with that many concurrent processes is impossible due to the combinatorial explosion of the total number of reachable states. As an illustration a system consisting of 100 processes, where each process only performs a single action and then terminates, would result in a search space[3] of 100! which is approximately equal to 10^{157}. To understand the sheer magnitude of this, consider that the total number of atoms in the known universe is estimated to be somewhere between 10^{78} and 10^{82}. POETS applications will have a far greater number of processes where each process can perform various possible actions several times. For a simple model, where the total possible next states are assumed to be a constant b and the total number of actions performed by a system (depth of graph) is assumed to be d, the total number of states explored is $\mathcal{O}(d * b^d)$, as shown in equation 1. However, in most cases b will significantly vary from one state to another. As such, equation 1 can only be used to give a very rough estimate of the model's complexity.

$$\begin{aligned} TotalStates &= \sum_{i=0}^{d} b^i < \sum_{i=0}^{d} b^d = d\ b^d \\ &= \mathcal{O}(d * b^d) \end{aligned} \tag{1}$$

Reducing the search space of the model is necessary if we want to be able obtain any useful results from the model checker. One possible way of reducing the search space is to specify the application at a higher level. Another method is to add constraints to the randomness of message delivery; this lowers the number of possible next states. Alternatively, certain nodes or process of the application can be prioritized. Even if all the methods mentioned above are used, model checking real world problem sizes will still be infeasible. However, as will be shown in Sections 5 and 6, using smaller problem sizes for applications with as low as 5 or 6 processes can still yield useful results, allowing us to to find bugs in the specification.

[3]The state space here would be the total possible permutations (ordering) in which the processes perform their actions.

5 Implementation

5.1 Gals Heat

Gals Heat was the first POETS application specified in TLA+ for this project. The application is used to model a solution for the discrete heat equation. The implementation for POETS maps the discrete physical space under consideration to asynchronous processes corresponding to individual cores in hardware, and models the transfer of heat with progression in time. Each process corresponding to a physical point in space is initialized at time $t = 0$. A process progresses to time t+1 and updates its value only after it has received the messages for time step t from all its neighbours.

While considering the abstraction level at which to specify the application it was decided that only the progression in time of each process needed to be captured, ignoring the mathematical calculation performed for computing other properties of the process. The calculation, while resulting in a more complete specification of the application, would not have benefited its verification. Rather, it would needlessly have increased the model's complexity. This is because only the progression of each process in time is needed to verify that the interaction between neighbouring nodes/process is correct. Additionally, the messaging infrastructure of POETS was also abstracted away, yielding a higher level specification where only the asynchronous, concurrent and even-triggered characteristics were captured.

The complete specification for Gals Heat is given in Appendix A.1. The specification was model checked and no problems were found. This was expected as the application's implementation, which can be found in [17], was known to be correct. The purpose of specifying it was to get familiar with TLA+ and obtain a general structure on which other POETS applications could also be specified.

5.2 Gals Izhikevich

The Gals Izhikevich application models the spiking and bursting behavior of neurons present in the cerebral cortex using the methods defined in [18]. The POETS implementation maps individual neurons onto processing cores. A neuron can progress to the next time step if it has received as many messages as the number of its input neighbours and the neuron can also fire during the progression depending on the state of its internal variables. The main difference between this and Gals Heat is that Gals Izhikevich does not require a two way connection with its neighbours, resulting in separate input and output neurons. However the structure of the neural networks used for Gals Izhikevich is also a strongly connected directed graph. The actual implementation of the application is given in [17].

Similar to Gals Heat only the progression of a neuron's times with respect to its neighbours was captured in the TLA+ specification. The reasons behind this are the same as before. Including other properties in the specification would have only increased the complexity of the model without significantly benefiting the verification process. The random message delivery characteristics of POETS were not captured in the initial specifications. This is because our aim here was to investigate whether the existing algorithm was correct, and if not, to devise and implement fixes. Once the underlying algorithm was verified to be correct, only then would there be a benefit to the added complexity of messaging protocols, because only then would the algorithms be tested for bugs that result from lower level characteristics. Therefore, this section only considers the higher level specification and once it is confirmed to be correct, only then are the messaging protocols implemented.

The specification is given in Appendix A.2. While model checking the specification the model deadlocked before all the nodes had reached the defined maximum time. This suggested that there was a bug in the application. The statement $!.c = 0$ for the *IF* condition case where $a = n$ was the cause of this error. The specification was fixed simply by deleting that statement. The fixed specification was model checked and it was found that all the neurons did reach the defined maximum time. Another thing that was considered was whether a new algorithm could be developed where a neuron could only progress to the next time step once it had received messages for the current time step from all of its input neighbours. The original algorithm only considers the total number of messages received, ignoring the time steps of the messages.

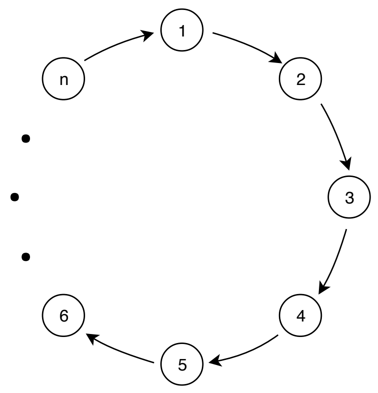
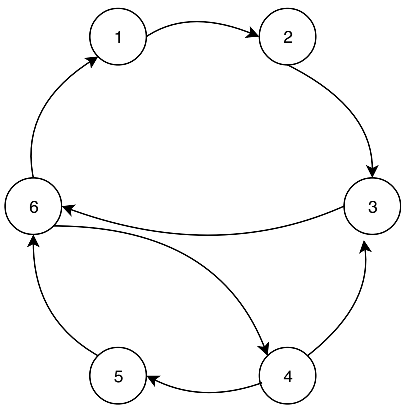


Figure 1: Circular network resulting in the worst case memory requirement.

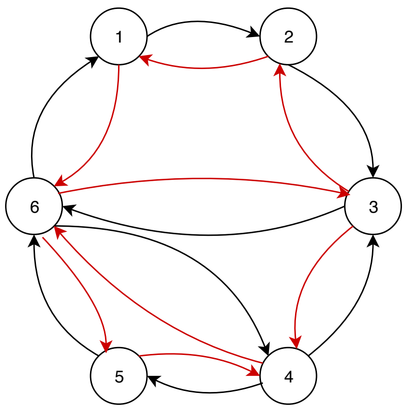
5.2.1 Alternate Algorithm

The simplest method to implement the alternative algorithm is to separately store the message count for each time step. Although this can be specified and verified the implementation is not feasible for actually running on the architecture. This is because there is a huge memory requirement dependent on the defined maximum time as each node will need to store the message count for all the time steps from 0 to *MaxTime*. This is not possible as each node only has access to a limited local memory. The structure of the application's network suggests that the actual memory requirement can be less than the one given above. Each neuron needs to receive a message for time t from all its input neighbours (implying that all the neighbours are at time t or higher) before progressing to time $t+1$. Hence, the input neighbours cannot be more than one time step behind but can be as far ahead as possible. From this, it can be deduced that worst case memory requirement is present in a circular network where the memory requirement becomes equal to the number of nodes. Figure 1 shows such a network where if the first neuron stays at time 0 and the rest of the neurons progress as far as possible, then the n^{th} neuron will eventually reach time $n-1$, hence making the maximum possible time difference between two neighbouring neurons $n-1$.

The memory requirement for the above scenarios is $min(MaxTime, No.of Neurons)$. Both of these will vary with different network models and will generally be a lot bigger than the available local memory. The challenge is to limit the memory requirement to some constant k in order to make it suitable for running on POETS. For this to be implemented a limit needs to be imposed on the neurons to not be more that k time steps ahead of their output neurons. This requires each neuron to



(a) Original Configuration



(b) Configurations with backwards messages

Figure 2: Configurations used for testing.

keep track of its output neurons. Hence, each neuron needs to send messages to its input neighbours in addition to the existing messages that are sent to the output neighbours. The additional messages are illustrated in Figure 2. A new variable that stores the time difference between a neuron and its output neighbours and the messages in the other direction are introduced in the specification, which when model checked gives the correct results. The specification for this was implemented while also incorporating the messaging protocols and is given in Appendix A.3.

It should be noted that while the specification stores the state of the complete system in a single variable, it is regarded as local and each neuron only accesses its own state. However, for the above specification, the messages were assumed to be instantaneous and hence the states of the neighbours were changed in the same process action. The following chapter introduces messaging protocols which makes the access to state completely local.

5.3 Messaging Protocols

Although POETS assumes random order of messages, a few other protocols were also specified to check for interesting results in state transitions while also reducing the total search space of the model. The expansion of the state space when considering all possible orderings of received messages has combinatorial complexity which is further worsened by the addition of new messages in the channel. Therefore, other protocols were also implemented to mitigate this effect while still giving useful results.

Once the high level algorithm for Gals Izhikevich was verified, the next step was to include the random messaging aspect of POETS in the specification and verify that it still worked. Although the primary aim was to test the Gals Izhikevich specification with different messaging protocols they were specified in a way that allowed their use in different applications' TLA+ specifications. The messaging protocols were specified as separate independent modules with the same outer interface but different internal implementations. The guide for using these protocols in other POETS applications is given in Section 9.1. The following messaging protocols were specified:

- **Random Messages**: There is no defined order in which the messages are received. This results in the model checking for all possible orders in which the messages could be received. This has combinatorial complexity, and since the model also has to check for other processes launching and sending more messages, the complexity of the state space is further increased. The channel was implemented as a set in which messages can be added or removed. As this is the most general implementation, it covers all possible state transitions including the ones searched by the other protocols mentioned below. The TLA+ specification for the protocol is given in Appendix A.4.

- **Choose Random Message**: This was added to avoid the combinatorial complexity of the above implementation while still keeping the random nature of message delivery. Here, a single message from all the possible messages is delivered and other possibilities are ignored. While it does significantly reduce the search space, there is also a chance of missing a certain sequence in which messages are received that would result in an error. It should be noted that the CHOOSE expression $CHOOSEx \in s : f$ used to select a single messages does not give a truly random result. The expression chooses an arbitrary value

x from a given set s that satisfies the condition f. However this arbitrary value will always be the same if s and f are the same.

- **First In Last Out**: FILO, as the name suggests, sends messages in the reverse order in which they were received. This can be thought of as the extreme case in which old messages are the slowest to be received and new messages are the fastest to be received. This causes a wider difference in the progression of different processes. The channel for this protocol was specified as a sequence.
- **First In First Out**: FIFO is the complete opposite of FILO. The messages are received in the same order in which they were sent. This is the behaviour of the general Gals Izhikevich specification which is mentioned in Section 5.2 and which abstracts away the messaging protocols. The difference here is that the process of sending and receiving messages is separated, allowing race conditions (due to other processes starting while the messages are still in the channel) to be checked for.

All of the above protocols were tested with the Gals Izhikevich specification and the observations are discussed in Section 6.2. The Gals Izhikevich specification for random messages is given in Appendix A.3.

5.3.1 Priority Nodes

Another possible modification to reduce the search space is to prioritize certain processes (neurons in the case of Gals Izhikevich) to send or receive messages before other processes. Since this requires the knowledge of processes that need to be prioritized, it cannot be implemented as an independent module and has to be included within the application specification. However, the changes needed are not very significant and would not require major changes to process descriptions in most cases. For Gals Izhikevich the process specification was not changed at all. Instead, only the *next state* description was changed. The TLA+ specification for Gals Izhikevich with prioritized nodes is given in Appendix A.5.

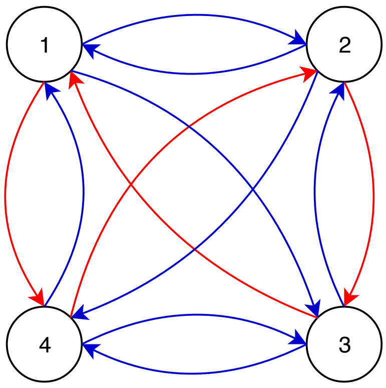


Figure 3: A configuration for Storm (Red: Narrow, Blue: Wide).

5.4 Storm

The third POETS application that was specified in TLA+ was Storm. Each node in the application has two output channels: **narrow** that connects it to a single node and **wide** that connects it to multiple nodes. One node, considered the root node, starts with all the credit, distributes all but one of them and then does not release any credits again. The application terminates when all the credit has been returned to the root node. If a node has at least as much credit as the number of nodes it's connected to through the wide channel, it randomly sends credits through either the wide channel or the narrow channel, otherwise it sends the credit through the narrow channel.

The TLA+ specification is given in Appendix A.6. The application was specified without considering out of order messages and assuming that the messages were received instantaneously, similar to the original Gals Heat and Gals Izhikevich specifications. When choosing between the narrow and wide channel the model checker explores both options. A few configurations were tried for model checking without success. The model kept on running for an arbitrarily long time even for configurations with small number of nodes. After further investigation it was realized that there could be a possibility of livelock[4]. Hence, an invariant was added in the specification to check for repeated states. Running the model checker again proved that there were repeated states. Since the model explored both the narrow and wide channels where possible, there existed a sequence of state transitions that resulting

[4]The state of application keeps on changing constantly without making any progress. A good analogy would be a program stuck in an infinite loop

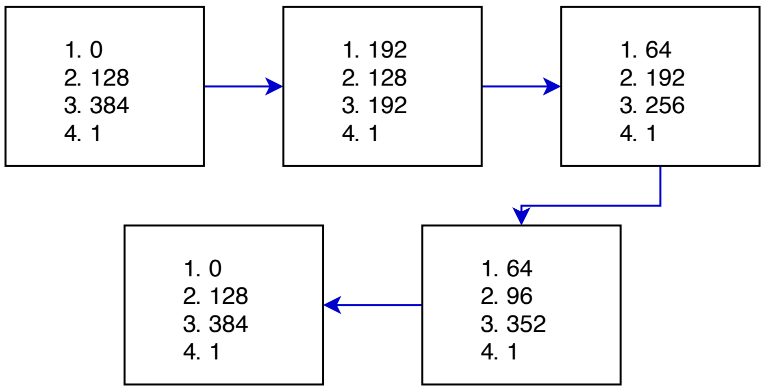


Figure 4: A configuration for Storm (Red: Narrow, Blue: Wide).

in repeated states and hence the model checker did not stop. One such configuration is given in Figure 3 and the exchange in credits resulting in a loop is given in Figure 4.

As of writing this report no modifications to the algorithm have been discovered that would avoid such loops. However, constraints can be set on the application graph so that no such loops exists. The configuration used here is very dense, resulting in loops. There is a much lesser chance of such loops in a sparse configuration. The main constraint that can be set on the application graph is that no independent loop exists that does not goes through the root note. It should also be noted while the model explores both the wide and narrow nodes where possible, the actual POETS implementation chooses one of them forever. Therefore, it is extremely unlikely that the same channel will always be selected and hence the loop will eventually be broken.

6 Evaluation

6.1 Hypothesis One

The first hypothesis stated that:

TLA+ is a suitable language for specifying POETS applications.

This can be proven by investigating the implementations of Gals Izhikevich and Storm applications in Section 5. These applications had doubts about their correctness. We were able to specify their original algorithms in TLA+ and use model checking to verify whether or not they were correct. For Gals Izhikevich specifically we were able to discover a bug in original algorithm, implement a fix and verify that the fix corrected the application. Furthermore, we were also able to propose an alternate algorithm for the application with added constraints on the message time stamp, devise an algorithm for it and verify that it was correct. Similarly, for the Storm application the algorithm was specified in TLA+ and a bug was discovered through model checking. Although a fix could not be explored due to time constraints, a possible constraint was suggested. Different messaging protocols were also specified in TLA+ with the same interface so that they could be used interchangeably. This was demonstrated through the Gals Izhikevich specifications that used these messaging protocol specifications to test the application with different messaging order behaviours.

Hence, we were able to specify POETS application in TLA+, verify that they were correct and if not, test out possible alterations that would make them correct. Therefore, it can be substantiated that TLA+ is indeed suitable for specifying POETS applications.

6.2 Hypothesis Two

The second hypothesis stated that:

The TLC model checker can provide useful results when run on the specified applications.

Similar to the first hypothesis, this can also be proven by investigating the implementation of the applications in Section 5. Hence, we can say that the model checker does provide useful results. What we are evaluating here is how the state space of the model varies with the problem size and application complexity. Since the most complete specification implemented during this project is Gals Izhikevich, the variations in state space with problem size and complexity were explored only for it. The state space variations will be similar for the other applications since they have very similar structures.

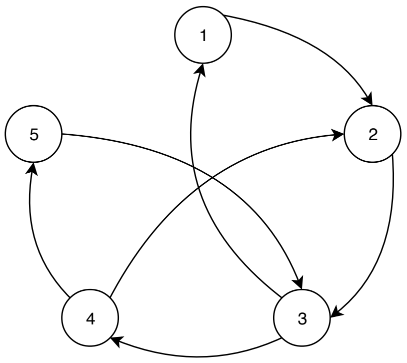
The specifications were tested for liveness[5] and safety[6] properties. The liveness property tested here was that all the processes/nodes in the system reach a specified time t. While TLA+ allows specifying liveness properties that was not done as the TLC models checker requires significantly greater resources to check for liveness properties. Hence, the property was checked by setting the maximum time reachable by each node so that the model deadlocked when all the nodes reached that time. While being less resource intensive it also had the benefit of displaying the total number of states, the total number of distinct states and the depth of the state space graph generated by the model. The safety test checks if any of the invariants described in the specification are violated by the model. Since the specifications being model checked were correct, an additional invariant was specified so that model would terminate and report an error as soon as a node reached a certain time t.

The Gals Izhikevich specification without the messaging protocols was not tested as the state space of its model can be easily estimated. The specification has a single action that each node can perform. So for n nodes there are a maximum[7]

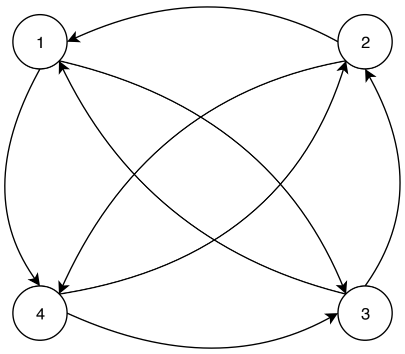
[5]The **liveness** property states that any action required for correct behaviour always happens.

[6]The **safety** property states that any action resulting in incorrect behaviour never happens.

[7]One of the nodes might not be able to progress if all the messages for previous time stamp have not been received yet making the actual number of next states $< n$.



(a) Configuration A



(b) Configuration B

Figure 5: Configurations used for testing.

of n possible next states actions for each state. If all the nodes have to reach time t the depth of the graph would be $d = n * t$ since at each state transition a node progresses by a single time-step. Hence the total number of states generated becomes $\leq \sum_{i=0}^{d} n^i$ which is $O(n^d)$. Hence only the state space of specifications including the messaging protocol will be explored.

Time	**Random**	**FILO**	**Choose**	**Priority**
1	2.35×10^5	5.1×10^4	4.9×10^3	8.1×10^4
2	1.18×10^8	1.5×10^8	6.0×10^4	6.6×10^5
3	7.83×10^9	-	1.3×10^5	9.4×10^6
4	-	-	1.9×10^5	1.2×10^8
5	-	-	2.6×10^5	1.5×10^9

Table 1: Total States vs Time (Config A: Liveness).

The configuration used for testing is given in Figure 5a. The differences in total number of state and the total number of distinct states with different time t for the liveness tests are given in Tables[8] 1 and 2 respectively. The depth of the state space graph given in Table 3 varies with time but is the same for all the messaging protocols.

[8]The "-" entries are the ones where the model checker was not able to complete the search in feasible time. The model was ran on a 16-core, ≈ 90-GB RAM machine for more than 24 hours but some bottleneck resulted in the model checker slowing down and not using most of the available processing power. The model may be able to finish the search on a more resource intensive system but this was not done due to cost and time restrictions.

Time	**Random**	**FILO**	**Choose**	**Priority**
1	3.4×10^4	3.7×10^4	2.4×10^3	2.8×10^4
2	1.17×10^7	1.1×10^8	2.2×10^4	1.8×10^5
3	6.48×10^8	-	4.4×10^4	2.3×10^6
4	-	-	6.8×10^4	2.5×10^7
5	-	-	9.1×10^4	3.0×10^8

Table 2: Distinct States vs Time (Config A: Liveness).

Time	1	2	3	4	5
Depth	20	39	58	77	96

Table 3: Depth of State Space Graph vs Time (Config A: Liveness).

As can be seen from Tables 1 and 2, the state space of the specification *ChooseRandomMessage* is the lowest. This is expected as here for the receive action a single message is randomly selected for the state transition while the rest are ignored. Hence, only a single receive action is added to the next state possibilities. This significantly reduces the search space of the model. The next model in terms of complexity is *PriorityNodes*. Here the search space is reduced by only considering the actions of Prioritized Nodes when possible for the next state actions. If no action is possible on the Prioritized Nodes then only the other nodes are expanded upon. The interesting result in this data is the difference between the search space of the specifications using *RandomMessages* and *FiloMessages* protocols. It was believed, as mentioned in Section 5.3, that *RandomMessages* being the most generalized implementation will have the largest search space. As can be seen from the results this is not the case.

The state space variations with different times for the safety tests are given in Table 4. As the time increases, the search space for FILO messages becomes greater than that of Random messages. This can be justified because the message order is constrained for FILO messages and the model needs to search to a greater depth before a node reaches the required time. Hence, the states found for FILO messages will be greater due to the increased depth, even though the possible next state actions might be less than that of Random Messages.

Since the state space results for the specifications using Random and FILO messaging protocols were unexpected another configuration given in Figure 5b was also

Time	Random	FILO
2	3.8×10^2	1.4×10^2
3	2.9×10^5	1.4×10^5
4	2.8×10^7	1.5×10^8
5	1.1×10^9	4.4×10^9

(a) Total States

Time	Random	FILO
2	2.5×10^2	1.4×10^2
3	5.8×10^4	1.3×10^5
4	3.6×10^6	1.3×10^8
5	1.1×10^8	3.5×10^9

(b) Distinct States

Time	Random	FILO
2	6	6
3	12	13
4	21	26
5	34	40

(c) Depth of Graph

Table 4: Results for safety tests (Config A).

tested for the liveness property. The results are given in Table 5. The results obtained here, while expected, are contrary to the ones in Tables 1 and 2. It could be the case that the state space for FILO messages in the second configuration does eventually become greater than that for Random messages as the time increases. However, a more plausible explanation would be that the state space of the model with respect to the messaging protocols is dependent on the structure of the underlying problem graph (configuration). These claims need to be further investigated, through testing for different configurations or analysing the possible next states of a model, before a definitive answer can be given.

Time	Random	FILO
1	6.5×10^5	1.3×10^4
2	4.7×10^8	1.1×10^7
3	5.0×10^{10}	1.9×10^8

(a) Total States

Time	Random	FILO
1	8.4×10^4	1.0×10^4
2	4.1×10^7	9.2×10^6
3	3.6×10^9	1.5×10^8

(b) Distinct States

Time	1	2	3
Depth	21	41	61

(c) Depth of Graph

Table 5: Results for liveness properties (Config B).

7 Further Work

One possible extension to the project would be to incorporate fault tolerance properties into the TLA+ specification. Another would be to verify the above mentioned applications by specifying them at a different abstraction level. The existing Gals Izhikevich specifications that use the messaging protocol consider the $Fire(n)$ and $Receive(m)$ steps to be atomic actions. Hence, scenarios like the arrival of a new message while the node is in the process of progressing to the next time step or receiving an earlier message are not considered. These scenarios are likely to occur in the actual system, hence specifying and verifying the application with these added scenarios will further strengthen the claim that the algorithm is correct.

7.1 Hypothesis 3

The third hypothesis stated that:

Existing POETS applications, written in XML and C++, can be mechanically converted to TLA+ specifications.

Due to time constraints this hypothesis was not properly explored during the project. TLA+ being declarative while C++ being imperative makes the possibility of mechanically converting applications written in C++ to TLA+ extremely challenging. However, there exists an algorithm language PlusCal [19] that can be transpiled to a TLA+ specification. PlusCal, being imperative and having constructs like the while loop, is a more suitable choice for our purposes. However, PlusCal being an algorithm language instead of a programming language requires some of its aspects to be restructured. Therefore further research is required to properly investigate the hypothesis.

8 Conclusion

The aim of this project was to explore the formal verification of POETS applications using TLA+. For this, three specific hypotheses were defined: whether TLA+ is suitable for formally specifying POETS applications, whether applications could be verified using the TLC model checker, and whether existing POETS applications could be mechanically converted to TLA+ specifications.

To help with the investigation of these hypotheses, three different POETS applications, called Gals Heat, Gals Izhikevich and Storm, were specified and verified using TLA+. Bugs in these applications were discovered and alternate algorithms, verified to be correct, were developed for them. They helped prove that the first two hypotheses were true. The third hypothesis could not be proved for lack of time, but a method to explore it has been provided.

The feasibility of using the model checker was also explored. While it is not possible to model check real world problem sizes, it was shown that even small problem sizes could still be very useful for detecting bugs.

Several messaging protocols were specified for testing POETS applications. These protocols can be used for other POETS applications without the need for any modification.

To summarise, we were able to demonstrate that TLA+ is indeed a suitable formal method for verifying POETS applications and that there are several benefits associated with using it.

9 User Guide

9.1 Messaging Protocols

The four messaging protocols specified are *RandMessages*, *ChooseRandMessage*, *FifoMessages* and *FiloMessages*. The properties of all the above protocols are described in Section 5.3. These properties were specified with the same interface so that they can be used interchangeably. To use any of the above protocols add its name to the $EXTENDS$ statement at the start of your specification.

- Define a variable to store all the messages currently being delivered.

$$VARIABLE \quad messages$$

- Initialize messages with *InitMsg* (provided in the message protocol).

$$messages = InitMsg$$

- To send a message use *SendMsg(messages, msg)*.

$$messages' = SendMsg(messages, msg)$$

- Use *MsgAvailable(messages)* to check if there is any message currently being delivered so that a receive step can be performed. Use *GetMsg(messages)* to get a set of messages[9] back. Use the above two functions to perform a receive message state.

$$\begin{aligned} &\land\ MsgAvailable(messages) \\ &\land\ \exists m \in GetMsg(message)\ :\ Receive(m) \end{aligned}$$

- Once a message *msg* has be received remove it from the channel .

$$messages' = RemoveMsg(messages, msg)$$

[9]There can either be a single message or multiple messages in the set depending on the protocol

9.2 Model Checking

The TLC model checker can be used through either an IDE called the TLA Toolbox or the command line TLA+ Tools. The download and install instructions for both are available at [20] and [21] respectively. Only the command line tools are discussed here.

A Docker image containing the TLA+ Tools and some other utilities like vim and tmux is provided. Use the following commands to download and run the it.

```
# docker pull maqur/tlaplus-tools
# docker run -it maqur/tlaplus-tools
```

Although the commands provided in [21] are available shorter commands have also been added for simplicity. These commands are *tlc*, *sany*, *pcal* and *tla2tex* respectively.

All the specification folders that include files *MC.tla* and *MC.cfg* can be model checked using the command `$ tlc MC` . While many different options are available for the model checker that can be access through `$ tlc -help` the main ones that were frequently used during the project are:

- -worker *num* : Runs the model checker using *num* parallel threads.
- -gzip : To compress the files that are generated during model checking.
- -config : To specify the config file if its name is different from the specifications name.

References

[1] S. Furber and A. Brown, "Partially-Ordered Event-Triggered Systems (POETS)," in *This Asynchronous World*, pp. 131–149, Newcastle University, 2017.

[2] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods," *Commun. ACM*, vol. 58, pp. 66–73, Mar. 2015.

[3] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems*. International computer science series, Pearson Education Limited, 2013.

[4] R. Guerraoui, M. Hurfinn, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper, *Consensus in Asynchronous Distributed Systems: A Concise Guided Tour*, pp. 33–47. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.

[5] D. Monroe, "Neuromorphic computing gets ready for the (really) big time," *Commun. ACM*, vol. 57, pp. 13–15, June 2014.

[6] A. Brown, D. Thomas, J. Reeve, G. Tarawneh, A. D. Gennaro, andrey Mokhov, M. Naylor, and T. Kazmierski, "Distributed event-based computing," in *Proceedings of ParCo 2017*, August 2017.

[7] H. N. Riley, "The von Neumann Architecture of Computer Systems." http://www-scf.usc.edu/~inf520/downloads/The%20von%20Neumann%20Architecture%20of%20Computer%20Systems.pdf, Sept. 1987. Accessed: 2018-06-19.

[8] N. B. Hare, "Modified Harvard Architecture: Clarifying Confusion." http://ithare.com/modified-harvard-architecture-clarifying-confusion/, Sept. 2015. Accessed: 2018-06-19.

[9] "POETS » Publications." https://poets-project.org/publications. Accessed: 2018-06-19.

[10] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu, "Specifying and verifying systems with tla+," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, (New York, NY, USA), pp. 45–48, ACM, 2002.

[11] M. Frappier and H. Habrias, *Software Specification Methods: An Overview Using a Case Study*. 01 2001.

[12] L. Lamport, “The temporal logic of actions,” *ACM Trans. Program. Lang. Syst.*, vol. 16, pp. 872–923, May 1994.

[13] A. Pnueli, “The temporal logic of programs,” in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS ’77, (Washington, DC, USA), pp. 46–57, IEEE Computer Society, 1977.

[14] L. Lamport, *Leslie Lamport: The Specification Language TLA+*, pp. 616–620. August 2008.

[15] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, June 2002.

[16] Y. Yu, P. Manolios, and L. Lamport, “Model checking tla+ specifications,” vol. 1703, pp. 54–66, June 1999.

[17] D. B. Thomas, “Graph Schema.” Commit Hash : c84dfe00ffc15d99f998941987a9ee67acd368f0.

[18] E. M. Izhikevich, “Simple model of spiking neurons,” *IEEE Transactions on Neural Networks*, vol. 14, pp. 1569–1572, Nov 2003.

[19] L. Lamport, “The pluscal algorithm language,” in *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, ICTAC ’09, (Berlin, Heidelberg), pp. 36–60, Springer-Verlag, 2009.

[20] “The TLA Toolbox.” `http://lamport.azurewebsites.net/tla/toolbox.html`. Accessed: 2018-06-20.

[21] “TLA+ Tools.” `http://lamport.azurewebsites.net/tla/tools.html`. Accessed: 2018-06-20.

A Specifications

All the application implementations, specification and other relevant software developed for this project can be at https://github.com/maqur/ee4-fyp. The main specifications developed during this project are also given below.

A.1 Gals Heat

```
─────────────── MODULE GalsHeat ───────────────
EXTENDS FiniteSets, Integers
───────────────────────────────────────────────
CONSTANTS Nodes,         Total number of nodes
          NEIGHBOURS,    A tuple of the set of neighbours with the index
                         being the node number
          MAXTIME        Maximum time for the systems

A variable that stores the state (t, cs, ns) of each node where “t” is the current time of the node,
“cs” is the count of messages received for the current time and “ns” is the count of messages
recieved for t + 1
VARIABLE state
───────────────────────────────────────────────
Initialize the state of each node so the t = 0, all the messages for t = 0 (cs) have been received
(the nodes are ready to progress to t = 1) and no messages for t = 1 (ns) have been received
GSInit ≜ state =
            [n ∈ 1 .. Nodes ↦
                [t ↦ 0, cs ↦ Cardinality(NEIGHBOURS[n]), ns ↦ 0]]

Next(n) ≜ ∧ state[n].t < MAXTIME
          Check if messages for the current time have been received from all the neighbours
          ∧ state[n].cs = Cardinality(NEIGHBOURS[n])
          ∧ state′ = [i ∈ 1 .. Nodes ↦
                 IF i = n THEN
                        [  t   ↦ state[i].t + 1,
                           cs  ↦ state[i].ns,
                           ns  ↦ 0]
                  ELSE IF i ∈ NEIGHBOURS[n] THEN
                      IF state[i].t = state[n].t + 1 THEN
                          Message received for the current time
                        [state[i] EXCEPT !.cs = 1 + @]
                      ELSE IF state[i].t = state[n].t THEN
                          Message received for time t + 1
                        [state[i] EXCEPT !.ns = 1 + @]
                      ELSE FALSE
                  ELSE
                        state[i]]

GSNext ≜ ∃ n ∈ 1 .. Nodes : Next(n)

GSSpec ≜ GSInit ∧ □[GSNext]_⟨state⟩
```

1

```
Check that the neighbours have a two way link
$NeighbourOK \triangleq$
        $\forall\, n \in 1 \,..\, Nodes :$
            $\forall\, a \in NEIGHBOURS[n] : n \in NEIGHBOURS[a]$

Check that the time difference between neighbours is never greater than 1
$TimeDiffOK \triangleq$
        $\forall\, n \in 1 \,..\, Nodes :$
            $\forall\, a \in NEIGHBOURS[n] :$
                $\land\, state[n].t - state[a].t < 2$
                $\land\, state[n].t - state[a].t > -2$

Check that the values of the state variables are correct
$StateVariablesOK \triangleq$
        $\forall\, n \in 1 \,..\, Nodes :$
            $\land\, state[n].t \leq MAXTIME$
            $\land\, state[n].cs \leq Cardinality(NEIGHBOURS[n])$
            $\land\, state[n].ns \leq Cardinality(NEIGHBOURS[n])$
```

A.2 Existing Gals Izhikevich

```
──────────── MODULE GalsIzhikevichOriginal ────────────
EXTENDS FiniteSets, Integers
────────────────────────────────────────────────────────

CONSTANTS  Neurons,   Total number of neurons
             A tuple representing the set of in neighbours of each neuron
           InNeighbours,
             A tuple representing the set of out neighbours of each neuron
           OutNeighbours,
           MaxTime

The state variable is a function with domain the set of neurons and range a record with fieds: t
(current time of neuron), p (number of pending fires), c (count of recieved messages)
VARIABLES  state

────────────────────────────────────────────────────────

Initialize the state of each neuron with t = 0, having 1 pending fire (p = 1) and not having
received any messages (c = 0)
GIInit  ≜  state = [n ∈ 1 .. Neurons ↦
                      [  t ↦ 0, p     ↦ 1, c ↦ 0  ]
                    ]

Next(n)  ≜  ∧ state[n].t < MaxTime
            ∧ state[n].p > 0
            ∧ state' =
               [a ∈ 1 .. Neurons ↦
                 IF a = n THEN
                    [  state[a] EXCEPT
                          !.t = @ + 1,
                          !.p = @ − 1,
                          !.c = 0
                    ]
                  ELSE IF a ∈ OutNeighbours[n] THEN
                    Increment pending fires if count equal to the number of in neighbours
                    IF Cardinality(InNeighbours[a]) = state[a].c + 1 THEN
                       [  state[a] EXCEPT
                             !.p = 1 + @,
                             !.c = 0
                       ]
                     ELSE
                       [ state[a] EXCEPT !.c = 1 + @ ]
                  ELSE
                    state[a]
               ]
```

1

$GINext \triangleq \exists\, n \in 1 \mathrel{..} Neurons : Next(n)$

$GISpec \triangleq GIInit \land \Box[GINext]_{\langle state \rangle}$

Check that the connections are correct

$$
\begin{aligned}
NeighbourOK \triangleq\ & \forall\, n \in 1 \mathrel{..} Neurons : \\
& \quad \land \forall\, i \in InNeighbours[n] : n \in OutNeighbours[i] \\
& \quad \land \forall\, o \in OutNeighbours[n] : n \in InNeighbours[o]
\end{aligned}
$$

Check that the out neighbour is not more then 1 timestep ahead, need to receive the message of current time step before jumping the next one

$$
\begin{aligned}
TimeDiffOK \triangleq\ & \forall\, n \in 1 \mathrel{..} Neurons : \\
& \quad \land \forall\, i \in InNeighbours[n] : \\
& \qquad \land state[n].t - state[i].t < 2 \\
& \quad \land \forall\, o \in OutNeighbours[n] : \\
& \qquad \land state[n].t - state[o].t > -2
\end{aligned}
$$

Check that the values of the state variables are correct

$$
\begin{aligned}
TypeOK \triangleq\ & \land\ \forall\, n \in 1 \mathrel{..} Neurons : \\
& \quad \land state[n].t \leq MaxTime \\
& \quad \land state[n].c \leq Cardinality(InNeighbours[n])
\end{aligned}
$$

A.3 Gals Izhikevich with Random Messages

```
──────────────────── MODULE GalsIzhikevich ────────────────────
EXTENDS     FiniteSets, Integers, Sequences, RandMessages

CONSTANTS   Nodes,
            InNeighbours,
            OutNeighbours,
            MaxTime,
            MaxMem

VARIABLE    state,
            messages

DCGInit ≜   ∧ messages = InitMsg
            ∧ state = [n ∈ 1 .. Nodes ↦ [
                    t ↦ 0,
                    c ↦ [m ∈ 1 .. MaxMem ↦
                        IF m = 1 THEN Cardinality(InNeighbours[n]) ELSE 0
                    ],
                     time difference (ahead) from out nueron
                    tDiff ↦ [o ∈ OutNeighbours[n] ↦ 0]
                ]]
├──────────────────────────────────────────────────────────────┤

Fire(n) ≜   ∧ state[n].t < MaxTime
            ∧ state[n].c[1] = Cardinality(InNeighbours[n])
            ∧ ∀ o ∈ OutNeighbours[n] : state[n].tDiff[o] < MaxMem − 1
            ∧ LET msg ≜
                    {[ type ↦ "fire",
                       sender ↦ n,
                       out ↦ o,
                       t ↦ state[n].t + 1
                    ] : o ∈ OutNeighbours[n]} ∪
                     send current time to IN nodes
                     to limit messages to a particular time ahead
                    {[ type ↦ "confirm",
                       sender ↦ n,
                       out ↦ i,
                       t ↦ state[n].t + 1
                    ] : i ∈ InNeighbours[n]}
                    IN  messages′ = SendMsg(messages, msg)
            ∧ state′ =
                [state EXCEPT
                    ![n].t = state[n].t + 1,
                    ![n].c =    [m ∈ 1 .. MaxMem ↦
                                IF m = MaxMem THEN 0 ELSE @[m + 1]
```

1

```
                    ],
             ![n].tDiff =
                   [o ∈ OutNeighbours[n] ↦ 1 + @[o]]
          ]

Receive(m) ≜   ∨ ∧ m.type = "fire"
                 ∧ messages′ = RemoveMsg(messages, m)
                 ∧ LET diff ≜ m.t − state[m.out].t + 1
                       IN   state′ =
                              [state EXCEPT ![m.out].c[diff] = 1 + @]

               ∨ ∧ m.type = "confirm"
                 ∧ messages′ = RemoveMsg(messages, m)
                 ∧ LET diff ≜
                           state[m.out].t − m.t
                   IN   state′ = [state EXCEPT
                            ![m.out].tDiff[m.sender] =
                                 IF diff < @ THEN diff ELSE @
                   ]

DCGNext ≜  ∨ ∃ n ∈ 1 . . Nodes : Fire(n)
           ∨ ∧ MsgAvailable(messages)
             ∧ ∃ m ∈ GetMsg(messages) : Receive(m)

────────────────────────────────────────────────────────────

NeighbourOK ≜  ∀ n ∈ 1 . . Nodes :
                    ∧ ∀ i ∈ InNeighbours[n] : n ∈ OutNeighbours[i]
                    ∧ ∀ o ∈ OutNeighbours[n] : n ∈ InNeighbours[o]

TypeOK  ≜  ∧ ∀ n ∈ 1 . . Nodes :
                 ∧ state[n].t ≤ MaxTime
                 ∧ ∀ m ∈ 1 . . MaxMem :
                        state[n].c[m] ≤ Cardinality(InNeighbours[n])

TimeDiffOK ≜  ∀ n ∈ 1 . . Nodes :
                   ∧ ∀ i ∈ InNeighbours[n] :
                        state[i].t − state[n].t < MaxMem
                   ∧ ∀ o ∈ OutNeighbours[n] :
                        ∧ state[n].tDiff[o] < MaxMem
                        ∧ state[n].t − state[o].t < MaxMem
```

A.4 Random Messages Protocol

MODULE *RandMessages*

EXTENDS *FiniteSets*, *Integers*

$InitMsg \triangleq \{\}$

$SendMsg(msgQueue, msg) \triangleq msgQueue \cup msg$

$MsgAvailable(msgQueue) \triangleq Cardinality(msgQueue) > 0$

$GetMsg(msgQueue) \triangleq msgQueue$

$RemoveMsg(msgQueue, msg) \triangleq msgQueue \setminus \{msg\}$

1

A.5 Gals Izhikevich with Priority Nodes

```
──────────────── MODULE GalsIzhikevichPriority ────────────────

EXTENDS      FiniteSets, Integers, Sequences, RandMessages

CONSTANTS    Nodes,
             PriorityNodes,
             InNeighbours,
             OutNeighbours,
             MaxTime,
             MaxMem

VARIABLE     state,
             messages

DCGInit ≜   ∧  messages = InitMsg
             ∧  state = [n ∈ 1 .. Nodes ↦ [
                    t ↦ 0,
                    c ↦ [m ∈ 1 .. MaxMem ↦
                        IF m = 1 THEN Cardinality(InNeighbours[n]) ELSE 0
                    ],
                     time difference (ahead) from out nueron
                    tDiff ↦ [o ∈ OutNeighbours[n] ↦ 0]
                ]]

───────────────────────────────────────────────────────────────

Fire(n) ≜   ∧  state[n].t < MaxTime
             ∧  state[n].c[1] = Cardinality(InNeighbours[n])
             ∧ ∀ o ∈ OutNeighbours[n] : state[n].tDiff[o] < MaxMem − 1
             ∧  LET msg ≜
                    {[ type ↦ "fire",
                       sender ↦ n,
                       out ↦ o,
                       t ↦ state[n].t + 1
                    ] : o ∈ OutNeighbours[n]} ∪
                     send current time to IN nodes
                     to limit messages to a particular time ahead
                    {[ type ↦ "confirm",
                       sender ↦ n,
                       out ↦ i,
                       t ↦ state[n].t + 1
                    ] : i ∈ InNeighbours[n]}
                    IN   messages′ = SendMsg(messages, msg)
             ∧ state′ =
                 [state EXCEPT
                    ![n].t = state[n].t + 1,
                    ![n].c =    [m ∈ 1 .. MaxMem ↦
```

1

```
                    IF $m = MaxMem$ THEN $0$ ELSE $@[m+1]$
                    ],
          $![n].tDiff =$
               $[o \in OutNeighbours[n] \mapsto 1 + @[o]]$
        ]

$Receive(m) \triangleq$  $\lor$ $\land$ $m.type = $ "fire"
                $\land$ $messages' = RemoveMsg(messages,\ m)$
                $\land$ LET $diff \triangleq m.t - state[m.out].t + 1$
                      IN  $state' =$
                          $[state$ EXCEPT $![m.out].c[diff] = 1 + @]$

            $\lor$ $\land$ $m.type = $ "confirm"
                $\land$ $messages' = RemoveMsg(messages,\ m)$
                $\land$ LET $diff \triangleq$
                        $state[m.out].t - m.t$
                    IN  $state' = [state$ EXCEPT
                         $![m.out].tDiff[m.sender] =$
                              IF $diff < @$ THEN $diff$ ELSE $@$
                    ]

$CanFire(n) \triangleq$  $\land$ $state[n].t < MaxTime$
               $\land$ $state[n].c[1] = Cardinality(InNeighbours[n])$
               $\land \forall o \in OutNeighbours[n] : state[n].tDiff[o] < MaxMem - 1$

$PriorityFire \triangleq$ IF $\exists n \in PriorityNodes : CanFire(n)$
                  THEN $\exists n \in PriorityNodes : Fire(n)$
                  ELSE $\exists n \in 1 \mathrel{..} Nodes \setminus PriorityNodes : Fire(n)$

$PriorityRecieve \triangleq$
      $\land$ $MsgAvailable(messages)$
      $\land$ IF $\exists m \in GetMsg(messages) : m.out \in PriorityNodes$
          THEN $\exists m \in GetMsg(messages) :$
                       $\land$    $m.out \in PriorityNodes$
                       $\land$    $Receive(m)$
          ELSE $\exists m \in GetMsg(messages) : Receive(m)$

$DCGNext \triangleq$  $\lor$ $PriorityFire$
              $\lor$ $PriorityRecieve$

--------------------------------------------------------------------

$NeighbourOK \triangleq$  $\forall n \in 1 \mathrel{..} Nodes :$
                  $\land \forall i \in InNeighbours[n] : n \in OutNeighbours[i]$
                  $\land \forall o \in OutNeighbours[n] : n \in InNeighbours[o]$

$TypeOK \triangleq$  $\land$ $\forall n \in 1 \mathrel{..} Nodes :$
```

2

$$
\begin{array}{lll}
 & & \land\ state[n].t \leq MaxTime \\
 & & \land\ \forall\, m \in 1 \mathrel{..} MaxMem : \\
 & & \qquad state[n].c[m] \leq Cardinality(InNeighbours[n])
\end{array}
$$

$$
\begin{array}{lll}
TimeDiffOK & \triangleq & \forall\, n \in 1 \mathrel{..} Nodes : \\
 & & \quad \land\ \forall\, i \in InNeighbours[n] : \\
 & & \qquad state[i].t - state[n].t < MaxMem \\
 & & \quad \land\ \forall\, o \in OutNeighbours[n] : \\
 & & \qquad \land\ state[n].tDiff[o] < MaxMem \\
 & & \qquad \land\ state[n].t - state[o].t < MaxMem
\end{array}
$$

A.6 Storm

```
─────────────────────────── MODULE Storm ───────────────────────────
EXTENDS TLC, Naturals, FiniteSets, Sequences
────────────────────────────────────────────────────────────────────
CONSTANTS Nodes,
          Root,
          NarrowEdge,
          WideEdges

VARIABLES state,
          previousStates
────────────────────────────────────────────────────────────────────
SInit ≜  ∧ state =
             [n ∈ 1 . . Nodes ↦ [credits ↦ 0, totalcredits ↦ 0]]
         ∧ previousStates = ⟨⟩

SendWide(n) ≜
     ∨     ∧  n = Root
           ∧  state[n].credits = 0
           ∧  LET perDevice ≜
                  CHOOSE c ∈ 1 . . 1024 :
                     ∧ ∃ m ∈ 0 . . 10 : c = 2^m
                     ∧ c ∗ Cardinality(WideEdges[n]) < 1024
                     ∧ ∀ j ∈ 1 . . 1024 :
                         ∧ ∃ m ∈ 0 . . 10 : j = 2^m
                         ∧ j ∗ Cardinality(WideEdges[n]) < 1024
                         ⇒ c ≥ j
              IN   state′ = [i ∈ 1 . . Nodes ↦
                  IF i = n THEN
                     [credits ↦ 1, totalcredits ↦
                        1 + perDevice ∗ Cardinality(WideEdges[i])
                     ]
                   ELSE  IF i ∈ WideEdges[n] THEN
                     [credits ↦ state[i].credits + perDevice,
                      totalcredits ↦ state[i].totalcredits
                     ]
                   ELSE  state[n]
              ]
           ∧  previousStates′ = Append(previousStates, state)
     ∨     ∧  n ≠ Root
           ∧  state[n].credits ≥ Cardinality(WideEdges[n])
           ∧  LET perDevice ≜ CHOOSE c ∈ 1 . . state[n].credits :
                    ∧ ∃ m ∈ 0 . . 10 : c = 2^m
```

```
            $\land c * Cardinality(WideEdges[n]) \leq state[n].credits$
            $\land \forall j \in 1 \mathrel{..} state[n].credits :$
                $\land \exists m \in 0 \mathrel{..} 10 : j = 2^m$
                $\land j * Cardinality(WideEdges[n]) \leq state[n].credits$
                $\Rightarrow c \geq j$
          IN    $state' = [i \in 1 \mathrel{..} Nodes \mapsto$
            IF $i = n$ THEN
              $[credits \mapsto$
                 $state[n].credits - perDevice * Cardinality(WideEdges[i]),$
               $totalcredits \mapsto state[i].totalcredits]$
             ELSE IF $i \in WideEdges[n]$ THEN
              $[credits \mapsto state[i].credits + perDevice,$
               $totalcredits \mapsto state[i].totalcredits]$
             ELSE $state[i]$
          $]$
      $\land$ $previousStates' = Append(previousStates, state)$

$SendNarrow(n) \triangleq$  $\land$ $n \neq Root$
                    $\land$ $state[n].credits > 0$
                    $\land$ LET $msgCredits \triangleq (state[n].credits + 1) \div 2$
                       IN    $state' = [i \in 1 \mathrel{..} Nodes \mapsto$
                         IF $i = n$ THEN
                           $[credits \mapsto state[n].credits - msgCredits,$
                            $totalcredits \mapsto state[i].totalcredits]$
                          ELSE IF $i \in NarrowEdge[n]$ THEN
                           $[credits \mapsto state[i].credits + msgCredits,$
                            $totalcredits \mapsto state[i].totalcredits]$
                          ELSE $state[i]$
                       $]$
                    $\land$ $previousStates' = Append(previousStates, state)$

$SNext \triangleq \exists n \in 1 \mathrel{..} Nodes : \lor SendWide(n)$
                             $\lor SendNarrow(n)$

$SSpec \triangleq SInit \land \Box[SNext]_{\langle state, Nodes, WideEdges, NarrowEdge, Root\rangle}$
```

```
$NoLoop \triangleq \forall i \in 1 \mathrel{..} Len(previousStates) :$
             $\forall j \in 1 \mathrel{..} Len(previousStates) :$
               IF $i \neq j$ THEN
                 $previousStates[i] \neq previousStates[j]$
                ELSE TRUE
```

2