TECHNISCHE
HOCHSCHULE
LÜBECK

Master-Thesis

Simulation and Visualisation of Distributed Algorithms using GO and UPPAAL

Vorgelegt von: Torben Friedrich Görner

Fachbereich: Elektrotechnik und Informatik

Studiengang: Informatik / Softwaretechnik für verteilte Systeme

Erstprüfer/in: Prof. Dr. Schäfer

Ausgabedatum: 05. Juli 2022
Abgabedatum: 05. Januar 2023

A. Hanemann

(Professor Dr. Andreas Hanemann)
Vorsitzender des Prüfungsausschusses

Aufgabenstellung:

Within the scope of the master's thesis, it is examined how concepts of distributed algorithms can be modelled, analysed and visualised.
The programming language GO is used to model and specify the algorithm. The specification is then converted into a net of timed automata and analysed using UPPAAL.
As a first step, it is investigated which concepts of distributed algorithms are crucial and how they can be realised in GO. As a second step, a translation scheme of the aforementioned specification from GO to UPPAAL is developed and prototypically implemented. Lastly, it is determined how traces are used in UPPAAL and how they can be used to visualise the simulation of distributed algorithms.

Prof. Dr. Schäfer

TECHNISCHE HOCHSCHULE LÜBECK

Eigenständigkeitserklärung

Declaration of Originality

Name, Vorname
Last name, first name

Matrikelnummer
Matriculation number

Ich versichere hiermit, dass ich die vorliegende
I hereby declare that this

☐ **Hausarbeit** term paper ☐ **Bachelorarbeit** bachelor‘s thesis ☐ **Masterarbeit** master‘s thesis

mit dem Titel
with the title

eigenständig und ohne unerlaubte fremde Hilfe angefertigt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und Entlehnungen aus anderen Arbeiten kenntlich gemacht. Für den Fall, dass die Arbeit zusätzlich elektronisch und/ oder digital eingereicht wird, erkläre ich, dass die schriftliche und die elektronische und/ oder digitale Form identisch sind. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

is my own original work and any assistance from third parties has been acknowledged. I have clearly indicated and acknowledged all sources and resources as well as any borrowings from other works. In case of an additional electronic and/or digital submission of this work, I declare that the written form and the electronic and/or digital form are identical. This work has not previously been submitted either in the same or in a similar form to another examination office.

Ich bin damit einverstanden, dass die vorliegende Hausarbeit/ Bachelorarbeit/ Masterarbeit für Veröffentlichungen, Ausstellungen und Wettbewerbe des Fachbereiches verwendet und Dritten zur Einsichtnahme vorgelegt werden kann.

I agree that this work can be used for publishing, exhibition or competition purposes and can be inspected by third parties.

☐ **ja** Yes ☐ **nein** no ☐ **es liegt ein Sperrvermerk bis _____vor** there is an embargo period until

Ort, Datum
Place, Date

Unterschrift
Signature

Simulation and Visualisation of Distributed Algorithms using GO and UPPAAL

Submitted by

Torben Friedrich GÖRNER

Thesis Advisor

Prof. Dr. rer. nat. Andreas SCHÄFER

INFORMATION TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
TECHNICAL UNIVERSITY OF APPLIED SCIENCES LÜBECK

A thesis submitted in fulfillment of the requirement to the degree
Master of Science (M.Sc.)

2022

Declaration of Authorship

I, Torben Friedrich GÖRNER, declare that this thesis titled, "Simulation and Visualisation of Distributed Algorithms using GO and UPPAAL" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly conducted while in candidature for a degree at Technical University of Applied Sciences Lübeck.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

TECHNICAL UNIVERSITY OF APPLIED SCIENCES LÜBECK

Abstract

Department of Electrical Engineering and Computer Science

Master of Science (M.Sc.)

Simulation and Visualisation of Distributed Algorithms using GO and UPPAAL

by Torben Friedrich GÖRNER

This thesis describes how distributed algorithms can be modeled using a subset of the GO language as a specification in order to automatically generate a UPPAAL model for analysis purposes. This model is used for simulation and verification with UPPAAL. Also it is shown how distributed algorithms can be visualised using UPPAAL traces. It is shown how various GO concepts can be translated into a UPPAAL model and in particular with focus on Channels. For distributed systems, we look at synchronous and asynchronous communication, channels with buffers, and non-FiFo channels. For our models and simulations, we also investigate sources of errors such as failing nodes and messages that are lost in transit. In the context of this thesis, 1.) a GO framework (DiAlGo Go Framework), 2.) an automated translation from GO to UPPAAL (DiAlGo Translator) and 3.) a viewer which visualises UPPAAL traces of distributed algorithms were developed. Three distributed algorithms from the disciplines of termination, mutual exclusion and election are used to illustrate the concepts and evaluate the results of the work.

Acknowledgements

I would like to thank my family who have supported me throughout my studies.

I would like to thank Prof. Dr. Andreas Schäfer for the excellent supervision.

I thank my girlfriend Franzi for the support and the laptop when mine broke.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This thesis describes how distributed algorithms can be described using a subset of GO [19] as a specification language in order to automatically generate a UPPAAL [29] model to simulate, analyse and visualise. For this purpose, a GO framework is presented, which provides various functionalities for modeling distributed algorithms, as well as a tool for the automated transfer to UPPAAL. In particular, different forms of communication between nodes in a network are examined and presented. These include synchronous and asynchronous communication, as well as non-FiFo Channel and lossy channel. Non-deterministic error sources such as crashing nodes are also examined. Non-deterministic behaviour such as randomly choosing a boolean or integer is also considered. For general control statements like loops or if blocks, as well as GO statements like the GO select or assignments, a translation scheme is presented. In addition to translation, the modification or annotation of UPPAAL models is also investigated. Here we show the adding of an extra automaton for switching off nodes. The developed prototypes are publicly available on GitHub [14].

As a second aspect of the work, an alternative visualisation to the UPPAAL simulator is examined. For this purpose, a prototypically implemented tool is presented, which realises an alternative visualisation. This is not dynamic like the UPPAAL simulations, but visualises a fixed trace. The new visualisation abstracts the internal logic of the nodes and focuses on the communication in the network.

In addition to a discussion of the developed concepts, an implementation is described and the work is compared with related work in the field of simulation and visualisation of distributed algorithms. Furthermore, the developed tools for translation and visualisation, as well as the GO Framework are evaluated and an outlook for possible extensions is given.

Besides smaller examples to demonstrate modelling and translations, 3 larger examples for distributed algorithms are considered in *chapter 7*. We consider an implementation of Huang's algorithm for detecting termination [16] in a distributed system as a continuous example in *chapters 4, 5* and *6*. Since the automatically generated UPPAAL model in particular is quite large and complex, we restrict the presentation to certain parts. The complete model and the GO code can be taken from the appendix.

1.1 Motivation

Compared to traditional algorithms, distributed algorithms face a variety of complex problems related to concurrent execution. Understanding the behavior of a distributed algorithm can be very difficult at times, both in teaching and in research and development. Due to different possible schedules, a system can behave unexpectedly and differently each time.

A well-known example of concurrent system is Edsger W. Dijkstra's dining philosophers example. Five philosophers sit at a table and eat spaghetti. Each philosopher has a plate and a fork on the left and right next to each plate. A philosopher needs 2 forks to eat, which means they can't all eat at the same time. So the philosophers have to share the forks in order to eat. When someone gets hungry, they first grab the fork on the left, then the fork on the right and start eating. When a philosopher is done, he puts the forks back down and starts thinking again. If a fork is not in its place when the philosopher wants to pick it up, he waits until the fork is available again. The procedure works fine as long as only individual philosophers are hungry. But if every philosopher grabs his left fork, for example, no one can eat and the system becomes deadlocked. The philosophers are starving.

To solve such problems, distributed algorithms such as the Suzuki-Kasami mutual exclusion algorithm [27] exist. Other problems of distributed algorithms may include detecting termination, for which Huang's algorithm [16], for example, provides a solution or consensus, for which the Raft algorithm [23] was developed. To understand such algorithms, it can be helpful to look at and analyse example schedules by simulating and visualising those algorithms.

The second aspect, which is difficult to understand when teaching or developing distributed algorithms, is the non-deterministic occurrence of technical errors like node failures or lost messages in a system. For example, when sending messages over the Internet, one message may overtake the other. Another problem with distributed programming can be, for example, the failure of a server that is perhaps only temporarily or permanently unavailable. Analysing the behavior under certain sources of error such as failing nodes, messages that have been mixed up in the order, or lost messages can help to better understand distributed algorithms, check the robustness or to support the modeling of new algorithms. For this it would be useful to simulate such errors and their non-deterministic occurrence.

In summary, it can be said that good distributed algorithms must be safe in terms of concurrent execution and robust in the face of technical failures or disruptions. Distributed algorithms can be very difficult to understand and hard to implement and debug. In order to analyse or understand the behavior of a distributed algorithm, be it in relation to teaching or in relation to the development of new algorithms, it makes sense to simulate and visualise them.

The UPPAAL model checker [29] offers a way of simulating, visualising and analysing such algorithms. With its simulator certain schedules (traces) can be analysed and safety properties about the behavior can be checked with its verifier. A modern programming language, which is particularly intended for concurrent programming, is GO [19]. With its concept of channels for exchanging messages and goroutines for parallelism, GO is ideal for programming distributed algorithms. UPPAAL and GO were chosen for these benefits. The interaction of GO and UPPAAL for the simulation and visualisation of distributed algorithms for teaching, research and development is presented in this work.

1.2 Structure

This work can be divided into 2 parts. First, distributed algorithms are to be specified in GO and automatically translated into a UPPPAAL model. Second, an alternative visualisation for UPPAAL traces will be developed.

FIGURE 1.1: Step 1 - From specification to formal model

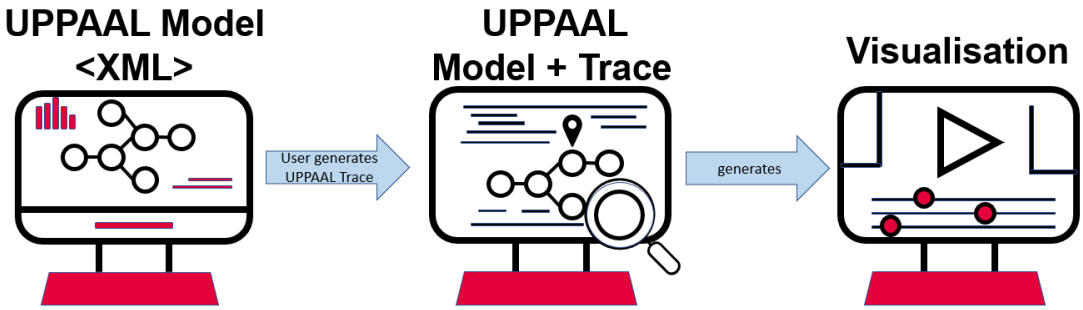


FIGURE 1.2: Step 2 - From UPPAAL Trace to visualisation

1.) From GO to UPPAAL

The first part (Fig.1.1) is the development of a concept for specifying distributed algorithms in the GO language. This specification should then be automatically translated to a formal model in UPPAAL, i.e. to an automaton network. Since the GO language is very extensive, only a subset can be considered and implemented in the implementation of the prototype. A GO framework and a translator tool are presented for this purpose.

2.) Visualisation of Simulations

The second part (Fig.1.2) is the development of a concept and prototype to descriptive visualise a UPPAAL trace with an associated model. A prototype tool is presented for this purpose.

Usage of the DiAlGo Project

These concepts have been prototypically implemented and tested in the tools DiAlGo Translator and DiAlGo Viewer. The process of using the DiAlGo Project can look like shown in Fig.1.1 and Fig.1.2. The user specifies a distributed algorithm such as a mutual exclusion or leader election algorithm in GO with the DiAlGo GO Framework. After the automated translation with the DiAlGo Translator to UPPAAL, the user can analyse the formal model, do verification and create UPPAAL traces. The created trace can then be visualised in connection with the associated UPPAAL model using the DiAlGo Viewer.

2

RELATED WORK

Various related works on the simulation of distributed algorithms can be found in the literature. Some describe more of a high-level approach to communication, while others examine and present more concrete models in which the communication itself (communication channels) is more in focus. The authors pursue different goals in their work. a) The visualisation of distributed algorithms for teaching and b) The investigation of distributed algorithms under verification aspects. Furthermore, many works on the visualisation of distributed algorithms can be found. In most works, both aspects are examined and implemented.

The basic idea of using a specification language to describe distributed algorithms in order to create formal models or animations for investigation is not new. There are some works and tools that, for example, use Java as a specification language to model concurrency and then derive a formal model from the specification or visually simulate the algorithm. First, we consider work on the specification for simulating distributed algorithms, and then more general work on visualising distributed systems and algorithms. We also consider related work on translating GO to UPPAAL. Later, we group the presented tools for a better overview in table 2.1. The DiAlGo project is also included in the table for a direct comparison.

2.1 Related Tools for Simulating and Visualising distributed algorithms

In the following, we present tools and compare them with this work (Dialgo GO Framework for specifications of distributed algorithms in GO, Dialgo Translator for translation to UPPAAL and Dialgo Viewer for visualisation of simulations).

jBACI : The tool jBACI is a concurrency simulator by Ben-Ari [5]. He developed jBACI for teaching purposes to graphically illustrate and explore concurrency programming. A subset of Pascal and C is used as the specification language of concurrent processes and threads. PCode is generated, an intermediate language which is executed by a Debugger. The debugger allows breakpoints and single-stepping.

Compared to jBACI, the DiAlGo project offers deeper analysis possibilities through the UPPAAL Verifier and Simulator. For example, assertions about the behaviour in CTL can be described and concrete traces can be analysed. With the UPPAAL simulator, it is also possible to examine the programme behaviour step by step and to examine various possible traces.

DAJ : The DAJ tool is also from Ben-Ari [4]. Distributed algorithms such as the Ricart-Agrawala algorithm or Byzantine generals algorithm for consensus can be specified in Java. Like jBACI, the tool was mainly developed for teaching. Each step of the execution sequence must be executed individually. Examined traces can be saved and reviewed. The tool is interactive and shows the current state of each node. Nodes exchange information or Java objects via methods. This is how channels between nodes are simulated.

One similarity to the DiAlGo project is the specification using a framework. Like the Java Framework for DAJ, the DiAlGo GO Framework is a framework that supports the implementation of distributed algorithms with functionalities such as sending from A to B. However, the analysis and visualisation is done in separate environments (UPPAAL, DiAlGo Viewer). Thus, the DiAlGo project offers deeper analysis possibilities.

ViSiDiA : Another tool is ViSiDiA [2]. Distributed algorithms can be specified here in Java or in a GUI and then simulated visually. The created simulation graphics are not dynamic. In this model, for example, processes and their communication are represented by a graph of nodes and edges. Communication takes place via channels which are represented by the edges in the graph. In ViSiDiA, the same algorithm is copied to each node. Each cloned algorithm is thus related to a node.

The ViSiDiA library is similar to the DiAlGo Go framework. It holds functionalities for the implementation of distributed algorithms. In contrast to ViSiDiA, it is possible to create dynamic simulations with the DiAlGo project. However, the visualisation of created simulations with the DiAlGo Viewer are static, just like ViSiDiA. The DiAlGo project offers deeper analysis possibilities with the UPPAAL verifier and simulator.

LYDIAN : In the Tool LYDIAN [20], distributed algorithms are described as a network, which is created visually by the user via a GUI. This tool was also developed for teaching. The model of a network of nodes is animated with a created trace file. The communication and the status of individual nodes are represented by means of directed edges and coloring in the graph. The main focus is on the analysis of the message traffic.

In contrast to LYDIAN, in the DiAlGo project the specification of the distributed algorithms is created in real programme code (Go) and not in a GUI. The programme code can therefore also be used independently of the DiAlGo project. When it comes to analysing traffic in the network, the DiAlGo project offers many possibilities for expansion, whereby LYDIAN can be taken as a reference. The DiAlGo project offers deeper analysis possibilities with the UPPAAL verifier and simulator.

Distal : With Distal [7], distributed algorithms can be specified in a domain-specific language (DSL) and translated into executable code. The aspect relevant to this work is the specification, in particular communication via channels. For example, lossy channels can be simulated with a messaging layer.

Sources of error are also considered in the DiAlGo project. Lossy channels, for example, are also implemented. In terms of analysis possibilities, DiAlGo offers

more options through the use of UPPAAL. Furthermore, in difference to Distal, a visualisation tool has been developed.

FADA : Algorithms (e.g. in a client-server architecture) can be simulated and visualised with the FADA tool [11]. A Java framework similar to that of DAJ [4] is used for the specification. The animations created are interactive, so that individual nodes in a network can be switched on or off, for example.

The Java Framework is similar to the DiAlGo Go framework. It holds functionalities for the implementation of distributed algorithms. Also, dynamic simulations are possible through the UPPAAL Simulator and switching nodes on and off through the DiAlGo Translator Tool. In terms of analysis possibilities, DiAlGo offers more options through the use of UPPAAL with it's verifier.

VADE : With the VADE tool [22], a server can be accessed remotely via a website in order to specify and visualise asynchronous distributed algorithms. The consensus at VADE is always reliable, and there is no global clock. A directed graph is used for the visualisation. In order to run an algorithm and watch its animation, the user opens a web page and selects an algorithm. This tool was also developed for the purpose of teaching.

The fundamental architecture is different in the DiAlGo project. Unlike VADE, in the DiAlGo project the user can specify a distributed algorithm themselves and execute it independently of UPPAAL or DiAlGo Viewer. In addition, deeper analyses are possible through the use of UPPAAL with its verifier.

Result : Using UPPAAL as a tool for analysis and simulation is a fundamentally new aspect compared to other work. The use of GO with a framework as a specification language is also new, as no comparable work has been found. The organisation of the project into independent parts and tools - *a)* specification in GO, *b)* translation from GO to UPPAAL for analysis and simulation, and *c)* an alternative visualisation for UPPAAL traces of simulations - is also a new approach.

Tool	Specification	Visualisation	Interactivity
jBACI [5]	(subsets of) Pascal and C	source code, PCode, variables, stack and history	step-by-step view
DAJ [4]	Java (Framework)	strings and integers in panels, message trees	dynamic interactive, step-by-step view
ViSiDiA [2]	Java or GUI	graph of nodes and edges with state information	static
LYDIAN [20]	GUI	graph of nodes and edges with state information, EnViDiA extension with 3D GUI	static
Distal [7]	DSL	-	-
FADA [11]	Java (Framework)	network graph with state information	dynamic interactive
VADE [22]	-	graph of nodes and edges with state information	dynamic interactive
DiAlGo	GO Subset	UPPAAL Simulator and in the DiAlGo Viewer node with information about the state, as well as moving messages.	dynamic interactive with UPPAAL and static with DiAlGo Viewer

TABLE 2.1: Grouping of the presented Tools

2.2 Related Work on Translating GO to UPPAAL

Related works that use GO as a specification language were not found. However, there are works on the general translation from GO to UPPAAL and vice versa [10][26]. These works do not sufficiently consider, for our purposes, how different types of channels (synchronous, asynchronous, asynchronous with buffer or non-FiFo) can be modeled or translated. General concepts such as translating a control flow (for loops or conditional statements, like if and else) are described.

2.3 Related Work on Visualising Distributed Systems

For presentation purposes, some developers of distributed algorithms have developed animations in the form of videos as well as small interactive tools embedded in a website wit JavaScript, for example. These were examined as part of the work in order to analyse which aspects of a distributed algorithm can be visualised and animated and how.

Raft Visualisation : A visualization tool is embedded on the GitHub page [23] as a simple visual explanation of the Raft consensus algorithm as screen-cast. Here, servers are represented by nodes, which communicate with each other and use the algorithm. The messages between the nodes are represented by moving entries. Status information is displayed by clicking on the nodes or message entries. The user can interact with the screen-cast and, for example, switch off nodes or adjust the speed.

The DiAlGo Viewer is inspired by this. The arrangement of the nodes in a circle as well as the directed moving entries for sending messages were adopted here. Clicking on a node to display its state (local variables) was also derived from this model.

3 PRELIMINARIES

In this section we take a look on the theoretical basics of distributed algorithms as well as the language GO and the model-checker UPPAAL. First, it is defined how distributed algorithms can be formally described. Then GO and UPPAAL will be introduced. In doing so, we consider most the aspects that are particularly important in relation to distributed algorithms.

3.1 Distributed Algorithms

In this work, we use the definition of distributed algorithms according to Fokkink [12]. A distributed algorithm can be described as an finite network N of processes (nodes) that communicate and exchange information via channels (edges). A node always has an unique ID. In our model, processes (nodes) can have local and shared memory. Fokkink defines processes in such a way that shared memory does not exist. However, this is optional possible in our model as Uppaal offers this possibility with global declarations. But we only use them to define constants and as a buffer for channels. In the following we consider how distributed algorithms can be formally described in detail. We consider transition systems, states and events, message passing, causal orders and assertions. At the end, some possible sources of error are described.

Transition Systems : The global state of a distributed algorithm across all processes is determined by transitions. As a result, the behavior is mostly deterministic. The global state is called *configuration*. The behavior of a distributed algorithm is described by a transition system as follows [12]:

- a set C of configurations,
- a binary transition relation $\longrightarrow$ on C, and
- a set $I \subseteq C$ of initial configurations.

A configuration γ is terminal if it has no outgoing transition. The execution of a distributed algorithm is a sequence of configurations $(\gamma_0, \gamma_1 ...)$. This can be infinite or terminate with a terminal configuration γ_k. A configuration δ is reachable if there exists an initial configuration $\gamma_0 \in I$ and a sequence $\gamma_0, \gamma_1, ...\gamma_k$ with $\gamma_i \longrightarrow \gamma_{i+1}$ for all $0 \leq i < k$ and $\gamma_k = \delta$.

States and Events : The configuration of a distributed algorithm consists of global states, local states of all processes and the messages in transit (stored in global states). A transition between configurations is called an event. Depending on whether we are looking at a synchronous or an asynchronous system, such a transition is one event or two distinct events. We will look at the difference between synchronous and asynchronous systems in the next section. An event is called *internal* if it only changes the state of its process. Such an event can be reading and writing to local variables. Further events can be sending and receiving messages. A process is called an *initiator* if it can start executing events without input from another process. An algorithm is said to be *centralized* if there is exactly one initiator. A *decentralized* algorithm can have multiple initiators.

Message Passing : The most important point of distributed systems is communication. Therefore, we take a closer look at message passing. In a network, nodes can use channels to exchange informations with other nodes. A node cannot have a channel to itself. In a *directed* network, messages can travel only in one direction through a channel, while in an *undirected* network, messages can travel bidirectional. A network topology is called *complete* if there is an undirected channel between each pair of different processes. In the topology of the network, nodes only know their immediate neighbors. Communication

in the network is *asynchronous*, meaning that sending and receiving a message are distinct events. However, we will deviate from this later so that synchronous sending and receiving is possible as an atomic event. Channels do not have to be FIFO. This means that some messages can overtake others and the logical order is not guaranteed. Messages can also be lost. So lossy channels can exist.

Causal Order : In asynchronous distributed systems, events that can occur on different processes are independent, which means they can occur in any order. The causal order $\prec$ is a relation between two events. The notation $a \prec b$ describes that *a* must happen before *b*. A rearrangement is not possible. For example, if *a* is a sending event of a message and *b* is the receiving event of that message, then $a \prec b$. If $a \prec b$ and $b \prec c$, then also $a \prec c$. We write $a \preccurlyeq b$ if either $a \prec b$ or $a = b$. Distinct events that are not causally related are called concurrent. A permutation of concurrent events does not affect the outcome of execution. An example of this would be two processes that are simultaneously and independently executing a write operation on their local variables. It is irrelevant for the behavior of the system which process writes first.

Assertions : An assertion is a predicate about the configurations of an algorithm, which is either true or false in each configuration. Assertions formulate safety properties that should apply to a system or in our case to a distributed algorithm. An *invariant* is a property that is satisfied by the initial configuration and all possible subsequent configurations. In general, it is undecidable whether a given configuration is reachable. However, we can use tools such as UPPAAL[29] to analyse special cases.

Sources of Error : The error sources described here relate to communication. In a distributed system, individual nodes can either fail briefly or completely. Messages can also be lost, for example when sent over the Internet or due to faulty cables. When communicating over the Internet in particular, messages may not arrive in the order in which they were sent. The causal order cannot be guaranteed. Distributed algorithms must be developed to be robust against such sources of error.

3.2 UPPAAL: Model-Checker

As a joint project of the Universities of Aalborg and Uppsala, UPPAAL [29][3] was developed as a model checker for timed automata to analyse real-time systems. In this project, the current version 4.1.26 from November 26, 2019 is used, which has a graphical editor, a simulator and a verifier. In templates, the automata can be created graphically and variables and functions can be defined in a C-like language in local and global declarations. A automaton network is formed in UPPAAL from one or more templates, declared by the user. This network formally describes the behavior of a system. With the simulator, traces can be simulated and analysed. Using temporal logic, automaton networks can be verified and examined with the verifier. UPPAAL uses a subset of the Computation Tree Logic (CTL) to make assertions about the behavior of a system. The UPPAAL system is saved in a XML file, a simulated trace can be saved in a separate file. UPPAAL is used in the field of model-checking for real-time systems in business and research. For academic purposes, UPPAAL [29] is available free of charge.

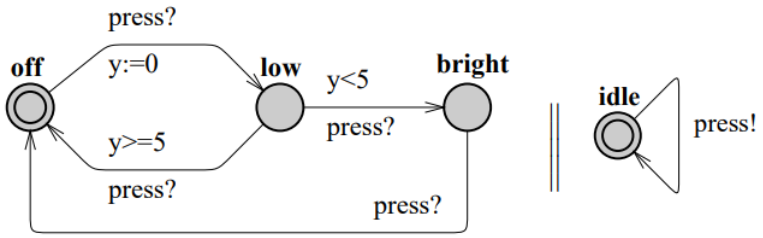
Formal Model : UPPAAL is based on the following definition of a real-time automaton [6]. We consider an alphabet Σ of actions for synchronizing transitions across channels, where UPPAAL distinguishes between sending and receiving actions on a channel. For a channel *ch*, *ch!* is a sending action and *ch?* is a receiving action on this channel. Channels are always synchronous. For the modeling of asynchronous behavior or the exchange of information (also buffered), other concepts have to be developed. UPPAAL also provided broadcast synchronizations, in which a send action synchronizes with all receive actions that are enabled at that point in time. If no receive transition is enabled, the send action on the broadcast channel can also be taken alone. We also consider a finite set *C* of clocks and a set *B(C)* that holds for time conditions over these clocks. A time condition from *B(C)* has the form: $x \sim c, \sim \in \{\leq, <, =, >, \geq\}, c \in \mathbb{N}, x \in C$. Time conditions are used as guards on transitions and as invariants on locations. Furthermore, let P(C) be a set of clocks from C, which are reset with a transition (updates). Other features such as integer and boolean variables for guards, invariants and updates are also used. It is also possible to integrate function code in a C-like language. It is possible to define variables and constants as well as

functions. These can be defined locally for a template or globally. However, these modeling aids (C-Code) are transferred internally to automaton logic. Formally, a timed automaton $A = (N, lo, E, I)$ is a 4-tuple, where

- N is a finite nonempty set of states,
- $lo \in N$ is a start state,
- $E \subseteq N \times B(C) \times \Sigma \times P(C) \times N$ a finite set of transitions and
- $I : N \longrightarrow B(C)$ are invariants of states.

Several Templates can be used to describe a system. This is possible because UPPAAL uses networks of automata. The system declarations specify which templates are used to describe a system. The channels are used to realize interactions between the individual automata.

Example with a lamp [3]: Fig.3.1 shows a timed automata of a simple lamp and a user interaction. These two automata model our system. The lamp (left) has the three locations *off, low*, and *bright*. When the user presses the button, the user automaton (right) sends on a channel *press!*. The lamp is synchronised on its transitions to this channel (*press?*) and switches to the next state. The user can press the button at any time. If the button is pressed again below a time unit of *5* ($y < 5$), the user can switch from the state *low* to *bright*. The clock *y* is used to detect if the user was fast ($y < 5$) and switches to *bright* or slow ($y >= 5$) and switches to *off*.



(a) Lamp. (b) User.

FIGURE 3.1: Simple Lamp Example [3]

3.3 Google GO: Concurrency Programming Language

Go is a programming language developed by Google which supports concurrency in particular. Go also supports object-oriented programming, but it is not class-based like Java. Syntactically, Go is based on C with influences from Pascal, Modula and Oberon. One can find an overview of the syntax in he Go Programming Language Specification [19]. Alternatively, a tutorial from Google can be used to learn Go [17]. The model for concurrency in GO is based on Tony Hoare's work Communicating Sequential Processes (CSP) [15]. In this work, Hoare described a process algebra for describing interactions between communicating processes.

Goroutines : Probably the most important key word in GO is "go". This is used to execute a method or function call in a goroutine. A goroutine is a lightweight thread managed by the Go runtime. The statement consists of the keyword "go" followed by a function or method call (e.g. go doSomething()). Unlike a regular call, the call runs as an independent concurrent thread, also called a "goroutine," within the same address space. Program execution does not wait for the called function to complete, instead the function executes on its own. When the function exits, its goroutine also exits. For example, if you want to wait in the main function for the completion of goroutines, so-called "WaitGroups" from the "sync" package [18] can be used.

Channel : To support message passing for concurrent programming, Go uses the concept of channels. This offers the possibility of synchronous or asynchronous communication between Go routines. A channel is a memory area that is protected by semaphores and provides a queue (buffered/asynchronous channel) or just an interface (unbuffered/synchronous channel). A channel is created with a specific type of data and only that specific type can be send and received on the channel. To create a channel, "make" is called and the data type of the channel is specified (e.g. *ch := make(chan int)*). Values can be written to or read from a channel with the "<-" operator. *ch <- v* sends *v* into a channel *ch*. With *v := <- ch* the variable *v* is assigned the value from the channel *ch*. Channels can be buffered. The size of the buffer is assigned as the second argument "make" (e.g. *ch := make(chan int, 100)*). A channel is always a FIFO. An important building

block when dealing with channels in Go is the "Select" statement. It is similar to a switch case statement in C++, Java or other languages. The select statement has cases for different send and receive statements instead of switching on cases for a specific variable. A *select* blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are enabled. If no channel is enabled, a default case can be set up. This is similar to the "default" in a switch case in C++.

These are the most important aspects we need in this work regarding concurrency. Since we use Go as the specification language, concrete examples with program code follow in the next chapter.

4

SPECIFICATION OF DISTRIBUTED ALGORITHMS IN GO

In this section we consider the specification for distributed algorithms using GO. For this purpose, a subset of GO is presented first, witch is used as specification language. It then describes how a network is basically set up and how communication (synchronous, asynchronous and asynchronous with buffer) between nodes is implemented. A framework for the specification is required, which was developed as part of this work. The *dialgo.go* framework was developed to make the specification easier for the user and to make the automated transfer to UPPAAL easier. This encapsulates building blocks that support, for example, the sending from node A to B or the non-deterministic selection of a value. At the end of this chapter, we consider an example implementation of Huang's algorithm for termination [16] for a distributed system. We will return to this example in later sections. This example illustrates the structure of the net of nodes, the communication between them, simple logic of a distributed algorithm and the use of the dialgo.go framework.

4.1 GO Subset for the Specification

Since GO is a very extensive language in terms of semantics and some concepts such as lists or semaphores are difficult to transfer to UPPAAL with C-like support in the declarations, a subset for GO was defined. This significantly simplifies the automated transfer to UPPAAL. In the following we consider the subset for GO which is used in this work. We take the official GO language specification [19] and restrict it as follows.

4.1.1 Data Types

Since UPPAAL with its C-like language does not support all data types of GO, the data types in the GO specification of the distributed algorithms have to be restricted. The subset of usable data types consists of *Boolean* and *Integer*. In principle, structs can also be translated with little effort, but were not implemented in the prototype.

4.1.2 Declarations of Variables and Constants

Go offers the interface of dynamic typing, but for translation we need static explicit types. The definition of variables must have the form *var* <*name*> <*type*> = <*value*>. Dynamic typing is not allowed. This restriction makes translating easier, since UPPAAL has a C like language. Constants may be defined globally and variables locally in functions. A variable must be initialised immediately. A variable must always be initialised with a value (E.g. *var b bool = false*). initialising a variable with a function call is not allowed. This also applies to the functions from the dialgo.go framework. For example, the statement *var b bool = RandomBool()* would be inadmissible. This is because functions in the UPPAAL model are modeled by automaton logic, but the variables are described in the declarations.

Local variables of a function are declared in the local declarations by translation in the UPPAAL model. To do this, all locally used variables (with the exception of loop counter variables) must be declared and initialised at the beginning of a function block. This means that first all locally used variables are declared in a function and then logic such as if statements, loops, etc. follows.

4.1.3 Declaration and Initialisation of Channels

Channels must be defined globally and always have the form *var <name> [EDGES]chan interface*{}. Since UPPAAL does not support dynamic or generic data types, the channel type must be specified as a prefix (Hungarian notation). For this, the form must be *var <type>_<name> [EDGES]chan interface*{}. If no type is specified (as shown in the code example above), a channel in UPPAAL is always treated as type *int*. An example for a channel of type *bool* would be: *var bool_chanMsg [EDGES]chan interface*{}.

EDGES is a constant that indicates the number of edges in the network. This is defined in the DiAlGo.go file. The initialisation takes place in the main function using a for loop. The following code shows an example for this. The so called "Short Variable Declaration" is used to take a value from a channel (*<Var. Name> := <- <channel>*).

```
 // Channel declaration globally
 var int_chanMsg [EDGES]chan interface{}
 ...
// Init channels
for i := range int_chanMsg {
   int_chanMsg[i] = make(chan interface{}, 1)
}
 ...
 // Read from a channel
 msg0 := <-int_chanMsg[GetReceiveIndex(0, 1)]
```

4.1.4 Statements

This section describes how various statements like *if/else, for, select* and *function calls* must be written in GO. The restrictions are necessary for the translation to UPPAAL.

If/Else Blocks

If/Else blocks can be used as usual. Single *if* statements, statements with *else*, and *else if* with any depth are supported.

For Loops

Only for loops with loop counter variables called *ForClause* or *Boolean* expressions may be used. Other notations such as *range* statements are not allowed. Loops without a condition are also not supported.

Selects

The Go *select* for reading messages from a channel can be used as usual, but must respect all other rules and the style guides. These rules are important for the naming of variables in the context of the *select*. A *select* can contain *causes* and optional a *default* case.

Function calls

Function calls, apart from functions that define a node called from the *main* in context of the *waitgroup*, are not allowed. For example, function calls from nodes (node functions) are not supported. Exactly one parameter must always be passed for the function that defines a node, the *PID*. The *PID* of a node must be unique and from *0* to ascending. For example, the IDs *0,1,2* and *3* exist for a net of *4* nodes.

4.1.5 Style Guide

In addition to the rules of the GO grammar[19], there are further rules for the translation according to UPPAAL, which are defined below.

Naming

- Channel names must not contain underscores (except for typing, e.g. *<type>_<name>*).
- The variable identifier *broadcastCounter* and *sendLossy* are used by the UPPAAL model. The variable identifier *i* is used in the UPPAAL model, but only in a Select Context. So it can be used, but you have to be careful with the RandomInt() function from dialgo.go (E.g. i = RandomInt() does not work.). The variable identifier *nodeCrash* is also reserved when a node crasher is added to switch off nodes.

For Loops

- Infinite loops are initialised with a condition that always evaluates to true. E.g. true or 1 == 1.
- A for loop without a header is not allowed.

Main Function

- The function calls of the nodes may only take place after the call of *Init()* and after the initialisation of all channels.
- Function nodes must always be executed in the context of a *waitgroup*.

4.2 Specification of an Distributed Algorithm as a Network of Nodes

The basic idea is to run a network of nodes in parallel, where each node has its own local storage, a pid, performs independent functions and can exchange information synchronously or asynchronously with other nodes. For this we use the goroutines. A node is defined as a function *f(s)* which is executed in a goroutine (*go f(s)*). The individual functions that specify nodes run asynchronously in goroutines. Since the program simply terminates after the main function has finished, we have to wait for all goroutines to finish. For this we use the concept of *WaitGroups* from the "sync" package [18]. In a *WaitGroups*, the functions (nodes) are started as an anonymous function in a goroutine. As shown in the following code example, the WaitGroup waits until all of the 3 called goroutines in it have terminated. The pid of a node is given to it as the first parameter. E.g. node(0) for a node with the pid 0. The size of the WaitGroup for n nodes is n because all nodes must be waited for. In our example, this is represented by a constant NODES with the value n for n nodes.

```
var wg sync.WaitGroup
wg.Add(NODES)
go func() {
   controlAgent(0)
   wg.Done()
}()
go func() {
   worker(1)
   wg.Done()
}()
go func() {
   worker(2)
   wg.Done()
}()
wg.Wait()
```

4.3 Specification of the Communication between Nodes

The communication for the exchange of information between nodes takes place via GO channels. It is assumed that the network topology is complete. For the number of edges, this means that there are $e = \frac{n(n-1)}{2}$ edges for n nodes. However, two channels are required for each transition between two nodes A and B. One channel for sending from A to B and one for sending in the reverse direction from B to A. This gives us $e = n(n-1)$ edges for n nodes. Thus, for each type of information we want to share between nodes, we need a channel array with $n(n-1)$ channels. The channel type must be of the interface type. This later makes it easier to send generic messages, described in *chapter 4.4*. E.g. *var chanMsg [EDGES]chan interface{}*, where EDGES is the number e of edges. The index of the channel on which node A sends to node B is calculated by

$$(n-1) * pid(B) + \begin{cases} pid(B), & \text{if } pid(B) < pid(A) \\ pid(B) - 1, & \text{if } pid(B) > pid(A) \end{cases}$$

Importantly, sending messages to yourself is not allowed. On which channel index node A receives a message from node B is calculated by

$$(n-1) * pid(A) + \begin{cases} pid(A), & \text{if } pid(A) < pid(B) \\ pid(A) - 1, & \text{if } pid(A) > pid(B) \end{cases}$$

4.3.1 Synchronous Communication

In the case of synchronous communication, the sender and receiver block until they can carry out the action jointly initialised by the sender. That means the sender waits until the receiver is ready to receive and the receiver waits until the sender sends. An unbuffered channel is used for this in GO.

4.3.2 Asynchronous Communication

With asynchronous communication, the channel should only block for the sender if a message is already in transit. This can be achieved in GO using a buffer with a size of 1. The sender sends a message and can then do other things until eventually the receiver gets the message. Sending a second message is only possible after the receiver has received the first message. A select statement with a default case can be used so that the receiver is not blocked by waiting for a message.

4.3.3 Asynchronous Communication with Buffer

The basic idea for asynchronous communication with a buffer > 1 is the same as above. For a buffer of size b, the channel should block as soon as b messages are in transit. If the receiver receives a message, the sender can send another.

4.4 The DiAlGo GO Framework

The DiAiGo framework provides the user with various building blocks for the specification of distributed algorithms. It was developed as part of this work to guide the user and to make the translation from GO to UPPAAL easier. Its functionalities are presented in the following using the GO Syntax. The full code of the DiAlGo GO framework is included in the appendix. In the dialgo.go file, the constants *NODES* and *EDGES* are defined for the number of nodes and edges in the network. These can be used in the user code to initialise channels or to iterate over all nodes in a loop.

Init() : This function must be called at the start of the programme to initialise the seed for randomness.

RandomBool() bool : In order to be able to map non-determinism in GO, randomness is used. This function can be used, for example, if an event is not to occur deterministically. E.g. the random change of a node from active to idle.

RandomInt(min int, max int) (i int) : If a number is to be chosen non-deterministically, this function can be used. It is equivalent to the select statement in UPPAAL, whereby a value from a range of values is selected non-deterministically.

Send(pidA int, pidB int, chans [EDGES]chan interface, msg interface) : If a message is to be sent from node A to node B in a network, the send function is used. This determines the index of the channel in a channel array on which is sent from A to B and sends a message msg on this. Channels and messages can be of any permitted data type, including struct. This is realized by the interface in GO.

SendLossy(pidA int,pidB int,chans [EDGES]chan interface,msg interface) : If it is to be simulated that messages can also be lost, this function can be used. This does send non-deterministically, i.e. randomly a message or not.

GetReceiveIndex(pidA int, pidB int) (index int) : This function determines the index of the channel on which A listens to B.

Broadcast(pid int,chans [EDGES]chan interface,msg interface) : This function serves to broadcast a message in a complete network.

BroadcastLossy(pid int,chans [EDGES]chan interface,msg interface) : This function serves to lossy broadcast a message in a complete network. It is simulated that messages can be lost non-deterministically.

These functions can be used by the user to simulate nondeterministic behavior or selecting a value, and for communication between goroutines. Typical behavior of distributed systems, such as losing a message, can also be simulated. The user code for specifying distributed algorithms consists of the Dialgo.go + a user-written GO file that contains the distributed algorithm.

4.5 Example with Huang's Termination Algorithm

Huang's algorithm [16][1] for termination detection is used to detect that a network of nodes that work distributively and call each other has terminated. For this we consider two types of nodes. A control node (control agent) and a worker node (worker). The idea behind the algorithm is easy to understand. The control agent activates a worker. This starts work and activates other workers. At some point, all workers have completed their work and no longer call each other again. The controlling agent wants to determine this status. For this purpose, it has a weight of 1. It shares this with a worker as soon as it starts a worker. If a worker calls another worker, it shares its weight with the called worker (e.g. wA has a weight of 0.5 and calls wB with a weight of 0. Then after the call wA has a weight of 0.25 and wB as well.). When a worker has completed its work, it goes into an idle state and sends its total weight to the controlling agent. If the control agent has a weight of 1, it knows that no workers are working. The termination is thus recognized. Since UPPAAL does not support doubles, we assume a weight of 4096 for our implementation, which is divided by activation messages. In our example there should be a control agent and 3 workers. In the following we consider certain parts of the implementation. The complete code can be found in the appendix.

Declaration, Initialisation and the Main Function

We need a channel on which we communicate the weight. This is declared as *var int_chanMsg [EDGES]chan interface{}*. We define the weight as a constant globally as *const WEIGHT int = 4096*. The following code shows the main function. Here we first call *Init()* of the *dialgo.go* framework. Here the random seed is initialised. Then the channels of the channel array are initialised. A wait group is now defined, in which the individual nodes of the network are executed using anonymous goroutines. We have a control agent with the PID 0 and 3 workers with the PIDs 1, 2 and 3. In the *dialgo.go* file we have to define the number of nodes (*NODES = 4*) and the number of edges (*EDGES = 12*) as constants.

```
func main() {
    Init()
    for i := range int_chanMsg { // Init chans
        int_chanMsg[i] = make(chan interface{}, 1)
    }
    var wg sync.WaitGroup // Waitgroup of all nodes
    wg.Add(NODES)
    go func() {
        controlAgent(0)
        wg.Done()
    }()
    go func() {
        worker(1)
        wg.Done()
    }()
    go func() {
        worker(2)
        wg.Done()
    }()
    go func() {
        worker(3)
        wg.Done()
    }()
    wg.Wait()
}
```

Control Agent

Th following code shows the function of the control agent. We see that its local variables (*terminated* and *weight*) are initially defined and initialised. First, *Send()* is called to start the first worker. The control agent activates the worker with the pid 1 and sends him half his weight. Now the worker is waited for in a loop until the control agent has reached its weight of 4096 again. To do this, a *select* is used to listen to the worker nodes. The *default* case is there to continue listening insofar as a channel is quiet.

```
// Control agent for worker nodes
func controlAgent(pid int) {
   var terminated bool = false
   var weight int = WEIGHT / 2

   Send(pid, 1, int_chanMsg, WEIGHT/2) // Send first msg

   for !terminated {
      // Listen for getting back weight
      select {
      case msg0 := <-int_chanMsg[GetReceiveIndex(pid, 1)]:
         fmt.Println("PID",pid,"received",msg0,"from PID 1.
        New weight=", weight+msg0.(int))
         weight += msg0.(int)
      case msg1 := <-int_chanMsg[GetReceiveIndex(pid, 2)]:
         fmt.Println("PID",pid,"received",msg1,"from PID 2.
        New weight=", weight+msg1.(int))
         weight += msg1.(int)
      case msg2 := <-int_chanMsg[GetReceiveIndex(pid, 3)]:
         fmt.Println("PID",pid,"received",msg2,"from PID 3.
        New weight=", weight+msg2.(int))
         weight += msg2.(int)
      default:
      }
        // Check termination
      if weight == WEIGHT {
         terminated = true
      }
   }
   fmt.Println("ControlAgent with PID",pid,"terminated!")
   os.Exit(3)
}
```

Worker

The logic of the worker is more complex, which is why it is not illustrated here. However, the full code can be found in the appendix. The worker node runs in an endless loop. This simulates an endless life cycle. It can be active or inactive. If it is inactive, it waits for activation messages from the control agent or from other workers. A *select* in a loop is used for this. The *default* case is there to continue listening insofar as a channel is quiet. When the worker is active, it invokes any number of other workers by sending activation messages. However, it only does this once and not every cycle. Then the *RandomBool()* function of the *dialgo.go* framework is used. To become randomly inactive. Thus, the worker becomes inactive at a random time after any number of cycles. When the worker becomes inactive, it sends its total weight back to the control agent.

5

TRANSLATION OF DISTRIBUTED ALGORITHMS FROM GO TO UPPAAL

The basic idea is to parse a GO file, create the basic structure of the network using UPPAAL templates and build the logic of these template automata using customisable building blocks. The automaton is created through composition. This section introduces the individual "building blocks" and shows how GO blocks are translated into UPPAAL.

Fig.5.1 shows this composition by an example. This example shows how logic in Go functions is transferred into automaton logic. A GO file is read and translated line by line from top to bottom. The machine, on the other hand, is built from the bottom up and thus describes the program flow. In this example we see that there is a send action in the GO file after, the program start. This is transferred to a sub-automaton in the UPPAAL model. The UPPAAL automaton is composed of sub-automata. After the send action comes an *If/Else* block. This leads to two traces in our UPPAAL model, the program flow is divided into *If* and *Else*. These sub-automata for *If* and *Else* are also transferred in the UPPAAL model to sub-automata, which are added to the UPPAAL Template.

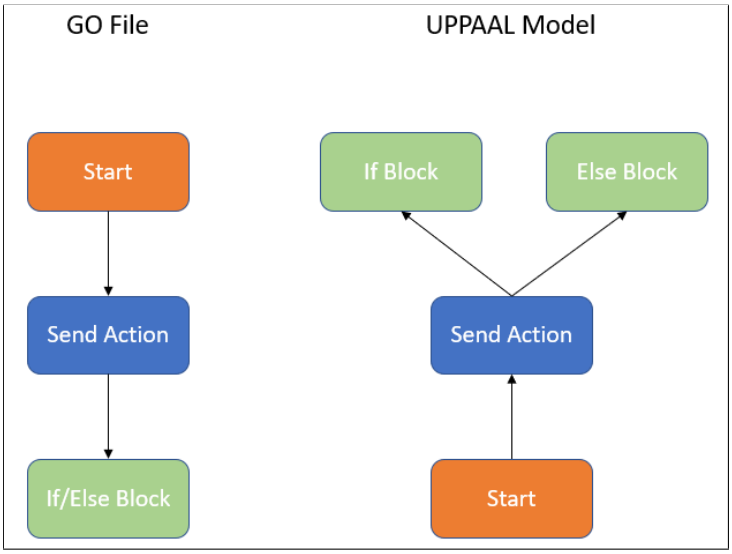


FIGURE 5.1: Composition - Go to UPPAAL

5.1 Parsing GO

We use Antlr4 [24] in the version 4.10.1 to parse GO files. Antlr (ANother Tool for Language Recognition) is a parser generator, which generates a parser from grammars (.g4 files) that can create and run through parse trees. Antlr can easily be integrated into Java using the Maven build tool [13], for example. Antlr is widely used to build languages, tools, and frameworks.

We used the grammar [8] from the antlr GitHub repository which contains a collection of antlr grammars for various programming languages. From this grammar (*GoParser.g4* and *GoLexer.g4*) a parser, lexer and listener is generated using Antlr4, which are used to parse GO files.

The original grammar has an error so that no parser can be generated. With the *GoParser.g4* grammar, the replacement rule for *eos* had to be corrected. The *EOS* token is used for any newlines, semicolon and multi-line comments. The token can be replaced to an *EOF* (end of file) token. This causes the (*EOF_CLOSURE*) error in the replacement rule for "*statementList: (eos? statement eos)+*". It is possible to produce *(EOF)+* which is not allowed in antlr grammars. Since an *EOF* should only stand at the end of a file (*sourceFile: .. EOF;*), the *EOF* replacement rule must be removed from the *eos* rule. The new rule has the form *eos: SEMI | EOS | {closingBracket()}?*.

5.2 Translating the Network of Nodes into UPPAAL Templates

As described in *chapter 4*, functions called in goroutines specify the nodes in a network. A *WaitGroup* determines which functions are nodes. These functions are modeled in UPPAAL using templates. Such a template is initialized with parameters such as the unique PID which each node requires. In this automaton template, all local variables used are defined in the local declarations and the logic of the function, i.e. the logic of the distributed algorithm, is modeled in the automaton template.

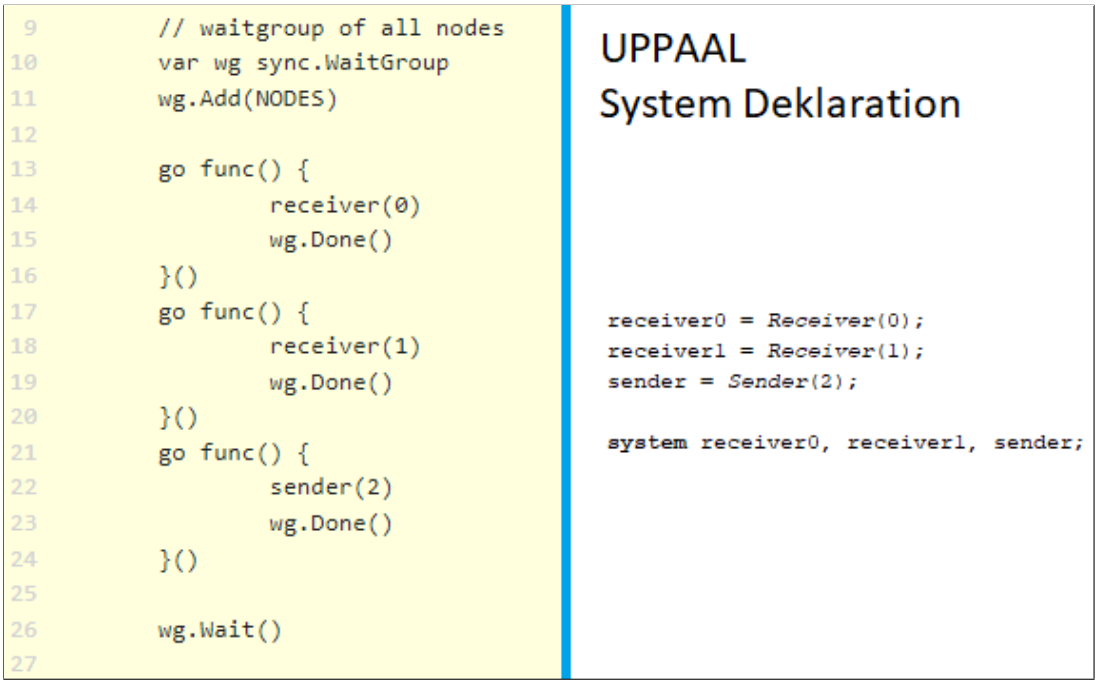


FIGURE 5.2: Network of Nodes from GO to UPPAAL System Declaration

On the left side Fig.5.2 shows the definition of a network of 3 nodes in GO, consisting of a sender (pid 2) and two receivers (pid 0 and 1). The structure of the UPPAAL project is shown on the right side. A template is created for each of the two functions "sender" and "receiver", which are initialized in the system declarations with the appropriate parameters (Pid). The UPPAAL network thus consists of 3 automata (one sender and two receivers).

5.3 Translating GO Channels and Communication into UPPAAL Logic

The following sections describe how communication between nodes (goroutines) is implemented in the UPPAAL model. The specification of a distributed algorithm in GO uses functions of the DiAlGo framework, which is presented in *chapter 4*. For example, so that sending can be implemented generically in a framework, GO channel and data types of the *interface{}* type are used. Since there are no generic types in UPPAAL, the corresponding variables and channels can only ever be of one type (e.g. int or bool). This is particularly important to take into account when modelling. Otherwise, a translation from GO to UPPAAL is not possible.

For communication via channel, variables are used as buffers in the global declarations in all variants (synchronous, asynchronous, asynchronous with buffer and non-FiFo). For each channel array there is a buffer array which is named with the name of the channel array + Var. These buffers are arrays of booleans for boolean channels or arrays of integers for integer channels. They buffer the elements that a sender sends to a receiver. If a buffer is full, the UPPAAL model blocks, as it does in GO.

5.3.1 Synchronous Communication

In the case of synchronous communication, channels are used in the UPPAAL model, so that the sending and receiving action represent a single action that takes place at the same time. Since no data can be exchanged via the UPPAAL channel during synchronisation, a global variable is required to temporarily store the information. We take a look at the following example: A sender wants to synchronously send an integer value to a receiver. In the GO specification of the system, a channel of the *interface{}* type is used, as well as the Send(...) function of the DiAlGo framework. Two functions are written *sender(pid int)* and *receiver(pid int)* which are executed in a wait group from the main function in a go routine (Fig.5.3). In the UPPAAL model, an array of int *chanMsgVar* is used as the global variable in which the data (int) for sending is temporarily

stored. Since a complete network is generically assumed, an array of the size of the number of edges is used. In this case we have 2 nodes and therefore 2 edges. A channel is also required for synchronisation (*chan chanMsgChan* in the global declarations). Fig.5.3 shows the example. The sender writes the value 42 in the *chanMsgVar* int array in place of the receiver's corresponding index, and the receiver takes the value from its corresponding index. (The calculation of the index is described in *chapter 4.*) Since the communication is synchronised via the *chanMsgChan* channel, both automata take the corresponding transition at the same time. Since several receivers can wait for a message from the sender, the *chanMsgChan* channel must also be an array. The size of the array corresponds to the number of nodes. A sender synchronises on the channel on the index of its pid and a receiver receives on the pid of the sender whose message is being waited for. In the example on *chanMsgChan[0]*.

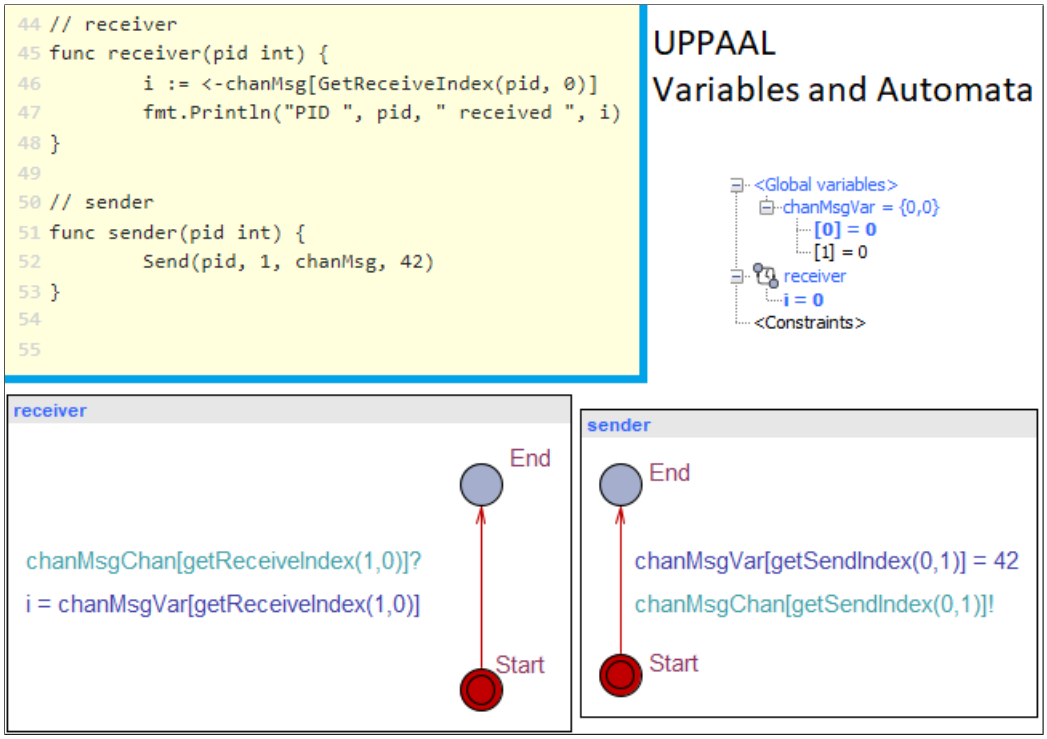


FIGURE 5.3: Example for Synchronous Communication

To calculate the indices when accessing the *chanMsgVar* array, a *getReceiveIndex(int pidA, int pidB)* or *getSendIndex(int pidA, int pidB)* function is called. This is implemented analogously to the calculation rule according to *chapter 4.3*, as

it is also used in the DiAlGo framework in GO.

$$(n-1) * pid(B) + \begin{cases} pid(B), & \text{if } pid(B) < pid(A) \\ pid(B) - 1, & \text{if } pid(B) > pid(A) \end{cases}$$

For sending pid A to pid B.

$$(n-1) * pid(A) + \begin{cases} pid(A), & \text{if } pid(A) < pid(B) \\ pid(A) - 1, & \text{if } pid(A) > pid(B) \end{cases}$$

For receiving pid A listens to pid B.

5.3.2 Asynchronous Communication

For asynchronous communication between two nodes we consider a similar example as before (Fig.5.4). A sender wants to send a message asynchronously to a receiver. For this we no longer need synchronisation in the UPPAAL model. We also have our array *chanMsgVar*, in which the data is cached globally and which the receiver takes when receiving, and also an array with the amount of messages in transit on the channel (*chanMsgSize*). Initially, the size of each channel is 0 since there are no messages in transit. If a message is sent, the number of messages in transit is incremented at the corresponding point in the array. The index is determined using the *getReceiveIndex(int pidA, int pidB)* or *getSendIndex(int pidA, int pidB)* function. The size of the array corresponds to the number of edges. If a message is taken, it is decremented at the corresponding point in the array. Since a message can only be received if the number of messages in transit is > 1, the receiver has a corresponding guard ($chanMsgSize[getReceiveIndex(pid, 0)]! = 0$). Since the channel should block, as in the implementation of the algorithm in GO, if a message is already in transit, the transition to send may only be taken if the number of messages in transit is 0. A global constant is created for this purpose, which specifies the size of the channel's buffer ($constintchanMsgBuffer = 1$). This is similar to the implementation in GO, where the channel also has a size 1 buffer.

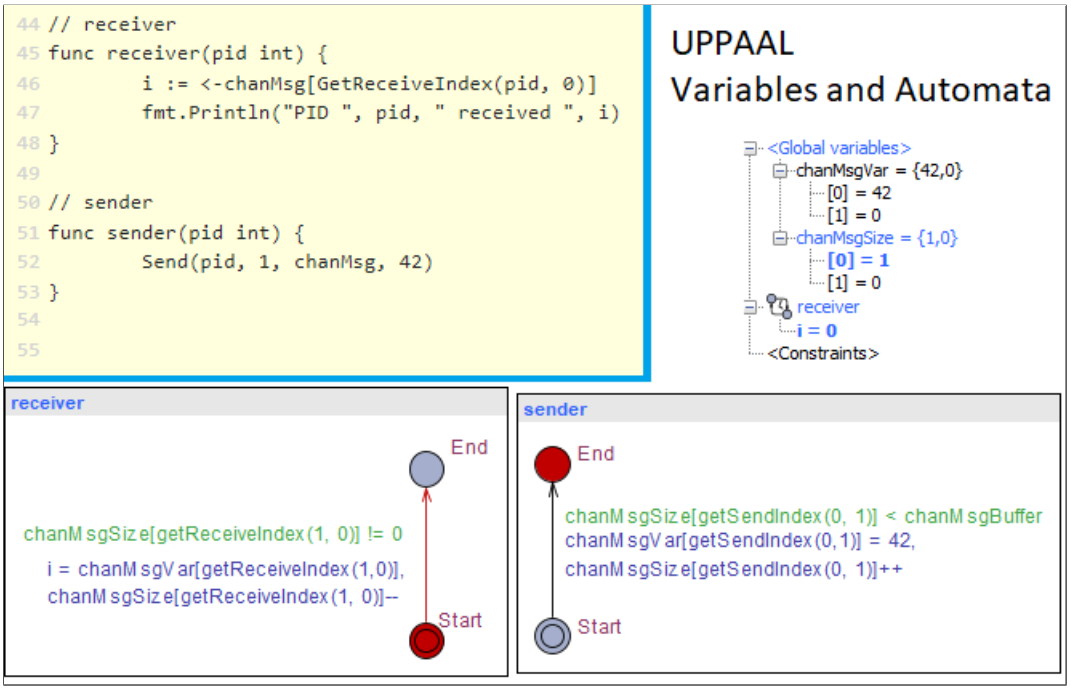


FIGURE 5.4: Example for Asynchronous Communication

5.3.3 Asynchronous Communication with Buffer

For asynchronous communication with a buffer > 1 we consider the following example. A transmitter first wants to send the value 42 and then the value 420 to a receiver. The channel should have a buffer of size 2. The following schedules are therefore possible a) the sender sends both values one after the other and the receiver then extracts them or b) the sender sends the first message, the receiver extracts it, then the sender sends message 2 and the receiver extracts it. The example is shown in Fig.5.5.

To do this, we extend our UPPAAL model from the previous example as follows. We now need a 2 dimensional array for the *chanMsgVar* array. In addition, we need an array that stores the respective index for reading and writing on the *chanMsgVar* array. These are the arrays *chanMsgReadIndex* and *chanMsgWriteIndex*. The size of the arrays corresponds to the number of edges and the corresponding indices can be accessed via the *getSendIndex(int pidA, int pidB)* and *getReceiveIndex(int pidA, int pidB)* functions. If a value is written, then *chanMsgVarIndex* is written in the place of the send index in the array *chanMsgVar*

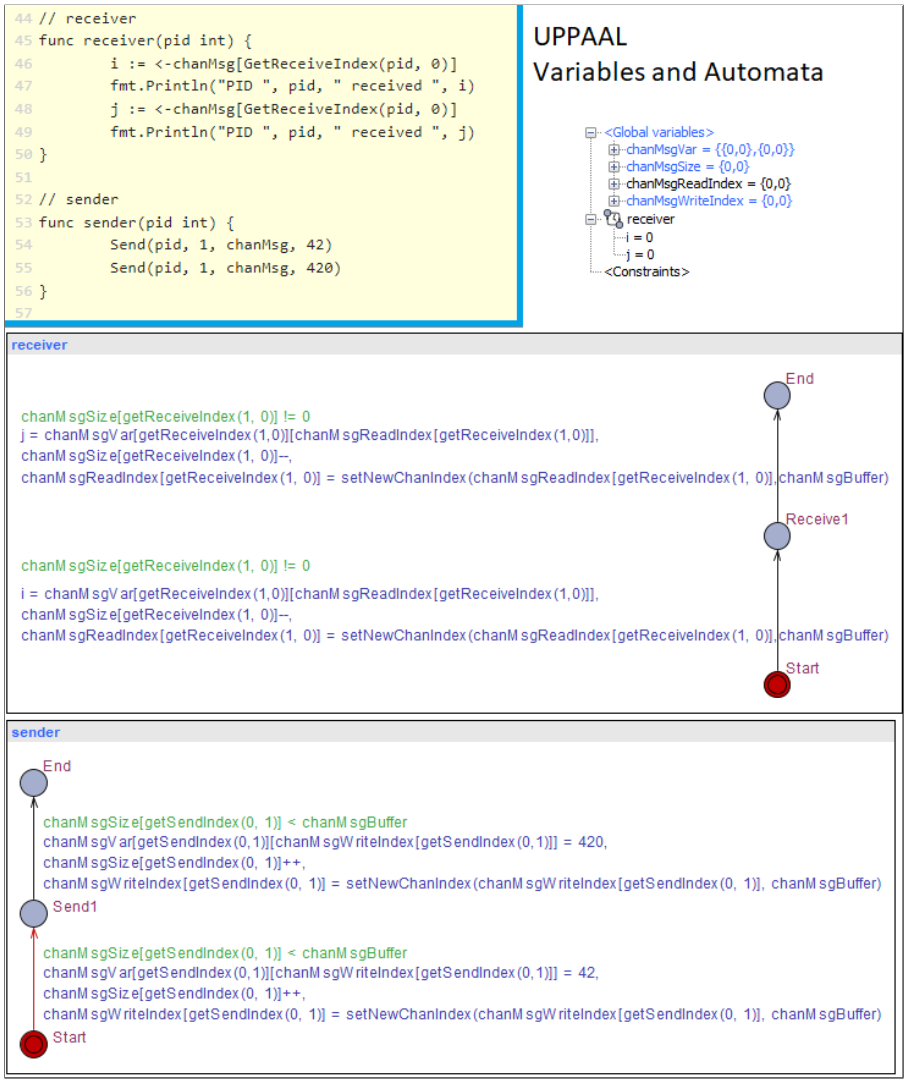


FIGURE 5.5: Example for Asynchronous Communication with a Buffer (Buffer_Size = 2)

of the send index from Pid A to Pid B in the place of the write index (*chanMsgVar[getSendIndex(pid,1)] [chanMsgWriteIndex[getSendIndex(pid,1)]]*). The same applies to reading, corresponding to the read index (*chanMsgReadIndex*). To recalculate the read or write index after a read or write action, a UPPAAL function *setNewChanIndex(int index)* is used. This function determines the new index based on the old one. If the $index == chanMsgBuffer - 1$, the new index is 0. Otherwise the index is simply incremented. The implementation of the buffer in UPPAAL by an array with an index variable for writing is a ring buffer.

5.3.4 Broadcasting Messages

The translation of the Broadcast function from the DiAlGo framework for broadcasting messages to all participants in a network is trivial. In the UPPAAL model, all send actions are simply executed one after the other. This corresponds to a 1 to 1 translation of the behavior of the algorithm in GO with the DiAlGo Framework. In a loop over all nodes, the send action is executed when the loop counter does not match the sender's PID. In our case, a broadcast action is a send action for $n - 1$ nodes in a network of size n.

UPPAAL natively offers a broadcast function, whereby all send actions are executed synchronously as one action, but this is not used. Broadcast channels allow 1-to-many synchronisations with UPPAAL automata. An transition with an emit-synchronisation on a broadcast channel can always fire (provided that the guard is satisfied). One way to model this UPPAAL behavior in GO would be using the fan-out or publish-subscribe pattern. The basic idea of this pattern is that several receivers listen to one input source. However, using this only synchronous communication (in GO a channel without a buffer) would be possible. Asynchronous broadcasts would not be possible with this modeling.

5.3.5 Non-FiFo Channels

When messages are transmitted over the Internet, the temporal order of the receiver may not match that of the sender. This behavior can be simulated by non-FiFo channels. Therefor, we want to model a non-FiFo buffer where it is random which value is taken from the buffer.

First we take a look on the basic idea. Data that is transmitted is held in a ring buffer. We need a write index so that the sender knows where to write (see *chapter 5.3.3*). In addition, we need an array *A* of the size of the buffer, which maps the state in the buffer. If a value in array *A* is *true*, then there is a value at the corresponding index in the buffer. If a value in array *A* is *false*, then there is no value at this index in the buffer. With each write operation, the sender sets the corresponding value in array *A* to *true* and when the receiver takes it, it sets the value to *false*. If the receiver wants to take a non-FiFo value from the

buffer, he selects a random index, which must be *true* in the array *A*, and takes the corresponding value from the buffer. Fig.5.6 show the data structure with an example.

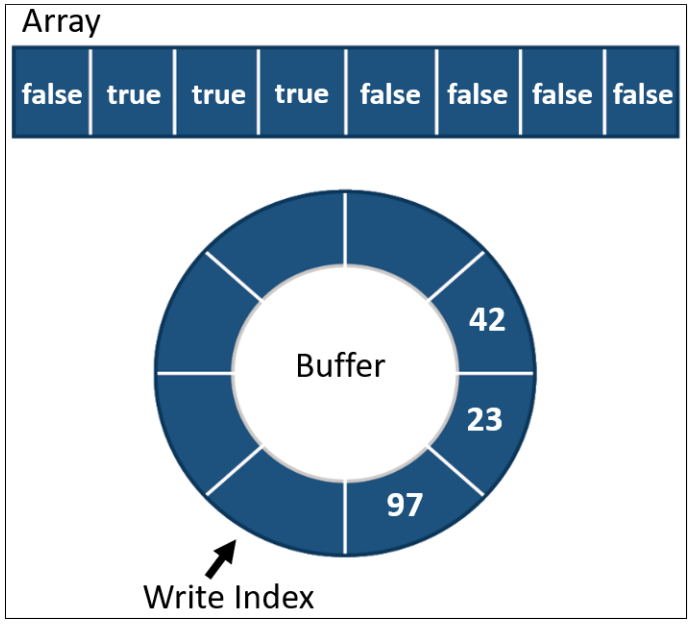


FIGURE 5.6: Non-FiFo Ring Buffer with Array

In order to use a non-FiFo channel or a non-FiFo ring buffer in the UPPAAL model, we change our example from *chapter 5.3.3* as follows. We no longer need a reading index. Instead, we need a boolean array (*chanMsgState*) that maps the state of the buffer. As in example 5.3.3, this must be 2-dimensional, i.e. we need a boolean state array for all edges. When sending, the sender now writes true at the corresponding index into the array *chanMsgState*. Since it should be non-deterministic which value is taken from the buffer, the receiver uses the UPPAAL select to choose a non-deterministic value between 0 and the buffer size. If the corresponding index in the array is *false*, it must be chosen again. However, if true is at the index in the array, the corresponding value is taken from the *chanMsgVar* array, the size (*chanMsgSize*) is decremented and "*false*" is written at the corresponding index in the *chanMsgState* array. Fig.5.7 shows the receiver automaton. In a loop, the automaton takes values from the buffer, which a sender sends as in 5.3.3. For the sender, the calculation of its write index changes as follows. Before each send action, the write index (*chanMsgWriteIndex*) is calculated as a first step. A UPPAAL function (*setNewChanIndexNonFifo(bool chanState[EDGES][chanMsgBuffer], int pidA, int pidB)*) is used for this. It checks

the corresponding array with the state of the buffer and returns the first index that is free in the buffer. This index then becomes the new write index. For example, in Fig.5.6 we see that the first entry in the buffer is free. This occurs, for example, when the receiver has taken the first element. Accordingly, in the next step, the write index is adjusted and points to the first element (the element to the left before 42). If the receiver takes an element from the middle, the sender will write the next element to this taken index. For example, if from a buffer *b* = {*10, 2, 42, 5, 10*} the 2nd element (42) has been taken, the next element will be written at index 2. Fig.5.8 show the implementation of this UPPAAL function. It is called as an update for the write index.

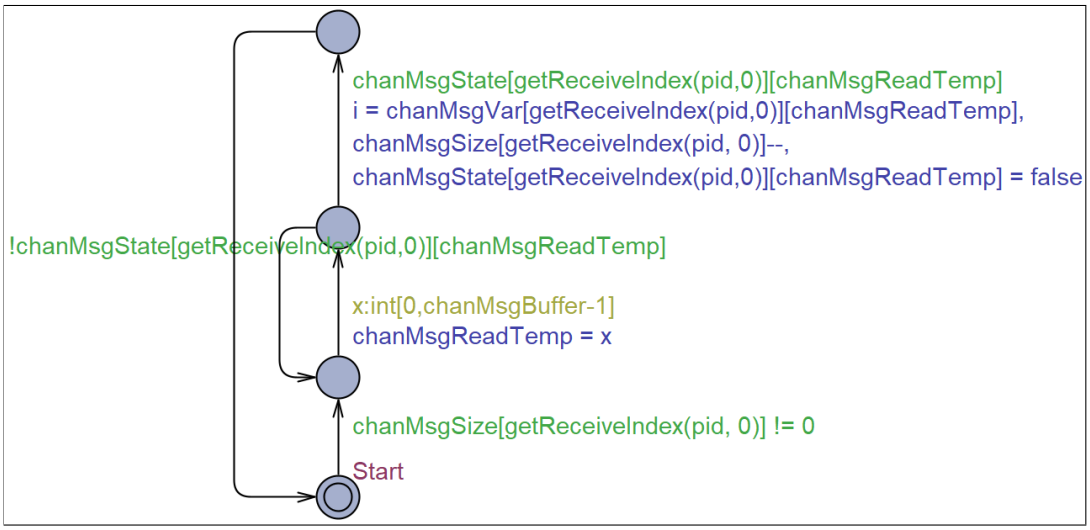


FIGURE 5.7: Non-FiFo Channel UPPAAL Example

```
int setNewChanIndexNonFifo(
            bool chanState[EDGES][chanMsgBuffer],
            int pidA, int pidB){
    int i;
    for(i = 0; i < chanMsgBuffer; i++){
        if(!chanState[getSendIndex(pidA, pidB)][i]){
            return i;
        }
    }
    return -1;
}
```

FIGURE 5.8: Non-FiFo Write Index

When reading a value, the non-deterministic select of an index can lead to an endless loop. Instead of selecting a new index, we can select the next free index from the selected index. In this way, an infinite loop is avoided, but not all indices are equally probable. The probability of indices at the edges increases. However, the probability is not relevant for UPPAAL models. This alternative solution can be implemented in a UPPAAL function, which is then executed as an update if the selected index is empty. The transition, which leads back to the select, then leads to the next state. It is equivalent to the other send transition, but calls as first step the described function. Alternatively, an intermediate state can be used, in which it is checked whether the index is empty and then either executes the function or goes directly to sending. This model is much clearer, since the update logic of sending only occurs once.

5.3.6 Translation of the GO Select Statement

The select statement is used in go to read elements from a channel without blocking if no elements are present. Prioritisation among the cases does not exist in GO.

We consider the following example. Two senders send asynchronous without a buffer random messages to a receiver in a loop. The receiver receives messages from the two senders in a loop, taking a message whenever there is a message on a channel in transit. If there is no message, nothing is done in the default case. The code of the receiver is shown below.

```
func receiver(pid int) {
    for true {
        select {
        case msg0 := <-chanMsg[GetReceiveIndex(pid, 2)]:
            fmt.Println("PID ", pid, "received", msg0)
        case msg1 := <-chanMsg[GetReceiveIndex(pid, 3)]:
            fmt.Println("PID ", pid, "received", msg1)
        }
    }
}
```

In order to translate the select statement in the UPPAAL model, a location is generated for each of the 3 cases (*Msg0, Msg1* and *Default*). The receive actions according to *chapter 5.3.2* are carried out at the two receive actions (*Msg0* and *Msg1*). The translated UPPAAL model for the receiver is shown in Fig.5.9. Since Go has no prioritisation among the cases, no prioritisation of transitions in UPPAAL has to be considered either. The only prioritisation must be set for the default case. The default transition may only be enabled if there is no message in the buffer on any other channel. This is realised via a corresponding guard at the default transition. The buffer size of all other channels must be 0.

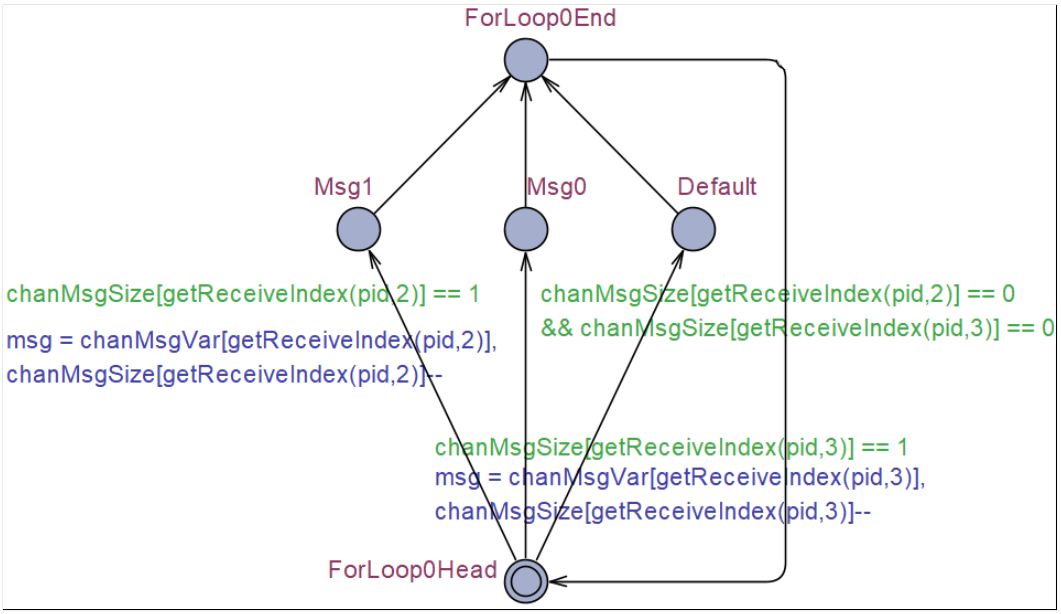


FIGURE 5.9: Example for the GO Select-Statement in the UPPAAL

With synchronous communication, things are a little different. Here, the default transition has no guard and is therefore always enabled. However, this is not a problem since the other transitions are synchronised with the sender. The sender sends on a channel (!) and the receiver synchronises in the select (?). So when a message is sent, the receiver must synchronise and cannot take the default transition. The default transition can only be taken by the receiver before the sender sends a message. This exactly models the behaviour of channels without buffers in GO.

5.4 Simulating Errors in Communication

With the DiAlGo Framework and the DiAlGo Translator it is possible to simulate some errors. For simulating errors we consider the following error cases which we then present in detail.

- A message is lost in transit
- The order of the messages is swapped, so we have non-FiFo channels
- A node crashes

5.4.1 Lossy Channels

Lossy sending means that a message can get lost. In the GO specification of a distributed algorithm, this is implemented using the *SendLossy(...)* function of the dialgo.go framework (*chapter 4*). In UPPAAL, the translation of this behavior is similar. A non-deterministic decision is made between two transitions, with the message being sent on one trace and not on the other. Then, the user can select one of two traces in the simulator. A specific error behavior can be analysed in the simulator and the general behavior can be checked with the verifier.

For this we consider the following example, in which a sender sends a message to a receiver, whereby the message can be lost. Fir.5.10 shows the sender in UPPAAL. The receiver is the same as in the examples above.

```
func sender(pid int) {
  SendLossy(pid, 1, chanMsg, 42)
}
func receiver(pid int) {
  i := <-chanMsg[GetReceiveIndex(pid, 0)]
  fmt.Println("PID ", pid, " received ", i)
}
```

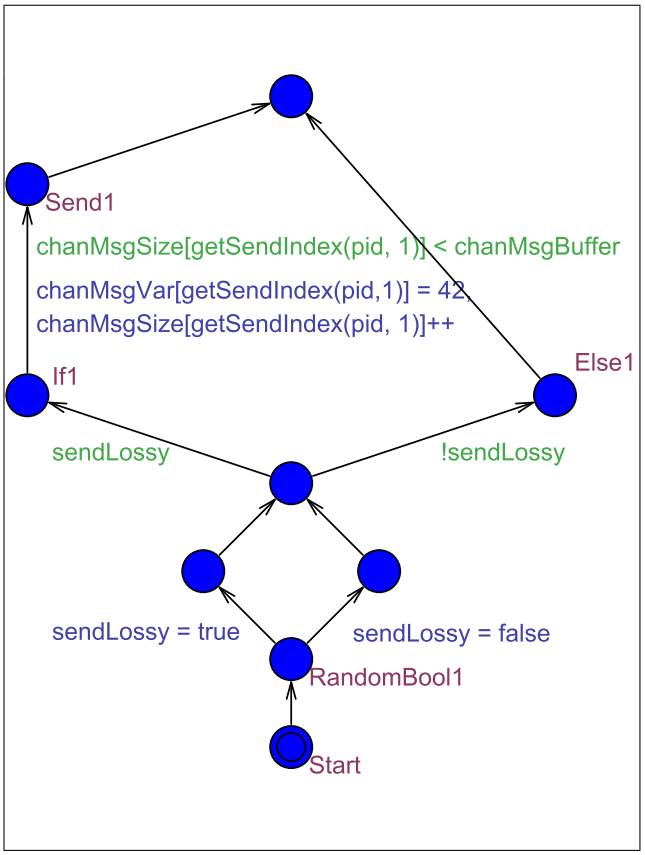


FIGURE 5.10: Sending lossy a Message

A lossy broadcast can be implemented analogously. Sending actions are carried out in a loop, with a non-deterministic decision being made in each run between two traces as to which is to be sent or not.

5.4.2 A Node crashes

In the UPPAAL model, a node can be crashed by an external automaton. The idea is that each transition of an automaton is linked to a condition (*guard*). In an array (*nodeCrash*) we hold Boolean which indicate whether a node has crashed or not. The external crash automaton can then non-deterministically set a value of this array to *true* or *false* using of a *select* to select the crashing node (index in the array). Fig.5.11 shows this automaton. Each transition of the other automata must now be annotated with a *guard*, which is fulfilled if the corresponding value in the array *nodeCrash* is *false*, i.e. the automaton has not

crashed. For transitions without a *guard*, this is annotated and for transitions with a *guard*, it is added using a "and" (<*condition*> && *not crashed*). The user can now select at any time which node should crash and whether a node should become active again. The behavior can then be analysed in the simulator and properties can be checked with the verifier. It is important that a node can become active again, but does not have to. This should reflect the real behavior of a distributed system.

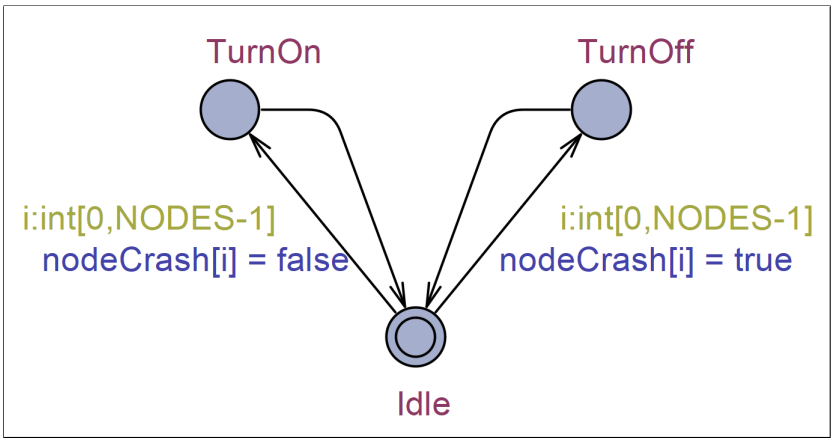


FIGURE 5.11: UPPAAL Crasher Automaton for simulating crashing nodes

If behavior is to be modeled where nodes can never come back online, the "TurnOn" state can be removed. In this way it is only possible to switch nodes off but no longer on.

5.5 Translating Non-Deterministic behavior

With the DiAlGo Framework it is possible to simulate non-deterministic behavior. A random bool can be generated with the RandomBool() function, for example to send or terminate randomly. Likewise, the function RandomInt(min int, max int) can be used to generate a random integer in a specific range between . This behavior can be translated into UPPAAL quite easily. A bool can be assigned the value True or False by non-deterministic selection of an edge, and a value from a specific range can be randomly selected in UPPAAL by a select. Fig.5.12 shows

this. On the left the non-deterministic choice of a bool and on the right a value being chosen by a select.

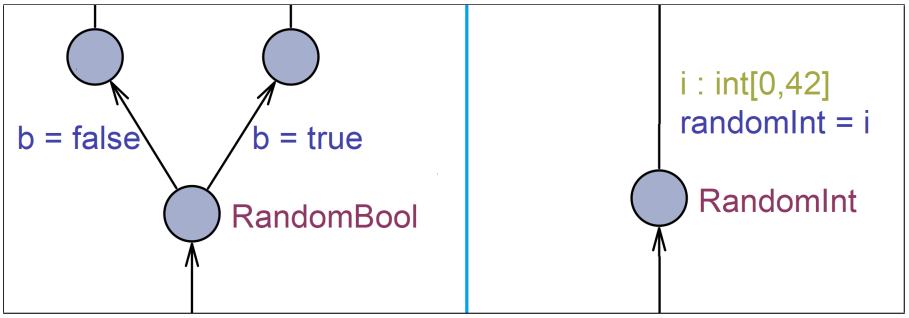


FIGURE 5.12: Non-Deterministic Bool and Select

5.6 Translating Loops

For loops there is only the keyword *for* in GO. While loops, as known from other well-known programming languages such as C or Java, also exist in GO, but are also built with the keyword *for*. For our implementation, we consider two cases of loops. Counting for loops and loops with an expression (while loops). Loops across a range are not considered. Arne Philipeit [26] has already done some basic work on loops. He investigated the range of channels. We extend and improve on these for our own purposes.

Fig.5.13 shows a small example to get the idea. From the loop head *For1*, a transition leads to the start of the *for body* (*Condition1*) and another to the end of the loop (*id4*). The guard for the transitions is determined by the loop head (*For1*). In our example, an integer x should be $x < 42$, that's how long the loop will remain. If the value is $x >= 42$, the loop should be exited. The counter-condition to the loop head can be determined via the complement to the loop condition ($!(x < 42)$). The loop then increments x and the next transitions leads back to the loop head (*For1*).

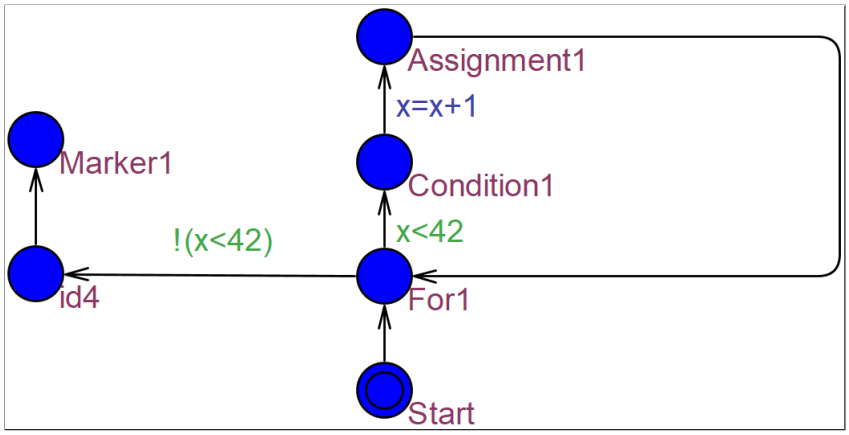


FIGURE 5.13: Translation For-Loops

5.7 Translating If/Else Blocks

The translation of *If/Else* blocks with *Else If* statements can be implemented trivially as an extension to the *If/Else* block translation according to Arne Philipeit's work [26]. Fig.5.14 shows a simple example with an *If*, an *Else If* and an *Else* Statement. It is important to adhere to the priorities given by the order of the statements. For this reason, not all statements can be locations at a branch. Basically, for every Statement(*If*, and every *Else If*) an *If* and an *Else* Location is generated. A transition with the corresponding condition (in the example $i == 42$ and $i == 420$) leads to the *If* and *Else If* location and the complement of the *If* and *Else If* condition to the *Else*. For each additional *Else If*, another branch is added and the tree becomes deeper.

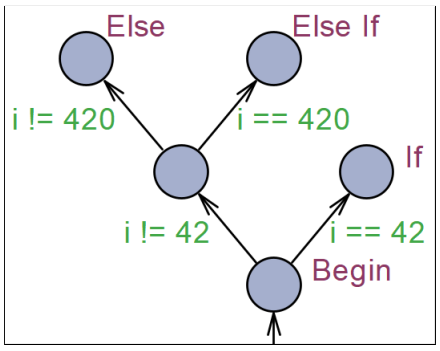


FIGURE 5.14: Example for If/Else Block translation

5.8 Translating Assignments

The translation of assignments is quite trivial. An assignment in our UPPAAL models is always an update of a variable in the local declarations of a template. An assignment like e.g. $i = 42$ becomes a new location in the UPPAAL model with a transition at which an update is performed. The label of the update then corresponds to the assignment. An example of this is shown in our For loop (Fig.5.13). Here *x* is incremented at the end of the loop (update on transition to *Assignment1*).

The translation is different for assignments that call a dialgo.go function. For example, an assignment can call the *RandomInt(..)* or *RandomBool(..)* function to assign a random integer or boolean. In this case, the corresponding function call is used and the appropriate logic for the translation (see *chapter 5.5*) is used.

5.9 Translating Marker Statements

It may be interesting to create marker locations for print statements. These can be used, for example, for reachability analyses with the UPPAAL verifier. A marker location is a location that is entered without guard and has no side effects. In our prototype, the generation of such marker locations can optionally be switched on (with command -m). Fig.5.13 shows a marker location in our *for* loop example. The location *Marker1* is in the GO code a print statement used to check that a loop has terminated.

An example for using marker states in verification is a print statement in the default case of a select statement. If we want to verify that the default block is reachable, the marker state can be used here (*E<> automation.MarkerLocation*). One use like this example is shown in the queries of the following example with Huang's algorithm.

5.10 Example with Huang's Termination Algorithm

Now we take a look at our example from *chapter 4*, an implementation for Huang's algorithm. The two functions (worker and controlAgent) are translated from the GO file to two templates of the same name. Since the worker is quite complex, we only take a closer look at the control agent here, as in *chapter 4*. The full model is included in the appendix.

The local variables that were declared first in the function are in the template's local declarations. Likewise, all other variables used, such as loop counters, are defined in the local declarations. The overall structure of the network is determined from the wait group in the main function. In the system declarations, 3 worker templates with the PIDs 1, 2 and 3 and a control agent with the PID 0 are created accordingly. Fig.5.15 shows the GO Code of the *controlAgent* function. In the following, we will only look at the translation of the control agent, as well as verification for it.

Fig.5.16 shows the Control Agent template. First, a send action is performed (*Send1*). The first message with half the weight is sent or written to Worker1 with PID 1 asynchronously, i.e. in a buffer of size 1. Then a loop (*For1*) is entered, which is only exited when all workers terminate and have sent their weight back to the Control Agent. A Boolean flag named *terminated* is used for this (*Condition1*). To exit the loop, the complement of the condition for Location *Condition1* is formed, i.e. *!<Condition1s condition>*. A select is then made in which the 3 workers are listened to as to whether they have sent back a message with their weight. If there is a message in the buffer (buffer size *!= 0*), it is retrieved and added to its own weight by a subsequent assessment (*Assinment1, Assinment2* and *Assinment3*). If all 3 buffers are empty, i.e. the buffer size at all indices in the array *== 0*, the system switches to *Default1*. After this select with 3 cases and a default case, the branches are merged. An *If* block follows (*If1* or *Else1*). Here it is checked whether the weight corresponds to the full original weight again. *WEIGHT* is a global constant which is also global in the UPPAAL model and was defined in the global declarations. The loop is then completed and it is switched back to the loop head (*For1*).

```
func controlAgent(pid int) {
    var terminated bool = false
    var weight int = WEIGHT / 2
    Send(pid, 1, int_chanMsg, WEIGHT/2) // Send first msg
    for !terminated {
        select { // Listen for getting back weight
        case msg0 := <-int_chanMsg[GetReceiveIndex(pid, 1)]:
            fmt.Println("PID",pid,"received",msg0,"from PID 1.
          New weight=", weight+msg0.(int))
            weight += msg0.(int)
        case msg1 := <-int_chanMsg[GetReceiveIndex(pid, 2)]:
            fmt.Println("PID",pid,"received",msg1,"from PID 2.
          New weight=", weight+msg1.(int))
            weight += msg1.(int)
        case msg2 := <-int_chanMsg[GetReceiveIndex(pid, 3)]:
            fmt.Println("PID",pid,"received",msg2,"from PID 3.
          New weight=", weight+msg2.(int))
            weight += msg2.(int)
        default:
        }
        if weight == WEIGHT { // Check termination
            terminated = true
        }
    }
    fmt.Println("ControlAgent with PID",pid,"terminated!")
    os.Exit(3)
}
```

FIGURE 5.15: Control Agent GO Code for Huang's Algorithm

The translation is done with the developed prototype DiAlGo Translator. The input is the user code with the specified algorithm (Fig.5.15) from Huang and the output is the UPPAAL XML (Fig.5.16) with the translated model. Then, we can analyse this model with the UPPAAL simulator and the verifier.

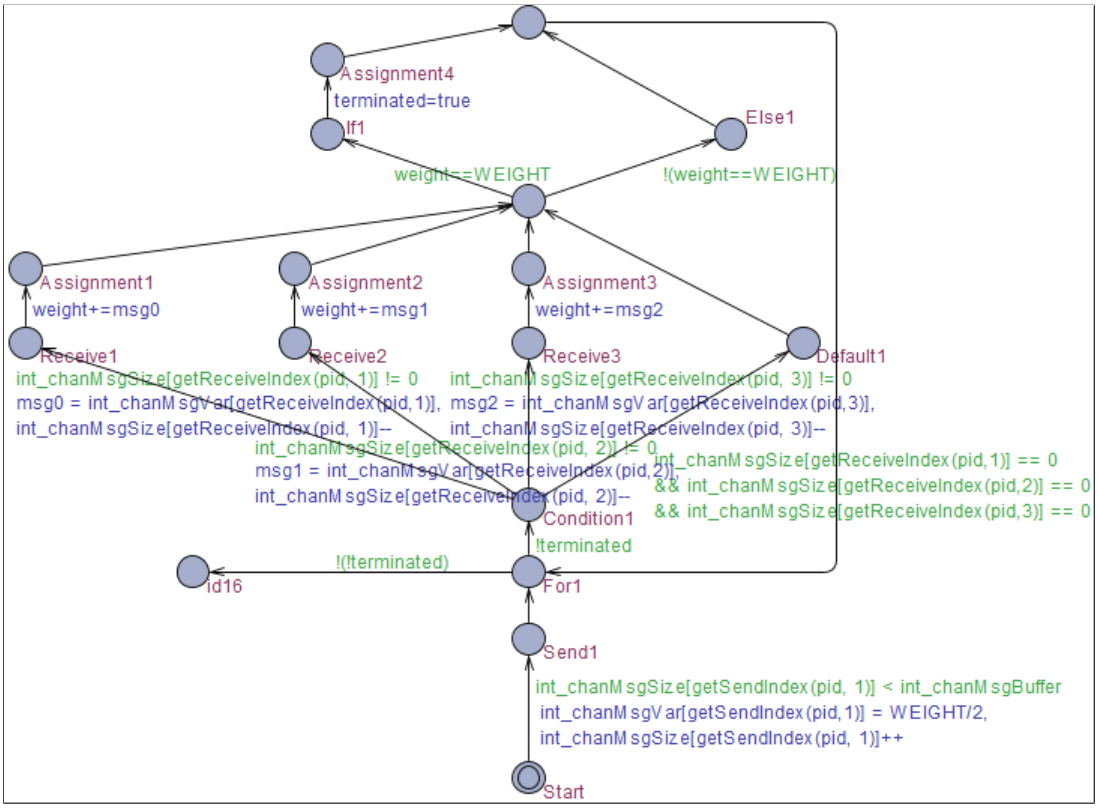


FIGURE 5.16: Control Agent Template for Huang's Algorithm

We can now formulate queries to analyse the behaviour of our specified algorithm. We can use the verifier to generate traces for simulation.

The first verification we look at is an reachability check for termination. The following query shows that a state of termination can be achieved. We can create a trace for this and analyse it. We get the following trace from the verifier. $controlAgent0 \xrightarrow{2048} worker1, worker1 \xrightarrow{1024} controlAgent0$

$$E <> controlAgent0.Marker4$$

The second verification is also an reachability check. We want to generate a trace in which the *ControlAgent* reaches a weight of *3584*. We get the following trace from the verifier. $controlAgent0 \xrightarrow{2048} worker1, worker1 \xrightarrow{1024} worker2, worker2 \xrightarrow{512} worker1, worker1 \xrightarrow{1536} controlAgent0$

$$E <> controlAgent0.weight == 3584$$

6

SIMULATION AND VISUALISATION OF DISTRIBUTED ALGORITHMS WITH UPPAAL TRACES

This chapter describes how distributed algorithms can be simulated, analysed and visualised using the UPPAAL [29] traces. The UPPAAL simulator and the verifier are used for this. Furthermore, it is shown why it makes sense to abstract some steps of the internal logic of a node and to develop an alternative visualiser. For this purpose, we will demonstrate with our prototype DiAlGo Viewer how such a tool can be implemented.

6.1 Usage of UPPAAL Traces for Simulation

Traces can be created and analysed using the UPPAAL simulator. The simulator is a validation tool that enables examination of the possible dynamic executions. In this way, certain events can be simulated, which can be saved as XTR files. This trace file contains all actions that the UPPAAL automata execute step by step. The XTR files are strongly linked to the model. Even small changes to the model can make an XTR file unreadable. The user must take this into account when making changes to the model. With the UPPAAL simulator it is possible

to create dynamic simulations and analyse them step by step. It is possible to simulate and analyse different schedules and non-deterministic behavior.

With the UPPAAL verifier, the user has another possibility to simulate the models of distributed algorithms. With it, queries can be checked and, if possible, counterexamples can be generated. These counterexamples can then be viewed in the simulator and also saved as XTR files. Generating traces from existing traces using the verifier is also possible For example, verifications like $E<>$ ... (Exists a trace, so that ...) can be written to get a trace from it, which can then be viewed and analysed in the simulator.This makes it possible to create specific traces with a desired state of the network. However, it should be noted that verification queries may change if the code is modified and translated again.

6.2 Usage of UPPAAL Traces for Visualisation

After examining the tools of the related work and testing some of our own examples, we have identified the following points as particularly valuable to visualise. **1)** The states of a node (active or inactive, as well as the local variables), **2)** Send and receive actions, as well as the content and assignment of a message in transit.

The individual automata are visualised with the UPPAAL simulator, but information about the communications between nodes is not sufficiently represented for our purposes. Likewise, the depth of visualisation in teaching can be confusing and counterproductive as it distracts from what is important. This is because the internal logic of each automata is shown, which can result in large and complex automata. For a better overview and a better understanding of the distributed algorithms, an alternative visualisation should be developed. This visualisation should focus on the important aspects mentioned in the paragraph above.

Using the UPPAAL models and XTR trace files created by the user, it is possible to implement an alternative visualisation. The visualisation is of course no longer dynamic, but may be more suitable for presentations or teaching.

6.2.1 Concept of the Visualisation

Fig.6.1 shows the visualisation concept using a small example. For this, we consider a distributed algorithm with 4 nodes and a message sent from *node3* to *node1*. The nodes are arranged in a circle and messages move between them. The status of a node or the status of its local variables can be viewed at any time (e.g. by clicking or hovering). In addition, a small lamp (green/red) indicates whether a node is active or inactive. Messages are also displayed, but only for the period in which they are in transit. Information about a message (sender, receiver and content, possibly also the time of sending and receiving) can also be visualised (e.g. by clicking or hovering). The information for nodes and messages can be displayed in text fields. A pause button and an acceleration or deceleration option can be offered as a control element for the user.

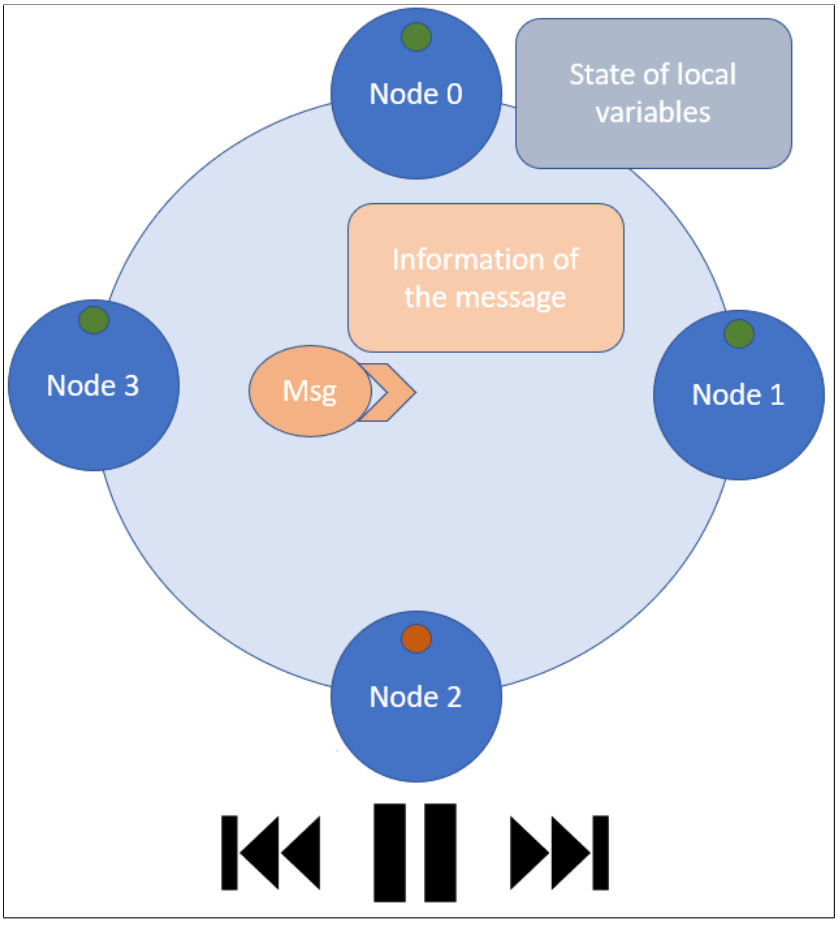


FIGURE 6.1: Concept of Visualisation

Events in UPPAAL traces always have a causal order without concurrency. For Example $event_1 \prec event_2 \prec event_3 \prec ..event_n$. This allows iteration through a list of events, changing local variables or sending messages. In our visualisation we consider two types of events. *Local Events* and *Communication Events*. Events for which no local variables change are dropped. Events occur at specific times and can be sequentially played, paused, and manually fast-forwarded or rewound.

Local Events

A local event in a node occurs whenever a local variable in a template changes. There are also changes to local variables related to communication, but these are treated separately as a *Communication Event*. Events in the UPPAAL model where no variable changes are of no interest for the visualisation, since they are abstracted from the internal logic of a node. A local evenet is a set of state assignments for all variables of the nodes of a distributed algorithm.

Communication Events

Communication events are the *sending* or *receiving* of a message at a node. Information about the sender, receiver and content is relevant here. A communication event is displayed as long as the message is in transit. Communication events have two timestamps. 1.) The timestamp of sending and 2.) The timestamp of receiving. Time here is logical time and corresponds to the order of events. A message is a 5-tuble consisting of (sender pid, receiver pid, content, time of sending, time of receiving).

Model

In a model we can realise a simplified representation of the UPPAAL model and its states in a simulation. Our model for visualising distributed algorithms consists of a list of nodes, a list of local events, a list of communication events (messages), global variables and constants, and a current timestamp. A node has a pid, a list of variables (state) and a name.

6.2.2 Realisation of the Visualisation

In order to visualise the simulation of a distributed algorithm, we need the XML of the UPPAAL model in addition to the XTR file (trace file). The XTR file contains the values of all local and global variables, line by line, as well as the IDs of the locations in which the templates are at a time. It is important to note that the order in which the values of the variables are listed in the XTR file corresponds to the order in the UPPAAL model, i.e. the line-by-line order. If the UPPAAL model changes, the XTR file may no longer match the model.

For our internal representation, we need a list of nodes and a list of events. In the following, we consider how these can be parsed from the XML and XTR files. The status of the network can be represented by the status of the nodes and the events that occur. This can then be visualised and examined step by step.

Parsing Nodes

In our visualisation, the following information from the XML model has to be parsed for nodes:

- The *name and PID* from the system declarations. It is determined from the system declarations, since it is defined here a) which instances are generated from templates and b) which PID is initialised with.
- What *local variables* a node has. These are read from the local declarations of the templates.

Structure of an XTR File and Parsing Events

In order to understand parsing, the format of the UPPAAL XTR file must be described first. The file contains all global and local variables line by line, as well as the locations of the templates. The order is the same as the order in which the templates are defined in the system declarations. An event is stored between two lines of with a dot. Since only the local and global variables are important for our events, but not information about the state of the locations in the templates, we must remove unnecessary information. This is trivial to implement. An event

always starts after two consecutive lines with a dot and ends after a line with only one dot. Information that is irrelevant to us lies below this, separated by a dot. Fig.6.2 shows this schematically. All data are stored as integers. For True and False, this must be taken care of in the visualisation.

```
1  <Init. Location of all Template>
2  .
3  .
4  <Global Variables>
5  <All Local Variables>
6  .
7  <Transitions (Irrelevant for us)>
8  .
9  .
10 <Global Variables>
11 <All Local Variables>
12 ...
```

FIGURE 6.2: UPPAAL XTR Format

Local Events

A local event consists of a new set of values for the template variables and a timestamp. They must be parsed from the XTR before playing a simulation. Together with the communication event, they form a list of events, which is sorted by timestamp. All events without global or local variable changes must be filtered out of the XTR file during parsing.

Communication Events

Parsing communication events is more complex than parsing local events. Communication events can be grouped again into two types of events, Send and Receive events. A send event can be detected when an entry is written into the global array (the buffer) with the messages. A receive event can be detected when an entry is taken from this array. In addition to the XTR file, the model is necessary to obtain the required information.

To get the information a) content of a message, b) sender PID and c) receiver PID, the global array of variables in Channels is used. The index i is used to calculate the sender and receiver PID for a net of n Nodes and e Edges. The

sender PID can be calculated by

$$\left\lfloor \frac{i}{n/e} \right\rfloor$$

The receiver PID is a more complex. It can be calculated by

$$p = i - (\frac{n}{e} \cdot pidSender), \text{where } pidReceiver = \begin{cases} p+1, & \text{if } p >= pidSender \\ p, & \text{if } p < pidSender \end{cases}$$

Assignment of a set of values for local variables of a template

Each event is a set of values for all global and local variables. For the assignment of a set of values for local variables to a node, the order of the templates in the system declarations is needed. In the XTR, the values are sorted in descending order, matching the order of the templates in the system declarations. In order to parse individual sets for nodes, the amount of local variables of the node must be calculated, which in connection with the position in the system declarations defines the corresponding set's start and end index.

6.3 Example with Huang's Termination Algorithm

Now we want to show how a distributed algorithm is visualised using our prototype (DiAlGo Viewer). For this we will consider the example from the previous chapters, an implementation of Huang's algorithm for the detection of termination. Fig.6.3 shows the DiAlGo Viewer. A simple trace was created in which 2 activation messages are sent. For this purpose, a trace was created using the UPPAAL Verifier under the query *E<> worker2.weight == 1024*. The communication graph for this looks as follows:

$$controlAgent \xrightarrow{2048} worker1, worker1 \xrightarrow{1024} worker2$$

In the UPPAAL simulator this is 130 steps, in our simplified alternative visualisation only 15. This is because all steps in which no local variables have changed

and no messages have been sent are removed. The UPPAAL simulator shows every step, i.e. every transition in the model. In our alternative visualisation, only transitions are shown that result in a update in the set of defined variables. For example, if a transition is taken in the UPPAAL model with a *guard* but without an *update*, this is not shown in our viewer.

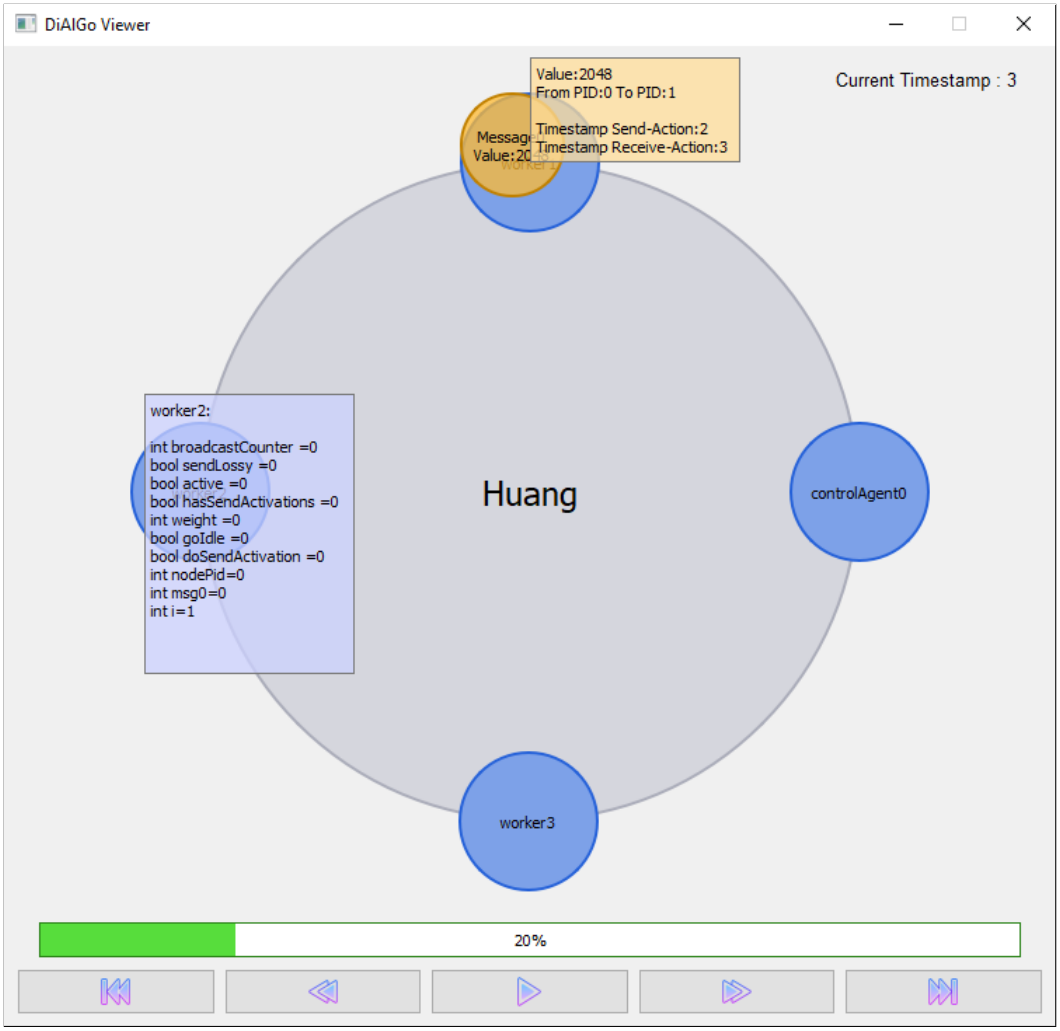


FIGURE 6.3: DiAlGo Viewer with Huang's Algorithm

The nodes of the network are shown in a circle with their name + PID (blue). Clicking on a node opens a text field (blue), which shows the state of all local variables of the node. Messages are represented by smaller circles (orange) that move from sender to receiver nodes. In our example, a message from the *controlAgent* has just arrived at *worker1*. If a message is clicked, a text field opens which shows the content, sender and receiver, as well as the start time and

the time of arrival. The user can control the simulation via the buttons at the bottom. The 5 buttons are for autoplay, step by step forward or back, reset and jump to the end. A progress bar shows how far the simulation has progressed in percent. In the upper right corner, the current time of the simulation is shown. It is a logical time, which means that it shows how many events have already occurred.

7 IMPLEMENTATION AND VALIDATION

In the chapter the implementations of the GO framework for the specifications of distributed algorithms, the translation of GO code to UPPAAL XML (DiAlGo Translator) and the visualisation of UPPAAL traces (DiAlGo Viewer) are presented and validated. For this purpose, we have implemented 3 distributed algorithms from the categories of mutual exclusion, leader election and termination were implemented, which we present and evaluate this work on. Huang's algorithm [16][1] was implemented for termination, the Bully algorithm [9] for leader election and Suzuki-Kasami [27] for mutual exclusion. Finally, we list all functionalities of the 3 tools (DiAlGo GO Framework, DiAlGo Translator, DiAlGo Viewer) in tables.

7.1 DiAlGo GO Framework

The specification of distributed algorithms is done in GO using the DiAlGo GO framework. This is a GO file which the user imports into their specification code. A distributed algorithm is thus specified by user code + dialgo.go. The GO Framework are functions that support the user in the implementation of their distributed algorithms. These are described in detail in *chapter 4.4*. The source code of the DiAlGo GO Framework (dialgo.go) is included in the appendix.

7.2 DiAlGo Translator

A console application was developed as part of this thesis for the transfer of GO code to UPPAAL models. The DiAlGo tool implements the logic to generate UPPAAL XML from GO files and to modify UPPAAL XML for specific purposes. The User Manual in the appendix explains in detail how to use the console application. DiAlGo offers the following features:

- Translating (*-translate*) Go code to UPPAAL models with
  - synchronous communication.
  - asynchronous communication.
  - asynchronous communication with a buffer.
  - non-deterministic behavior (choose a bool, choose an integer).
  - the GO Select.
  - logic like assignments, for loops and if/else blocks.
- Modifying (*-modify*) UPPAAL models by
  - adding a Template and annotations to simulate crashing nodes.

7.2.1 Architecture of DiAlGo Translator

DiAlGo Translator is a console application that runs with arguments from the terminal. The DiAlGo Translator is implemented in Java 17 as a Maven [13] project. Antlr4 [24] version 4.10.1 is used to parse GO files and the StringTemplate [25] template engine is used to create string templates. String templates are used, for example, to create *guards*, *updates*, etc. for the UPPAAL templates. The Maven build runs automated unit tests for individual components with JUnit4 [28] and builds a fat jar with all dependencies.

Internal Java Representation of Models

For the internal representation of the model, a net of automata based on the UPPAAL model consisting of templates, locations and transitions is used. So an object-oriented Java variant of the UPPAAL XML was implemented. The Java object for the net of automata can be translated directly into XML using its methods. An implemented Sax parser also made it possible to read UPPAAL XML and transfer it to the Java model. This model can then simply be annotated or expanded. This is used for modifying net of automata.

Parsing and Translating GO Files

The GO parser generated with Antlr is used to read a source file and create a parse tree. This is then run through and translated context-wise. Listeners or the visitor pattern were not used, as this option was quicker and easier to implement and test as part of a prototype.

Templates are created from the function contexts (which represent nodes) and the internal logic of a template is implemented by parsing the individual function contexts. In the function contexts, the statements (statements are, for example, variable assignments, function calls, selects, if/else blocks or for loops) are run through and translated one after the other. Contexts can also be nested in for loops, if/else blocks and selects. To do this, the nested list of statements is then run through and translated so that the logical order of the statements remains intact. The contexts are translated as described in *chapter 5*. The automaton template is composed of the individual building blocks and the net of automata is composed of the templates.

UML of the Architecture

Fig.7.1 shows the architecture of the DiAlGo Translator in UML notation. We abstract from attributes and methods for a better overview. The 4 packages of the project are shown. The package *res* contains all resources (constants, strings and string templates). The package *automata* contains the classes for the Java representation of a UPPAAL model as well as a *SAXHandler* for parsing UPPAAL XML. The package *goparser* contains all classes for parsing GO files. The *GoParserBase* was added manually and extends the *Parser* from Antlr. The automated generated *GoParser* extends this class. They were mainly generated automatically

from the GO grammar using Antlr4. The package *dialgo* contains the main (*App*) and the *Modifier* for modifying UPPAAL models. The modifier annotates UPPAAL models and adds templates. It is used to integrate the node crasher for switching nodes on and off. The package also contains the 3 classes (*GoHandler, GoContextHandler* and *AutomataNetworkHandler*) for translating. The GoHandler parses the contexts of a GO file with the help of the *GoContextHandler*. The *GoContextHandler* uses the *AutomataNetworkHandler* to create the individual automaton components of the UPPAAL model.

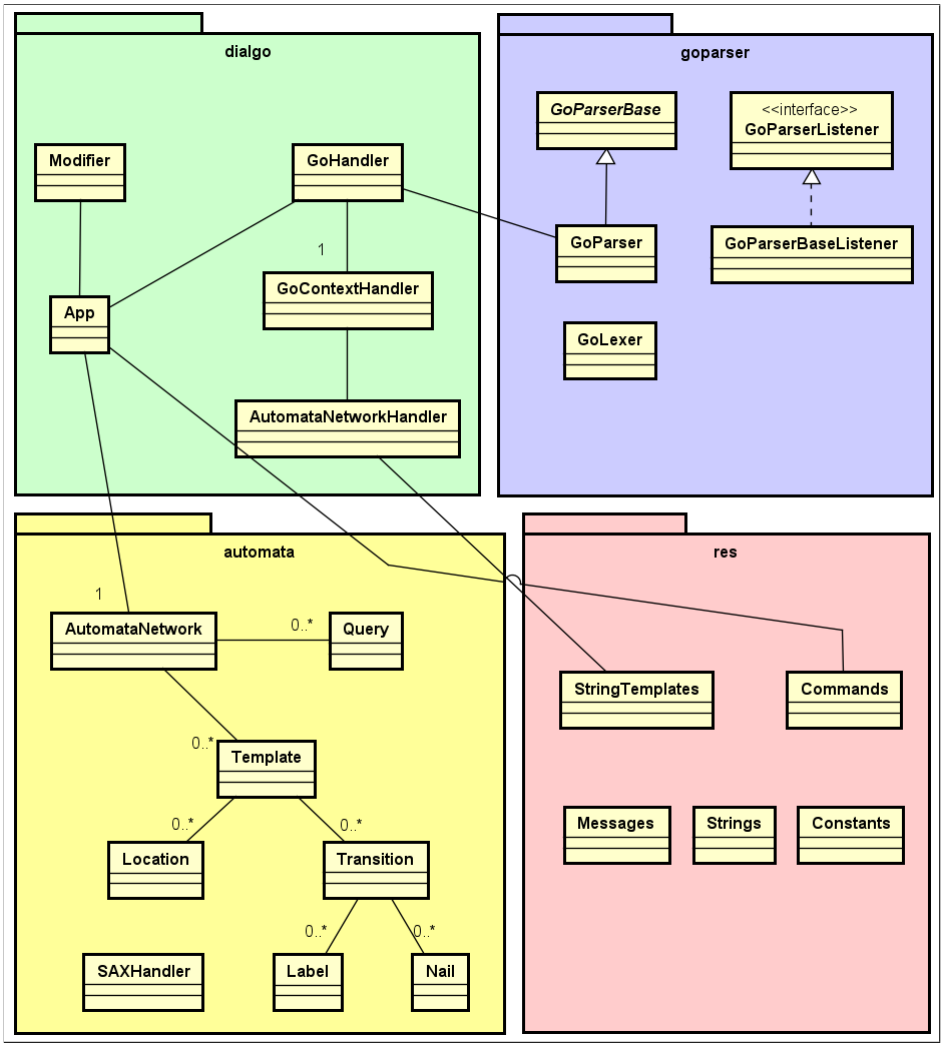


FIGURE 7.1: DiAlGo Translator UML

7.2.2 Simulation with UPPAAL

In order to support the manual simulation, the identifiers of the locations were chosen to be understandable. They have the structure <*kind*><*id*>. For example, the first for loop of a function is named *For1* in the model. The order of the event ids follows the logical order of execution according to the parse tree. However, the simulation environment of UPPAAL is only conditionally suitable for simulating manually distributed algorithms. In principle, it is possible to simulate the translated models, but they can be very confusing due to complex logic. Testing with the example for an implementation of the Huang algorithm (described in sections 4 and 5) showed that it makes sense to include the verifier. For example, we can write verifications like $E <> \phi$ (Exists a trace, so that ϕ satisfies) to get a trace from it, which can then be viewed and analysed in the simulator.

7.2.3 Verification with UPPAAL

A successful verification (with a good execution time) is strongly dependent on the complexity of the model, i.e. the complexity of the distributed algorithm. Tests with different implementations of distributed algorithms such as Huang's algorithm have shown that loops in particular have a strong impact on execution time. For reachability analyses, it has been shown that it makes sense to include time in order to force the automaton to take a step at some point. For this purpose, a watch automaton can be integrated manually. From a usability point of view, it has been shown that the speaking identifiers of the states make queries much easier. However, it must be noted that verifications have to be rewritten or adapted if the code is adapted. This is because the identifiers consist of event type + event number and the event number can change.

7.3 DiAlGo Viewer

The prototype DiAlGo Viewer was developed in Python with PyQt5 [21] as GUI framework for the visualisation of UPPAAL traces. Qt is a set of cross-platform C++ libraries that implement high-level APIs for desktop and mobile apps. PyQt5 is a Python binding for Qt version 5. Python was chosen because it is a simple and modern language and there are many libraries with which the prototype can be extended in any complex way. Especially with regard to the analysis of traffic in the network or the visualisation of such data. Such analyses have not been implemented, but could be included in future extensions.

UML of the Data Structure

Fig.7.2 shows a UML of the data structure of the viewer. A model is a network consisting of nodes, events and messages. A network holds a timestamp of the current time of the simulation, as well as global variables and constants in a string. A node also has local variables of a template in a string, as well as a PID and a name. An event has a timestamp and a set of new variable assignments for all nodes as a string. A message has the PID of the sender and receiver, as well as a content (string) and a send and receive time.

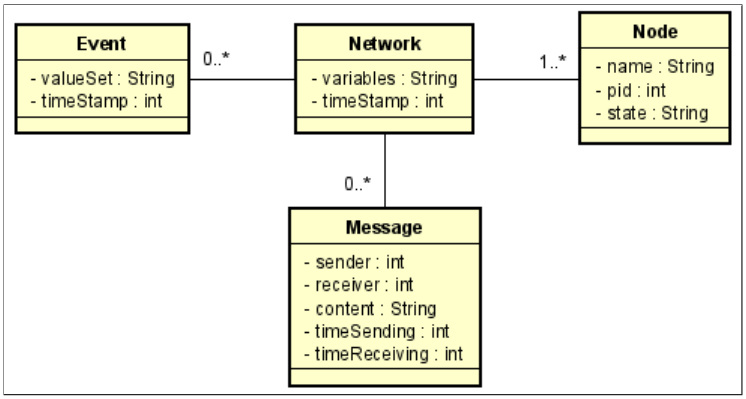


FIGURE 7.2: DiAlGo Viewer UML

7.4 Implemented Distributed Algorithms

With this work, 3 distributed algorithms from the categories of mutual exclusion, leader election and termination were implemented. These are important fields for the use of distributed algorithms. Huang's algorithm [16][1] was implemented for termination, the Bully algorithm [9] for leader election and Suzuki-Kasami [27] for mutual exclusion. With these examples, the functionalities of the tools were tested, analysed and evaluated. In this section, the implemented distributed algorithms are presented and evaluated in terms of specification in GO, translation to UPPAAL, simulation and verification and visualisation. The algorithms are included with their GO code and UPPAAL models in the appendix.

7.4.1 Huang's Algorithm

Huang's algorithm [16][1] for termination detection is used to detect that a network of nodes that work distributively and call each other has terminated. For this we consider two types of nodes. A control node (control agent) and a worker node (worker). The control agent activates a worker. This starts work and activates other workers. At some point, all workers have completed their work and no longer call each other again. The controlling agent wants to determine this status. The algorithm was described in detail as an continuous example in the *chapters 4,5* and *6*. It was chosen as an continuous example because it only requires one channel or one channel array of one type and is therefore comparatively manageable. The complete example (code and model) is attached in the appendix.

Specification in GO

The GO Framework is suitable for the specification. The functionalities *Send, Receive, Random generation of an Integer* and *Random generation of a Boolean* were used. We specified our algorithm with 4 nodes (3 worker and 1 control agent). An asynchronous integer channel was used to send the weight. The GO code has console outputs and can be tested with any number of worker nodes. Due to the built-in randomness, each run is non-deterministic.

Translation to UPPAAL

All concepts were translated without errors. The automatically generated UPPAAL model is large but still comprehensible, thanks to its descriptive naming.

Simulation and Verification with UPPAAL

The UPPAAL simulator and verifier could be used for simulation and verification. The verifier was used in particular to analyse certain traces. For example, *E<> worker2.weight == 1024* to generate a trace in which *worker2.weight = 1024*. The simulations were easy to follow and create after a short familiarisation with the model.

Visualisation

The DiAlGo Viewer shows all essential aspects of the algorithm. Events (state changes) can be analysed step by step and all information about node states and messages can be taken from the GUI.

7.4.2 Bully Algorithm

The bully algorithm [9] is a recursive distributed election algorithm. The task is for a network to independently detect the failure of the leader and resolve it through a new election. We take a look at the following implementation. The basic idea is that the node with the highest PID always has the privilege of becoming leader. Initially, the process with the highest PID is the leader. The worker nodes send heartbeats to their leader to check if the leader is still active. In our implementation, we use a timeout of 2 seconds, which cannot be represented in the UPPAAL model because we have not taken time into account. If a heartbeat is not answered (because the leader crashed or terminated), the node starts an election. The node sends an election request to all nodes with a higher PID. If it does not receive an response, it becomes the new leader itself. If a node with a higher PID receives an election request, the node responds to the request and itself starts an election request to all nodes with a higher PID. At the end of this process, the node with the highest PID becomes the new leader and informs all other nodes of this via a broadcast. When implementing

the algorithm in GO, it must be ensured that all channels have a buffer with a suitable size, as otherwise the system will get stuck due to blocking channels if a node fails. Our implementation is set on 10 iterations. After 10 iterations, a node terminates. The complete example (code and model) is attached in the appendix.

Specification in GO

The algorithm was developed in GO using the DiAlGo GO framework.The functionalities of *Send* and *Receive* were used. We specified our algorithm for testing with 3, 4 and 5 nodes. For communication between nodes 3 channels are used. One for the heartbeats with the leader (bool type), one for the election requests (int type) and one for the election response (int type). These channels have a buffer of size 10. The GO code has console outputs and can be tested with any number nodes.

Translation to UPPAAL

The translated model was extended with the DiAlGo Translator modifier to include a node crasher to simulate failing nodes. It is now possible to analyse various traces and create simulations in which individual nodes crash and new leaders are elected. The UPPAAL model cannot map the time for timeouts, which is why it can happen that a node appoints itself leader directly after sending the heartbeat, as it does not wait correctly (2 sec.) for responses. Time could be integrated in future extensions to more accurately model the behaviour of the specified program code.

Simulation and Verification with UPPAAL

Because of its complexity, the UPPAAL model is a bit confusing, which is why the Verifier was mainly used for analyses. It was used to write queries such as *E<> node1.isLeader && nodeCrash[2] == 1* to analyse a simulation of a trace in which node2 fails and node1 becomes the new leader. During verification, a problem was found in the design that was not noticed during implementation. If a leader node fails and later rejoins, becoming a leader again, a heartbeat channel may block until an old message is read as a heartbeat. The fact that the channels need a buffer was revealed by verification and simulation in which the machine got stuck with a smaller or without a buffer.

Visualisation

The DiAlGo Viewer shows all essential aspects of the algorithm. Events (state changes) can be analysed step by step and all information about node states can be taken from the GUI. Whereby messages cannot be displayed because the prototype of the viewer does not support channels with buffers. As a possible improvement, it has been noted that it can be useful to highlight certain local variables. For example, it would be interesting to highlight the variable *isLeader* (a flag indicating whether a node is a leader), as it contains the elementary information of the algorithm.

7.4.3 Suzuki-Kasami Algorithm

The Suzuki-Kasami algorithm [27] is a token based distributed algorithm for mutual exclusion. The idea of the algorithm is that nodes want to enter a critical section. To do this, they need a token. There is only one token in the system, which ensures that only one node is allowed to enter the critical section at a time. To request a token, a node sends a broadcast to others. The nodes manage a queue in which they store all requests. The node with the token sends its token with the queue to the first node in the queue when it leaves the critical section.

In our implementation, we use 2 channels. One to send a request for the token (asynchronous) and one to send the token with a queue of waiting nodes (synchronous). Each node that wants to enter the critical section (CS) announces its need by a broadcast. In the original description of the algorithm, the nodes hold a queue in which they manage the incoming requests. In our implementation, this queue is realised with an Integer because no channels of array types were implemented in the prototype. In order to get the last value (digit) from our simulated queue (Integer) we do *modulo 10*. For example, if we have a queue of {*1,2,4*}, we get the last digit with *421 % 10 = 1*. Then, we do *421 / 10 = 42* to remove the *1*. The new queue is {*2,4*} To add a value to the queue we have to append as many zeros to the value as our queue is long. We then add this value to our queue. For example, if we add the value 3 to our queue of {*2,4*}, the new queue is *42 + 300 = 342*. The new queue now has the form *Queue* = {*2,4,3*}. This queue is also sent with a token as a reply message. The reply is

sent via a synchronised channel. The request broadcast is sent asynchronously. Our implementation is set on 5 iterations were all nodes want to enter the cs ones. After 5 iterations, a node terminates. If all nodes were in the cs, the last node can no longer release its token. This is why the implementation always ends up in a deadlock. The complete example (code and model) is attached in the appendix.

Specification in GO

The algorithm was developed in GO using the DiAlGo GO framework.The functionalities of *send, receive* and *broadcast* were used. We tested our algorithm with 4 and 3 nodes. The request for the token is sent asynchronously via a boolean channel. The response with token and queue is sent synchronously. The first implementation had a bug, which could be found through verification. It could happen that a request was ignored if the schedule was unlucky. A request could be lost if it was sent while the token with the queue was in transit. In a second version this bug was fixed. The GO code has console outputs and can be tested with any number nodes.

Translation to UPPAAL

All concepts were translated without errors. The automatically generated UPPAAL model is large but still comprehensible, thanks to its descriptive naming.

Simulation and Verification with UPPAAL

Various traces were analysed. In doing so, a variable *wasInCs* was used in particular, which is a flag that indicates whether a node was already in the critical section. It was used to formulate queries such as *E<> node0.wasInCs && node1.wasInCs* to get a trace were node0 and node2 were in the critical section (cs). The verification *E<> node2.wasInCs* was tested to compare the duration. For 3 nodes the verification took < 3 seconds, for 4 nodes already > 300 seconds. The error of a request being lost was found through verification and then corrected. The problem was that nodes that did not have the token ignored a request. It could happen that a request was lost while the response with token and queue was in transit. With the verification *E<> (!node0.wasInCs || !node1.wasInCs || !node2.wasInCs) && deadlock* it can be verified that in each run all nodes can reach the critical section because this query is not satisfied.

Visualisation

A trace was chosen in which *E<> node1.wasInCs && node2.wasInCs && node0.wasInC* holds. In this simulation, all nodes have entered and left the critical section at some point. The DiAlGo Viewer shows all essential aspects of the algorithm. Like in the Bully algorithm example, it would be useful to highlight certain local variables such as *hasToken* or *wasInCs*.

7.5 Validation

In this section, all functionalities are presented and the robustness and performance are evaluated. The functionalities of the tools and the GO Framework are presented in tabular form.

7.5.1 Robustness

The two tools and the framework were manually tested by 3 large algorithms(Huang, Bully Election, Suzuki-Kasami). Different types of communications (synchronous, asynchronous) via int and boolean channels have been used. All functionalities (loops, if statements, GO select, etc.) were used and successfully tested. In addition to the 3 large examples, numerous smaller algorithms were tested for all functionalities. The DiAlGo Translator was also tested with automated unit tests. This showed that the system works reliably as long as the input is a syntactic correct GO file and uses the specified subset.

7.5.2 Performance

The performance of the GO Framework and the DiAlGo Viewer are unproblematic. The biggest difficulty is a performant UPPAAL model in which verification can be done in an acceptable time. For large models of distributed algorithms or models with many nodes, the performance drops.

To evaluate the performance for an increasing number of nodes, the following test was performed. Huang's algorithm was tested with the query $E<>$ *worker2.weight* $==$ *1024*. Table 7.1 shows the evaluation results. The test showed that the performance gets significantly worse as the number of nodes increases.

Nodes	Time	Memory
3	0.047 sec	13.892 KB
4	2.219 sec.	55.220 KB
5	68.844sec.	1.213.596 KB

TABLE 7.1: Performance with Increasing Number of Nodes

7.5.3 Limitations

This section lists the limitations of DiAlGo Translator and DiAlGo Viewer.

DiAlGo Translator

The DiAlGo Translator cannot create UPPAAL models with time. Especially in the example with the Bully Election Algorithm it was noticed that this feature can be interesting for future work. Non-FiFo channels were only conceptually presented but not implemented in the prototype.

DiAlGo Viewer

The DiAlGo Viewer cannot display arrays in templates. In addition, messages from channels with a buffer cannot be parsed. The display that a node is active or inactive has not been implemented.

7.5.4 Functionalities

Table 7.2 shows the supported functionalities of the DiAlGo Translator and the DiAlGo GO Framework. Table 7.3 shows the supported functionalities of the DiAlGo Viewer.

Functionality	DiAlGo GO Framework	DiAlGo Translator
Typical control elements such as loops, if/else blocks and assignments.	YES	YES
GO Select for Channels	YES	YES
Sending a message from node A to node B (synchronous, asynchronous and with buffer).	YES	YES
Receiving a message from a node via an index from an array of channels (synchronous, asynchronous and with buffer.	YES	YES
Generating a random boolean.	YES	YES
Generating a random Integer over a given range.	YES	YES
Broadcasting a message (synchronous, asynchronous and with buffer).	YES	YES
Sending lossy (simulating a message got lost in transit) a message from node A to node B (synchronous, asynchronous and with buffer).	YES	YES
Simulating crashing nodes.	NO	YES
Non-FiFo Channels.	NO	NO*

* Prof-Of-Conzept, but not implemented in prototype.

TABLE 7.2: Validation of Functionalities in DiAlGo Translator and the DiAlGo GO Framework

Functionality	DialGo Viewer
Show nodes and their state (local variables).	YES
Play events step by step.	YES
User controls for the simulation and display of the current time of the simulation.	YES
Display of messages, as well as display of information.	YES

TABLE 7.3: Validation of Functionalities in the DiAlGo Viewer

8

CONCLUSION AND OUTLOOK

In this thesis, it was shown 1.) how distributed algorithms are specified using a developed GO framework, 2.) how these can be automatically translated into a UPPAAL model for simulations and analyses, and 3.) an alternative visualisation of UPPAAL traces. As part of this thesis, a GO framework (DiAlGo GO Framework) was developed for the specification of distributed algorithms in GO, as well as a console application in Java for automated translation and modification (DiAlGo Translator), and a viewer in Python (DiAlGo Viewer) for visualising UPPAAL traces. Various aspects of communications via channels were investigated, such as synchronous and asynchronous channels, channels with buffers and non-FiFo channels. Also, possible sources of errors were examined and presented, such as crashing nodes and lossy channels were messages can get lost in transit not reaching their destination. A realisation of GO concepts in UPPAAL such as *for* loops, *if* statements, *select* and others was shown. The translation of these GO concepts can be used independently of the context of distributed algorithms, as it is a generic approach. Furthermore, it was researched how a distributed algorithm can be visually represented and how an alternative visualisation of UPPAAL traces can be realised. Our approach was focused on the representation of traffic (message passing) in a network. In doing so, the internal logic of a node was abstracted. With 3 distributed algorithms from the areas of mutual exclusion, termination and leader election, the concepts were presented and the developed tools evaluated.

In the context of this thesis, comparable work in the field of simulation and visualisation of distributed algorithms was researched and the project of this thesis was compared with related tools. It was shown that the approach of using UPPAAL as a model checker for simulations and analyses is new and offers many excellent possibilities in terms of analysis, simulation and verification. The use of GO for this purpose is also new. It was shown that GO is highly suitable for the specification of distributed algorithms due to its native features. The developed prototypes and examples are available on GitHub [14].

Out of this work, there are plenty of opportunities for extensions in our Tools. Some possible extensions are presented below.

The DiAlGo Go Framework.

The GO Framework can be extended with further functions for the implementation of distributed algorithms in GO. For example, support for non-FiFo channels could be developed, non-complete networks could be realised (i.e. individual channels and no channel arrays), time could be embedded and many more to assist modeling distributed algorithms.

The DiAlGo Translator.

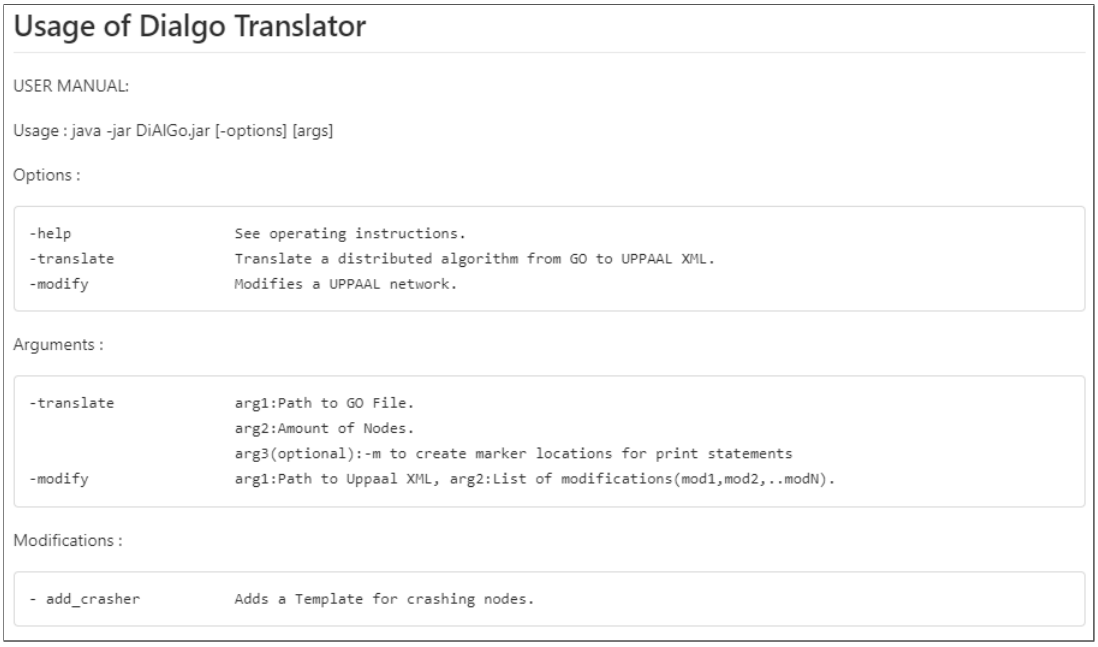
The DiAlGo Translator can also be extended with further functions. For example, a modifier could be implemented which extends and annotates a UPPAAL model with time, so that transitions have to be taken after a certain time. A automated added watch-automaton can be used for this purpose. Also a modifier for changing buffer channels to non-FiFo channels can be implemented. The concepts for this have already been presented in *chapter 5.3.5*.

The DiAlGo Viewer.

The DiAlGo Viewer is a prototype which can be completed and extended in many aspects. Buffer channels cannot be displayed yet. As an extension of the visualisation itself, it is conceivable to introduce highlighting for certain user-defined variables. In addition, the implementation in Python offers numerous possibilities for traffic analyses. For example, graphs could be plotted, such as where communication in a network is highest.

A

APPENDIX



Usage of Dialgo Translator

USER MANUAL:

Usage : java -jar DiAlGo.jar [-options] [args]

Options :

```
-help                 See operating instructions.
-translate            Translate a distributed algorithm from GO to UPPAAL XML.
-modify               Modifies a UPPAAL network.
```

Arguments :

```
-translate            arg1:Path to GO File.
                      arg2:Amount of Nodes.
                      arg3(optional):-m to create marker locations for print statements
-modify               arg1:Path to Uppaal XML, arg2:List of modifications(mod1,mod2,..modN).
```

Modifications :

```
- add_crasher         Adds a Template for crashing nodes.
```

FIGURE A.1: DiAlGo Translator Manual

DiAlGo GO Framework

```
package main

/*
* Framework DiAlGo (Distributed Algorithms with Go).
* For writing and simulating distributed algorithms.
* Author : Torben Friedrich Goerner
 */

import (
        "math/rand"
        "time"
)

const NODES int = 4 // amount of nodes
const EDGES int = 12 // amoount of edges (n*(n-1)/2)*2 --> n*(n-1)

const (
        TRUE  = 1
        FALSE = 0
)

// init function
func Init() {
        rand.Seed(time.Now().UnixNano())
}

// returns a random bool
func RandomBool() bool {
        ran := rand.Intn(TRUE + 1)
        if ran == TRUE {
                return true
        }
        return false
```

```
}

// returns a random int in range min - max
// (range limits are inclusive)
func RandomInt(min int, max int) (i int) {
        i = rand.Intn(max-min+1) + min
        return
}

// returns a random int in range min - max
// (range limits are inclusive) without a value 'myPid'
func RandomOtherPid(min int, max int, myPid int) (i int) {
        i = rand.Intn(max-min+1) + min
        if i == myPid {
                i = min
        }
        return
}

// gets the index of a recieving channel were a listens on b
func GetReceiveIndex(pidA int, pidB int) (index int) {
        var in int = 0
        if pidA > pidB {
                in = pidA - 1
        } else {
                in = pidA
        }
        index = ((NODES - 1) * pidB) + in
        return
}

// sends a msg from pid A to pid B via a channel
func Send(pidA int, pidB int, chans [EDGES]chan interface{}
            , msg interface{}) {
        var in int = 0
```

```
        if pidB > pidA {
                in = pidB - 1
        } else {
                in = pidB
        }
        var index int = ((NODES - 1) * pidA) + in
        chans[index] <- msg
}

// sends lossy a msg from pid A to pid B via a channel
func SendLossy(pidA int, pidB int, chans [EDGES]chan interface{}
                , msg interface{}) {
        var send bool = RandomBool()
        if send { Send(pidA, pidB, chans, msg) }
}

// broadcasts a msg from pid to all other nodes
/func Broadcast(pid int, chans [EDGES]chan interface{}
                , msg interface{}) {
        for i := 0; i < NODES; i++ {
                if pid != i {
                        Send(pid, i, chans, msg)
                }
        }
}

// broadcasts a msg lossy from pid to all other nodes
func BroadcastLossy(pid int, chans [EDGES]chan interface{}
                    , msg interface{}) {
        for i := 0; i < NODES; i++ {
                if pid != i {
                        SendLossy(pid, i, chans, msg)
                }
        }
}
```

Huang's Algorithm for Termination

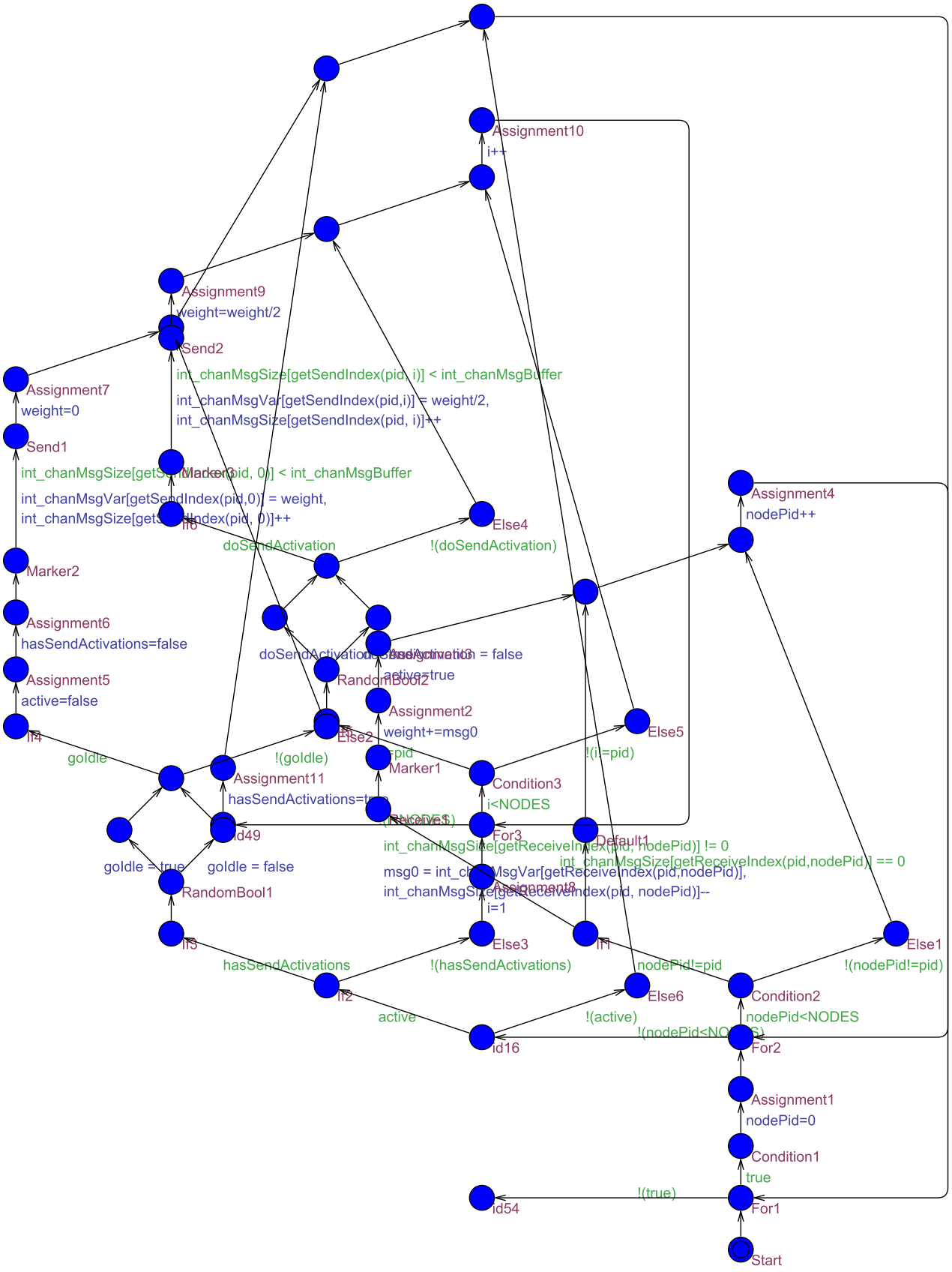
First the GO specification is shown and then the UPPAAL model. The Worker Template is shown first and then the ControlAgent Template.

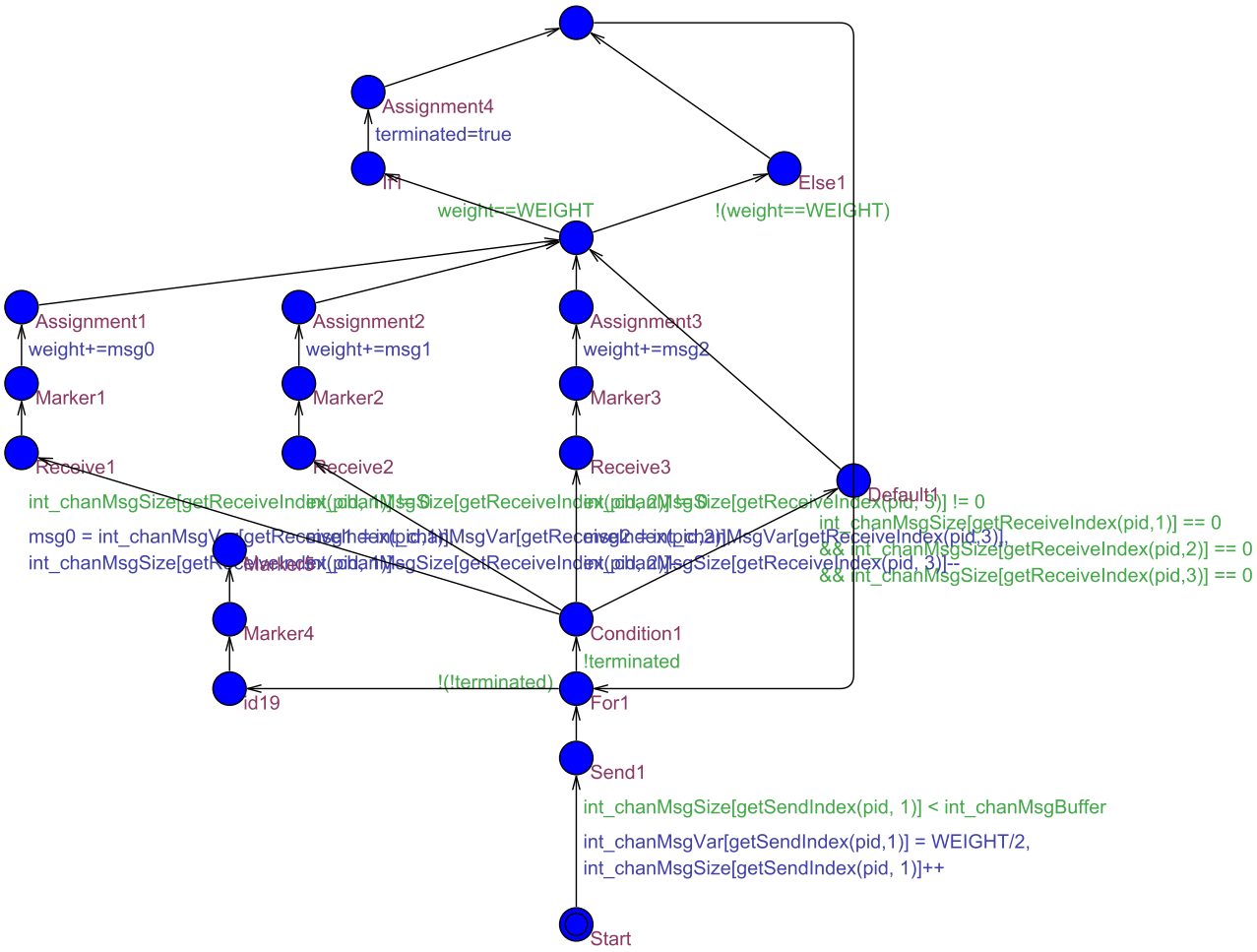
```
package main

/*
Example for Huang's algorithm.
*/

import (
  "fmt"
  "os"
  "sync"
)

// Weight = 2^12
const WEIGHT int = 4096
// Array of chan for sending msg with weight
var int_chanMsg [EDGES]chan interface{}

func main() {

  Init() // Call DiAlGo Init

  // Init chans
  for i := range int_chanMsg {
    int_chanMsg[i] = make(chan interface{}, 1)
  }

  // Waitgroup of all nodes
  var wg sync.WaitGroup
  wg.Add(NODES)
```

```
  go func() {
    controlAgent(0)
    wg.Done()
  }()
  go func() {
    worker(1)
    wg.Done()
  }()
  go func() {
    worker(2)
    wg.Done()
  }()
  go func() {
    worker(3)
    wg.Done()
  }()

  wg.Wait()
}

// Worker node in a distributed system
func worker(pid int) {
  var active bool = false
  var hasSendActivations bool = false
  var weight int = 0
  var goIdle bool = false
  var doSendActivation bool = false

  for true {
    // Listen for activations
    for nodePid := 0; nodePid < NODES; nodePid++ {
      if nodePid != pid {
        select {
        case msg0 := <-int_chanMsg[GetReceiveIndex(pid,nodePid)]:
          fmt.Println("PID", pid, "received", msg0)
```

```
            weight += msg0.(int)
            active = true
          default:
          }
        }
      }

      if active {
        if hasSendActivations {
          goIdle = RandomBool()
          if goIdle { // Go non deterministic to idle
            active = false
            hasSendActivations = false
            fmt.Println("PID", pid, "sending", weight, "to CA")
            Send(pid, 0, int_chanMsg, weight)
            weight = 0
          }
        } else {
           // Send non deterministic activations to other nodes
          for i := 1; i < NODES; i++ {
            if i != pid {
              doSendActivation = RandomBool()
              if doSendActivation {
                fmt.Println("PID",pid,"sending",weight/2,"to PID",i)
                Send(pid, i, int_chanMsg, weight/2)
                weight = weight / 2
              }
            }
          }
          hasSendActivations = true
        }
      }
    }
}
```

```
// Control agent for worker nodes
func controlAgent(pid int) {
  var terminated bool = false
  var weight int = WEIGHT / 2

  Send(pid, 1, int_chanMsg, WEIGHT/2)

  for !terminated {
    // Listen for getting back weight
    select {
    case msg0 := <-int_chanMsg[GetReceiveIndex(pid, 1)]:
      fmt.Println("PID", pid, "received", msg0
            , "from PID 1. New weight=", weight+msg0.(int))
      weight += msg0.(int)
    case msg1 := <-int_chanMsg[GetReceiveIndex(pid, 2)]:
      fmt.Println("PID", pid, "received", msg1
            , "from PID 2. New weight=", weight+msg1.(int))
      weight += msg1.(int)
    case msg2 := <-int_chanMsg[GetReceiveIndex(pid, 3)]:
      fmt.Println("PID", pid, "received", msg2
            , "from PID 3. New weight=", weight+msg2.(int))
      weight += msg2.(int)
    default:
    }

    if weight == WEIGHT {
      terminated = true
    }
  }

  fmt.Println("Control Agent with PID", pid
        , ": Seems like everyone is done!")
  fmt.Println("Control Agent with PID", pid, "terminated!")
  os.Exit(3)
}
```


Assignment10
i++
Assignment9
weight=weight/2
Send2
int_chanMsgSize[getSendIndex(pid, i)] < int_chanMsgBuffer
int_chanMsgVar[getSendIndex(pid,i)] = weight/2,
int_chanMsgSize[getSendIndex(pid, i)]++
Assignment7
weight=0
Send1
int_chanMsgSize[getSendIndex(pid, 0)] < int_chanMsgBuffer
int_chanMsgVar[getSendIndex(pid,0)] = weight,
int_chanMsgSize[getSendIndex(pid, 0)]++
If6
doSendActivation
Else4
!(doSendActivation)
Assignment4
nodePid++
Marker2
Assignment6
hasSendActivations=false
Assignment5
active=false
If4
goIdle
!(goIdle)
RandomBool2
active=true
Assignment2
weight+=msg0
Else2
Marker1
Else5
!(i!=pid)
Assignment11
Condition3
i<NODES
For3
int_chanMsgSize[getReceiveIndex(pid, nodePid)] != 0
Default1
int_chanMsgSize[getReceiveIndex(pid,nodePid)] == 0
msg0 = int_chanMsgVar[getReceiveIndex(pid,nodePid)],
int_chanMsgSize[getReceiveIndex(pid, nodePid)]--
Assignment8
i=1
Id49
goIdle = true
goIdle = false
RandomBool1
If5
hasSendActivations
Else3
!(hasSendActivations)
If1
nodePid!=pid
Else1
!(nodePid!=pid)
If2
active
Else6
!(active)
Condition2
nodePid<NODES
Id16
For2
Assignment1
nodePid=0
Condition1
true
!(true)
Id54
For1
Start


Assignment4
terminated=true
If1
weight==WEIGHT
Else1
!(weight==WEIGHT)
Assignment1
weight+=msg0
Assignment2
weight+=msg1
Assignment3
weight+=msg2
Marker1
Marker2
Marker3
Receive1
Receive2
Receive3
Default1
int_chanMsgSize[getReceiveIndex(pid,1)] == 0
&& int_chanMsgSize[getReceiveIndex(pid,2)] == 0
&& int_chanMsgSize[getReceiveIndex(pid,3)] == 0
Marker4
Condition1
!terminated
!(!terminated)
Id19
For1
Send1
int_chanMsgSize[getSendIndex(pid, 1)] < int_chanMsgBuffer
int_chanMsgVar[getSendIndex(pid,1)] = WEIGHT/2,
int_chanMsgSize[getSendIndex(pid, 1)]++
Start

Suzuki-Kasami Algorithm

First the GO specification is shown and then the UPPAAL model.

```
package main

/*
Example for Suzuki-Kasami token algorithm for mutal exclusion.
*/

import (
  "fmt"
  "sync"
)

// Array of chan for requesting cs
var bool_chanRequest [EDGES]chan interface{}
// Array of chan for giving token to cs and sending the queue
var int_chanReply [EDGES]chan interface{}

func main() {

  Init()

  // Init chans
  for i := range bool_chanRequest {
    bool_chanRequest[i] = make(chan interface{}, 1)
  }

  for i := range int_chanReply {
    int_chanReply[i] = make(chan interface{})
  }

  // Waitgroup of all nodes
  var wg sync.WaitGroup
```

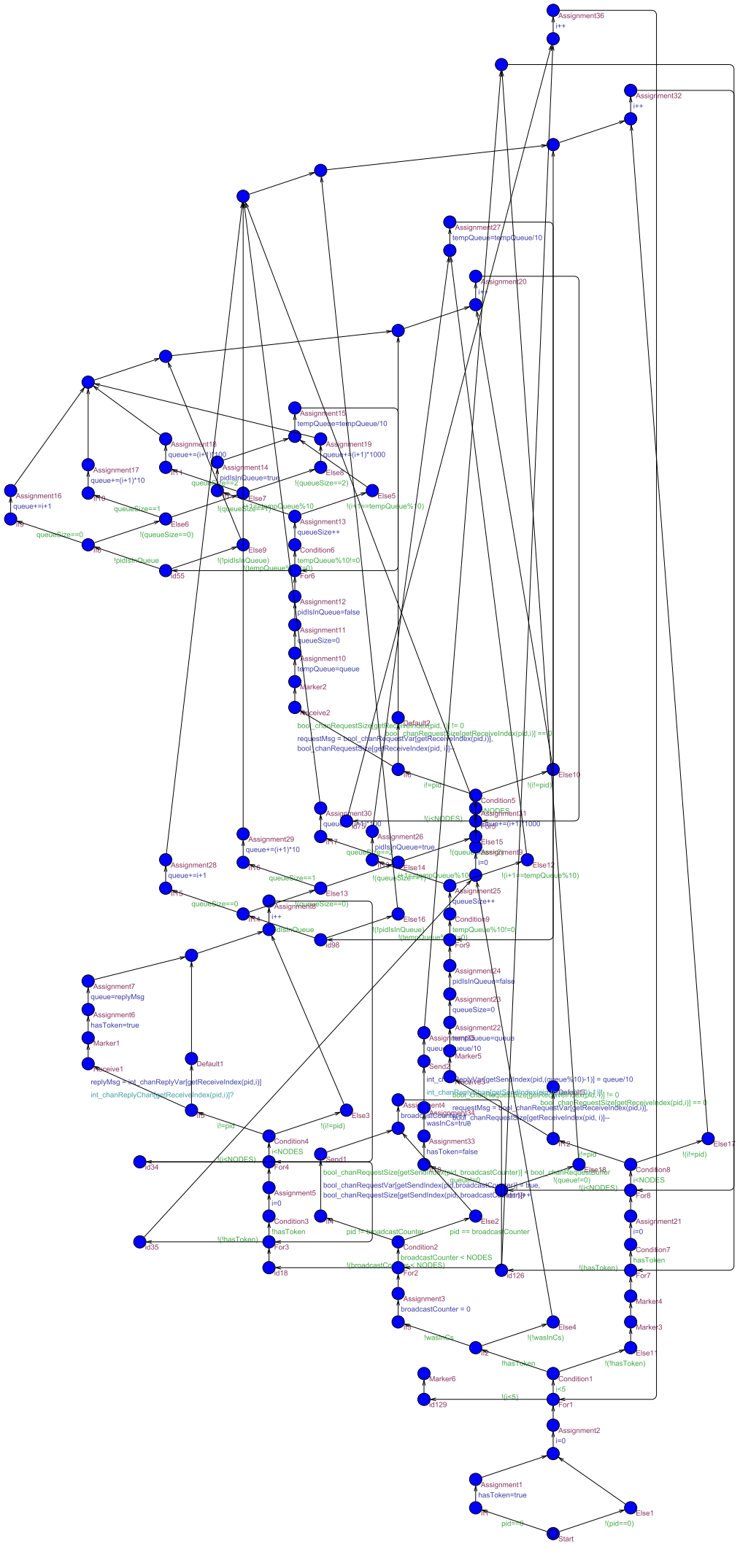
```
  wg.Add(NODES)

  go func() {
    node(0)
    wg.Done()
  }()
  go func() {
    node(1)
    wg.Done()
  }()
  go func() {
    node(2)
    wg.Done()
  }()

  wg.Wait()
}

// Node (all nodes want to enter cs ones)
func node(pid int) {
  var hasToken bool = false
  var wasInCs bool = false
  var queue int = 0
  var tempQueue int = 0
  var queueSize int = 0
  var pidIsInQueue bool = false

  // Init Token on PID 0
  if pid == 0 {
    hasToken = true
  }

  // Work 5 Steps
  for i := 0; i < 5; i++ {
    if !hasToken {
```

```
// Send Broadcast to request CS
if !wasInCs {
  Broadcast(pid, bool_chanRequest, true)

  // Wait for Token
  for !hasToken {
    for i := 0; i < NODES; i++ {
      if i != pid {
        select {
        case replyMsg :=
              <-int_chanReply[GetReceiveIndex(pid, i)]:
          fmt.Println("PID", pid, "received Token with Queue"
              , replyMsg, "from PID", i)
          hasToken = true
          queue = replyMsg.(int)
        default:
        }
      }
    }
  }
}

for i := 0; i < NODES; i++ {
  if i != pid {
    select {
    case requestMsg :=
              <-bool_chanRequest[GetReceiveIndex(pid, i)]:
      fmt.Println("PID", pid, "received Request", requestMsg
          , "from PID", i)
      tempQueue = queue
      queueSize = 0
      pidIsInQueue = false
      for tempQueue%10 != 0 {
        queueSize++
        if i+1 == tempQueue%10 {
```

```
                pidIsInQueue = true
              }
              tempQueue = tempQueue / 10
            }
            if !pidIsInQueue {
              if queueSize == 0 {
                queue += i + 1
              } else if queueSize == 1 {
                queue += (i + 1) * 10
              } else if queueSize == 2 {
                queue += (i + 1) * 100
              } else {
                queue += (i + 1) * 1000
              }
            }
          default:
          }
        }
      }

    } else {
      // Entering CS
      fmt.Println("PID", pid, "entering CS.")
      // Leaving CS
      fmt.Println("PID", pid, "leaving CS.")

      for hasToken {
        // Receiving Broadcast
        for i := 0; i < NODES; i++ {
          if i != pid {
            select {
            case requestMsg :=
                <-bool_chanRequest[GetReceiveIndex(pid, i)]:
              fmt.Println("PID", pid, "received Request"
                , requestMsg, "from PID", i)
```

```
            // Store in Queue
            tempQueue = queue
            queueSize = 0
            pidIsInQueue = false
            for tempQueue%10 != 0 {
              queueSize++
              if i+1 == tempQueue%10 {
                pidIsInQueue = true
              }
              tempQueue = tempQueue / 10
            }
            if !pidIsInQueue {
              if queueSize == 0 {
                queue += i + 1
              } else if queueSize == 1 {
                queue += (i + 1) * 10
              } else if queueSize == 2 {
                queue += (i + 1) * 100
              } else {
                queue += (i + 1) * 1000
              }
            }
          default:
          }
        }
      }
      // Send Token to First in Queue
      if queue != 0 {
        // Release Token
        hasToken = false
        wasInCs = true
        Send(pid, (queue%10)-1, int_chanReply, queue/10)
        queue = queue / 10
      }
    }
```

```
    }
  }
  fmt.Println("PID", pid, "terminated! I was in CS =", wasInCs)
}
```


Assignment36
i++
Assignment32
i++
Assignment27
tempQueue=tempQueue/10
Assignment20
i++
Assignment15
tempQueue=tempQueue/10
Assignment18
queue+=(i+1)*100
Assignment19
queue+=(i+1)*1000
Assignment17
queue+=(i+1)*10
Assignment14
pidIsInQueue=true
Else8
!(queueSize==2)
Assignment16
queue+=i+1
In1
In9
queueSize==2
Else7
!(queueSize==1)
Else5
!(i+1==tempQueue%10)
queueSize==1
Else6
!(queueSize==0)
Assignment13
queueSize++
In8
queueSize==0
Else9
!(pidIsInQueue)
Condition6
tempQueue%10!=0
pidIsInQueue
!(tempQueue%10!=0)
id55
For6
Assignment12
pidIsInQueue=false
Assignment11
queueSize=0
Assignment10
tempQueue=queue
Marker2
Receive2
bool_chanRequestSize[getReceiveIndex(pid, i)] != 0
requestMsg = bool_chanRequestVar[getReceiveIndex(pid,i)],
bool_chanRequestSize[getReceiveIndex(pid, i)]--
Default2
bool_chanRequestSize[getReceiveIndex(pid,i)] == 0
In6
i!=pid
Else10
!(i!=pid)
Condition5
i<NODES
!(i<NODES)
Assignment21
i++
Assignment30
queue=tempQueue
tempQueue=tempQueue/10
Assignment29
queue+=(i+1)*10
Assignment26
pidIsInQueue=true
Else15
!(queueSize==2)
Assignment9
i=0
In7
queueSize==2
Else14
!(queueSize==1)
Else12
!(i+1==tempQueue%10)
Assignment28
queue+=i+1
In5
queueSize==1
Else13
queueSize==0
Assignment25
queueSize++
Assignment8
i++
!(queueSize==0)
In4
pidIsInQueue
Else16
!(!pidIsInQueue)
Condition9
tempQueue%10!=0
!(tempQueue%10!=0)
id98
For9
Assignment24
pidIsInQueue=false
Assignment23
queueSize=0
Assignment22
tempQueue=queue
Assignment7
queue=replyMsg
Assignment6
hasToken=true
Marker1
Receive1
replyMsg = int_chanReplyVar[getReceiveIndex(pid,i)]
int_chanReplyChan[getReceiveIndex(pid,i)]?
Default1
In3
i!=pid
Else3
!(i!=pid)
Condition4
i<NODES
!(i<NODES)
id34
For4
Assignment5
i=0
Condition3
!hasToken
!(hasToken)
id35
For3
id18
Send1
bool_chanRequestSize[getSendIndex(pid, broadcastCounter)] < bool_chanRequestBuffer
bool_chanRequestVar[getSendIndex(pid, broadcastCounter)] = true,
bool_chanRequestSize[getSendIndex(pid, broadcastCounter)]++
Assignment4
broadcastCounter++
wasInCs=true
Assignment33
hasToken=false
Marker5
Send2
int_chanReplyVar[getSendIndex(pid,(queue%10)-1)] = queue/10
int_chanReplyChan[getSendIndex(pid,(queue%10)-1)]!
bool_chanRequestSize[getReceiveIndex(pid,i)] == 0
requestMsg = bool_chanRequestVar[getReceiveIndex(pid,i)],
bool_chanRequestSize[getReceiveIndex(pid, i)]--
In2
i!=pid
Else17
!(i!=pid)
Else18
!(queue!=0)
Condition8
i<NODES
!(i<NODES)
For8
Assignment21
i=0
Condition7
hasToken
!(hasToken)
For7
id126
Condition2
broadcastCounter < NODES
!(broadcastCounter < NODES)
pid != broadcastCounter
Else2
pid == broadcastCounter
For2
Assignment3
broadcastCounter = 0
Marker4
Marker3
In1
!wasInCs
Else4
!(wasInCs)
Else11
!(!hasToken)
!hasToken
Condition1
i<5
!(i<5)
Marker6
id129
For1
Assignment2
i=0
Assignment1
hasToken=true
In
pid==0
Else1
!(pid==0)
Start

Bully Election Algorithm

First the GO specification is shown and then the UPPAAL model.

```
package main

/*
Example for Bully election algorithm.
*/

import (
  "fmt"
  "sync"
  "time"
)

// Array of chan for sending heartbeat to the leader
var bool_chanHeartbeat [EDGES]chan interface{}
// Array of chan for sending election msg
var int_chanElection [EDGES]chan interface{}
// Array of chan for sending the election result msg
var int_chanElectionResult [EDGES]chan interface{}

func main() {

  Init()

  // Init chans
  for i := range bool_chanHeartbeat {
    bool_chanHeartbeat[i] = make(chan interface{}, 1)
  }

  for i := range int_chanElection {
    int_chanElection[i] = make(chan interface{}, 10)
  }
```

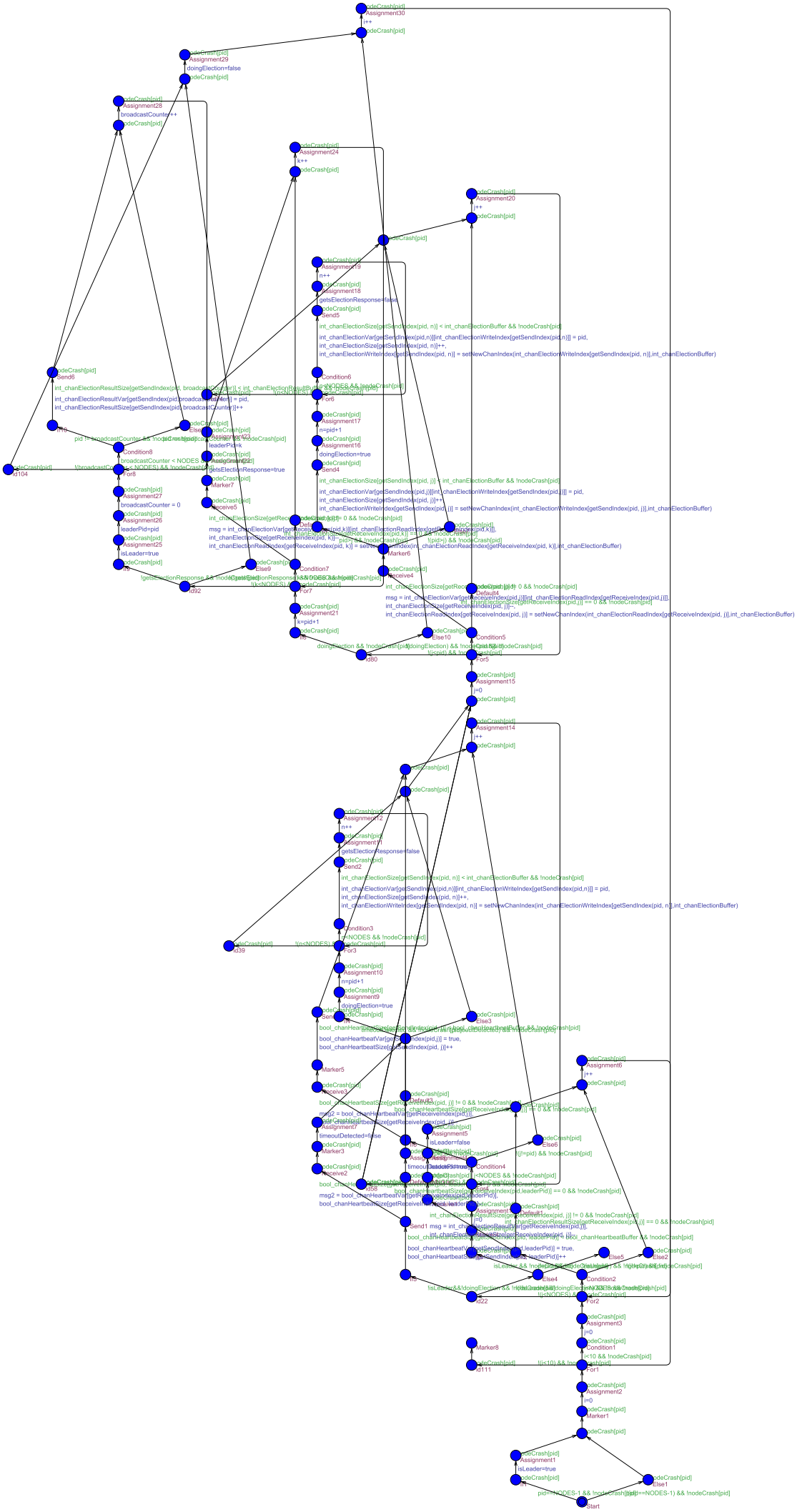
```
  for i := range int_chanElectionResult {
    int_chanElectionResult[i] = make(chan interface{}, 1)
  }

  fmt.Println("Nodes: ", NODES)

  // Waitgroup of all nodes
  var wg sync.WaitGroup
  wg.Add(NODES)

  go func() {
    node(0)
    wg.Done()
  }()
  go func() {
    node(1)
    wg.Done()
  }()
  go func() {
    node(2)
    wg.Done()
  }()

  wg.Wait()
}

// Node (can be leader or worker)
func node(pid int) {
  var isLeader bool = false
  var leaderPid int = NODES - 1
  var doingElection bool = false
  var getsElectionResponse bool = false
  var timeoutDetected bool = false
```

```
// Init Leader -> node with highest pid
if pid == NODES-1 {
  isLeader = true
}

fmt.Println("Hello from PID", pid, ". Leader is PID", leaderPid)

// Send to Leader while working -> working ends after 10 steps
for i := 0; i < 10; i++ {
  // Listen for new leader
  for j := 0; j < NODES; j++ {
    if j != pid {
      select {
      case msg :=
              <-int_chanElectionResult[GetReceiveIndex(pid, j)]:
        fmt.Println("PID", pid, "received new leader msg"
              , msg, "from PID", j, ".")
        leaderPid = msg.(int)
        isLeader = false
      default:
      }
    }
  }
  // Sending heartbeat if not leader node to leader
  if !isLeader && !doingElection {
    Send(pid, leaderPid, bool_chanHeartbeat, true)
    // Timeout simulation (wait 2 sec. for response)
    time.Sleep(2000 * time.Millisecond)
    select {
    case msg2 :=
          <-bool_chanHeartbeat[GetReceiveIndex(pid, leaderPid)]:
      fmt.Println("PID", pid, "received heartbeat response"
              , msg2, "from PID", leaderPid, ".")
      timeoutDetected = false
    default:
```

```
        timeoutDetected = true
      }

      if timeoutDetected { // Start election
        doingElection = true
        for n := pid + 1; n < NODES; n++ {
          Send(pid, n, int_chanElection, pid)
          getsElectionResponse = false
        }
      }
    } else if isLeader { // Listen for heartbeats if node is leader
      fmt.Println("Leader", pid, "answering heartbeats.")
      for j := 0; j < NODES; j++ {
        if j != pid {
          select {
          case msg2 :=
                <-bool_chanHeartbeat[GetReceiveIndex(pid, j)]:
            fmt.Println("PID", pid, "received received heartbeat"
                , msg2, "from PID", j, ".")
            Send(pid, j, bool_chanHeartbeat, true) // answer
          default:
          }
        }
      }
    }

    // Listen for election
    for j := 0; j < pid; j++ {
      select {
      case msg := <-int_chanElection[GetReceiveIndex(pid, j)]:
        fmt.Println("PID", pid, "received election msg", msg
                , "from PID", j, ".")
        if pid > j {
        // answer with pid if my pid is greater
          Send(pid, j, int_chanElection, pid)
```

```
        doingElection = true
        for n := pid + 1; n < NODES; n++ {
          Send(pid, n, int_chanElection, pid)
          getsElectionResponse = false
        }
      }
    default:
    }
  }

  if doingElection {
    for k := pid + 1; k < NODES; k++ {
      // Timeout simulation (wait 2 sec. for response)
      time.Sleep(2000 * time.Millisecond)
      select {
      case msg := <-int_chanElection[GetReceiveIndex(pid, k)]:
        fmt.Println("PID", pid, "received election response"
              , msg, "from PID", k, ".")
        getsElectionResponse = true
        leaderPid = k
      default:
      }
    }

    // Tell all that there is a new leader
    if !getsElectionResponse {
      isLeader = true
      leaderPid = pid
      Broadcast(pid, int_chanElectionResult, pid)
    }
    doingElection = false
  }

  // wait after every step
  time.Sleep(500 * time.Millisecond)
```

```
    }
    fmt.Println("PID", pid, "terminated!")
}
```


Start
Assignment1
isLeader=true
Else1
pid != NODES-1 && !nodeCrash[pid] || i==NODES-1) && !nodeCrash[pid]
Assignment2
i=0
Marker1
Assignment3
i=0
Condition1
i<10 && !nodeCrash[pid]
For1
!(i<10) && !nodeCrash[pid]
Marker8
d111
Assignment4
i=0
For2
!(i<NODES) && !nodeCrash[pid]
d22
Else2
isLeader && !doingElection && !nodeCrash[pid]
Else4
Condition2
Else5
isLeader && !doingElection && !nodeCrash[pid]
Else6
!(j!=pid) && !nodeCrash[pid]
Assignment5
isLeader=false
Assignment6
j++
Condition4
timeoutDetected && !nodeCrash[pid]
d88
Assignment7
timeoutDetected=false
Marker3
Receive2
Default3
Assignment8
i=0
For4
!(i<NODES) && !nodeCrash[pid]
d99
Assignment9
doingElection=true
Send1
Else3
Assignment10
n=pid+1
Condition3
For3
!(n<NODES) && !nodeCrash[pid]
d39
Assignment11
n++
Assignment12
getsElectionResponse=false
Send2
Marker5
Receive3
Default2
Assignment13
i=0
For5
!(i<NODES) && !nodeCrash[pid]
d80
Assignment14
j++
Condition5
Else10
doingElection && !nodeCrash[pid]
Assignment15
i=0
Marker6
Receive4
Default4
Assignment16
doingElection=true
Send4
Else9
Assignment17
n=pid+1
Condition6
For6
!(n<NODES) && !nodeCrash[pid]
d59
Assignment18
getsElectionResponse=false
Send5
Marker7
Receive5
Default5
Assignment19
n++
Assignment20
j++
Condition7
For7
!(k<NODES) && !nodeCrash[pid]
d92
Assignment21
k=pid+1
Assignment22
broadcastCounter=0
Condition8
For8
!(broadcastCounter < NODES) && !nodeCrash[pid]
d104
Assignment23
leaderPid=pid
Assignment24
k++
Assignment25
isLeader=true
Else8
getsElectionResponse && !nodeCrash[pid]
Assignment26
leaderPid=pid
Assignment27
broadcastCounter = 0
Assignment28
broadcastCounter++
Assignment29
doingElection=false
Assignment30
i++
Send6

BIBLIOGRAPHY

[1] GeekForGeeks (ihritik). *Huang's Termination detection algorithm*. 2019. URL: https://www.geeksforgeeks.org/huangs-termination-detection-algorithm/.

[2] Wahabou Abdou, Nesrine Ouled Abdallah, and Mohamed Mosbah. "ViSiDiA: A Java Framework for Designing, Simulating, and Visualizing Distributed Algorithms". In: *2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*. 2014, pp. 43–46. DOI: 10.1109/DS-RT.2014.14.

[3] Gerd Behrmann, Alexandre David, and Kim G Larsen. "A tutorial on uppaal". In: *Formal methods for the design of real-time systems* (2004), pp. 200–236.

[4] Mordechai Ben-Ari. "Interactive execution of distributed algorithms". In: *Journal on Educational Resources in Computing (JERIC)* 1.2es (2001), 2–es.

[5] Mordechai Ben-Ari. *jBACI Concurrency Simulator*. 2004. URL: https://www.weizmann.ac.il/sci-tea/benari/software-and-learning-materials/jbaci-concurrency-simulator.

[6] Johan Bengtsson and Wang Yi. “Timed automata: Semantics, algorithms and tools”. In: *Advanced Course on Petri Nets*. Springer. 2003, pp. 87–124.

[7] Martin Biely et al. “Distal: A framework for implementing fault-tolerant distributed algorithms”. In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2013, pp. 1–8. DOI: 10.1109/DSN.2013.6575306.

[8] Sasa Coh et al. *Github: antlr/grammars-v4*. 2021. URL: https://github.com/antlr/grammars-v4/tree/4c06ad8cc8130931c75ca0b17cbc1453f3830cd2/golang.

[9] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005, pp. 482–484.

[10] J Dekker, F Vaandrager, and R Smetsers. “Generating a google go framework from an uppaal model”. PhD thesis. Master’s thesis, Radboud University, 2014.

[11] O‘Donnell Fionnuala. “Simulation Frameworks for the Teaching and Learning of Distributed Algorithms.” In: (2006).

[12] Wan Fokkink. *Distributed algorithms: an intuitive approach*. Mit Press, 2018.

[13] Apache Software Foundation. *Apache Maven Project*. 2022. URL: https://maven.apache.org/.

[14] Torben Friedrich Görner. *Github: ZiggyStardustAndTheSpidersFromMars / DiAlGo-Project*. 2022. URL: https://github.com/ZiggyStardustAndTheSpidersFromMars/DiAlGo-Project.

[15] Charles Antony Richard Hoare. “Communicating sequential processes”. In: *Communications of the ACM* 21.8 (1978), pp. 666–677.

[16] Shing-Tsaan Huang. “Termination detection by using distributed snapshots”. In: *Information Processing Letters* 32.3 (1989), pp. 113–119.

[17] Google Inc. *A Tour of Go*. 2022. URL: https://go.dev/tour.

[18] Google Inc. *The Go ’sync’ Package*. 2022. URL: https://pkg.go.dev/sync.

[19] Google Inc. *The Go Programming Language Specification*. 2022. URL: https://go.dev/ref/spec.

[20] Boris Koldehofe, Marina Papatriantafilou, and Philippas Tsigas. “LYDIAN: An extensible educational animation environment for distributed algorithms”. In: *Journal on Educational Resources in Computing (JERIC)* 6.2 (2006), 1–es.

[21] Riverbank Computing Limited. *PyQt5*. 1995-2022. URL: https://pypi.org/project/PyQt5/.

[22] Y. Moses et al. “Algorithm visualization for distributed environments”. In: *Proceedings IEEE Symposium on Information Visualization (Cat. No.98TB100258)*. 1998, pp. 71–78. DOI: 10.1109/INFVIS.1998.729561.

[23] Diego Ongaro and John Ousterhout. *The Raft Consensus Algorithm*. 2014. URL: https://raft.github.io/.

[24] Terence Parr. *ANTLR (ANother Tool for Language Recognition)*. 2022. URL: https://www.antlr.org/.

[25] Terence Parr. *StringTemplate*. 2013. URL: https://www.stringtemplate.org/.

[26] Arne Philipeit. “Verification of concurrent Go programs using Uppaal _”. PhD thesis. Trinity College Dublin . . ., 2020.

[27] Ichiro Suzuki and Tadao Kasami. “A distributed mutual exclusion algorithm”. In: *ACM Transactions on Computer Systems (TOCS)* 3.4 (1985), pp. 344–349.

[28] The JUnit Team. *JUnit4*. 2021. URL: https://junit.org/junit4/.

[29] Uppsala University and Aalborg University. *UPPAAL*. 2008-2022. URL: https://uppaal.org/.