From Timed Automata to Timed Programs

Research by: Jonah Pears
`jp657@kent.ac.uk`

Supervised by: Laura Bocchi
`L.Bocchi@kent.ac.uk`

Word Count: 4417

University of Kent

Contents

Abstract

Communicating Timed Automata (CTA) are able to model a group of system's components which interact with each other. They are used as theoretical models of asynchronous communications, such as internet protocols. A CTA is comprised of multiple automata, each modelling some smaller subsystem which, together, yield a bigger system.

Establishing a specification in the form of a CTA model is useful when designing a timed asynchronous system. Specifying the constraints and details about how the different components can interact with each other is the purpose of modelling these systems. It allows the creator to restrict and ensure that everything can only act as intended, before committing to implementing the design. Theoretical models are also a powerful design tool as they are able to be verified and simulated. UPPAAL model checker [4] is one such tool. This allows the creator to establish if their design is fit for purpose rather than waste time implementing something that wouldn't work. The current issue is that these models only help the user understand *what* the system should do, not *how* this should be implemented in code. Once models are proven to be valid, the implementation is not always straightforward. It can be difficult to remain faithful to semantics of the original model, and preserving its properties and enforcing its constraints.

The goal of this project is to implement a system of generating `Go` code from an automata model, by mapping and establishing links between the two. There are several methods of verifying that the code generated retains the original meaning of the model. The way the project would be tested would be by using a model and its complements and seeing whether they retain their relationship in the generated code.

1 Introduction

Communicating Timed Automata are used to model the interactions between a group of systems. Their main purpose is to limit how the different systems can interact with each other and add structure by specifying the timing of their communications. It is the timing of the interactions between two given systems, and *when* these occur that is important in CTA models.

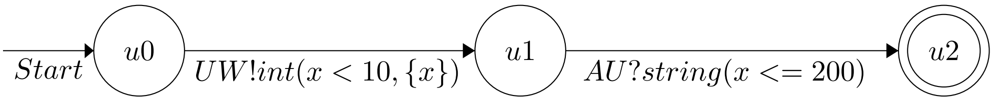
This is all CTA are; models of timing specifications of interacting systems. With a design specification in place, the actual process of implementing that specification is open to interpretation, and furthermore, misinterpretation. What this project explores is how feasible it would be to use the specification of the CTA to do just that. To take a CTA and generate its implementation as a `Go` program. This would leave no ambiguity in how the different systems can interact with each other as all of their possible interactions will come directly from the CTA model. An important thing to note is that the CTA models the timings from one state to another: *When* the different subsystems can interact with each other. *What* the system is actually computing or the tasks being carried out are outside the scope of this project. The programs generated by this project could be treated as scaffolding. The user would be able to insert *what* happens, once the structure is laid out.

1.1 Communicating Timed Automata

An automata can a model a system. A CTA is a model that consists of multiple automata that communicate with each other. They are a modelling tool used to specify how different systems interact with each other. What is being communicated is outside the scope of these models; rather they aim to specify the timing restrictions for transitioning between states, and how the timing affects the decision flow of the overall system. Because of this, the transition between different states consists of the destination, an arbitrary placeholder for the data and the time constraints of the transition. An example of this is shown in Figure 1.

Figure 1 models the communication specifications of a system that has 3 states. The first is a start state, *'u0'*. To progress to the next state, *'u1'*, this automaton needs to send an *'int'* to another system, or through a channel, under the label *'UW'*. This transition can only occur when the clock, x, is less than 10. Once this transition occurs the clock for that automata is reset. If that transition ever occurs, the automaton will then be in state *'u1'*, where it has one possible outward transition to *'u2'*, the end state. To transition to the end state it needs to receive a 'string' from the system 'AU' when the clock is less than or equal to 200.

Automata:



Notation: `Cta U = Init u0;u0 UW!int(x < 10,{x}) u1;u1 AU?string(x <= 200) u2;`

Figure 1: Example automata with corresponding notation

All of the automata graphs were made using this online tool: `http://madebyevan.com/fsm/`

1.2 CTA Notation

There is a specific notation for CTA that will be used for this project. This has been inherited from a previous project [3], and by using this, it opens up the possibility of compatibility. This project is motivated to have compatibility, as it could lead to the creation of a toolchain where the entire process of using CTA models to generate their own implementations could be automated. An example of an automata and its corresponding notation is shown in Figure 1.

The notation starts with defining the CTA name, and its initial state. In this case, *U* and *u0*. Then the notation continues by listing all the transitions that can be taken. A transition in this notation consists of a beginning and end state, the sending or receiving of data, and a time restraint for this to happen.

1.3 Artefacts

The program developed throughout this report can be found at `https://github.com/thecathe/From-Timed-Automata-to-Timed-Programs`. The program is written in Python and generates a `Go` program. The program synthesised only focuses on *how* the automata interact and communicate with each other, but not *what* the program as a whole does. The programs are able to run and will carry out their communications until they terminate. Whether the programs work is solely dependant on the quality of the CTA model used.

The program takes a CTA in the notation as arguments. Each automata should be given as a string, with as many arguments as there are automata in the CTA. Each automata in the CTA is generated as its own function, to be run as a goroutine. Each of the possible transitions within them must consist of both some form of communication with another automata, and a time constraint. The main function instantiates them all as goroutines.

1.4 Testing Overview

The testing carried out in this report will not be formal or theoretical. It will consist of using the traces that the generated programs produce, and determining if they are faithful to the CTA model they are supposed to implement.

2 Project Aims

This project aims to create a program that can synthesise code when given a CTA.

2.1 Toolchain compatibility

The long term goal is to be able to create a tool-chain that aids the development of asynchronous communicating systems. There is prior work in the field of aiding the design of CTA. With the products of these projects a toolchain may be able to be created. Below is an overview of each of the toolchains components.

1. **Model Creation** — First a CTA model needs to be created. The UPPAAL Model Checker is state of the art software that aids in modelling real-time systems.

2. **Model Verification** — The UPPAAL Model Checker is also able to verify and validate models [5]. If the user designs the model in this software, it is able to automatically verify the model's integrity. The software is also able to simulate the model.

3. **Model Refinement** — A valid model can be refined, however the issue is preserving the meaning behind the model. A previous paper explored this issue[1]. The paper proposes a theory of refining timed asynchronous systems. It achieves this by enforcing stricter time constraints on the models, whilst preserving their meaning. This part of the tool-chain supports the code generation the most.

4. **Code Generation** — This component is the subject of this project: to generate code from an automata model. The user would go through all previous steps and then be able to automatically generate `Go` code that implements their model.

An interesting thing to note is that UPPAAL have developed another piece of software [1] that aims at solving the same thing this project is, generating timed programs. In the paper [2], the implementations generated are compared to another tool.

2.2 Proposed Solution

A simplistic approach has been taken when solving this problem. The main focus is creating programs that completely implement a given CTA. The method that is developed in this project aims to be robust and simple. This is both so that it can handle complex CTA and at the same time, produce user friendly programs. A user needs to be able to understand the program and add *what* is actually happening at each state.

The implementation of CTA in this project will aim to emulate its behaviour as a CTA. This project is deciding to emulate the behaviour over the alternative of interpreting the meaning and generating code that implements it. Figure 2 shows an example of how an automaton would be emulated in pseudocode. In the automaton diagram it shows that it will send 'data' as many times as it can to A2, before x reaches 10. At that point it will wait 10 seconds and then wait until it receives 'data' from A2. Once 'data is received from A2 the clock is reset. It then waits 20 seconds and sends one final 'data' to A3. At this state it terminates. Following the flow of the pseudocode it is clear that this behaviour is directly imitated.

```
Automaton1()
    current_state = initial_state
    do
        select current_state
            case state_1
                // transition 1
                if x < 10
                    Automaton2!data
                    current_state = state_1
                // transition 2
                if x > 20
                    Automaton2?data
                    current_state = state_2
                    x = 0
                break
            case state_2
                // transition 1
                if x > 20
                    Automaton3!data
                    current_state = state_3
                break
    until current_state in end_states
```

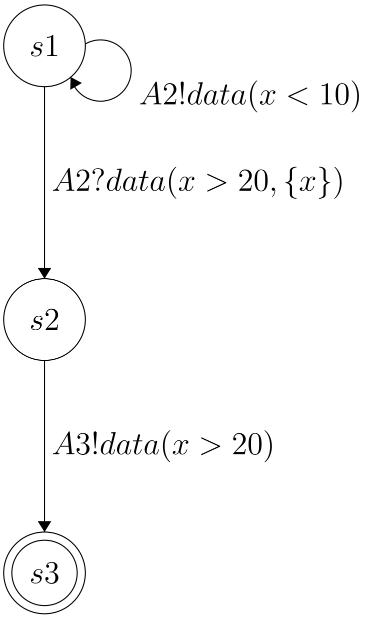


Figure 2: Pseudocode describing a single automaton, corresponding CTA.

2.3 Design Decisions

The way to implement a given CTA as a program whilst retaining its semantics is not explicitly defined, specifically when designing a method to automate this procedure. There are different ways to interpret the semantics of CTA, and, as a result, issues can arise, such as semantics of automata being misinterpreted and thus incorrectly implemented.

2.3.1 Structure

The overall structure that will be used in this project has been chosen due to its simplicity and robustness. These are the main foci for this project. Although elegant and succinct programs are likely to be more efficient, because this is a proof of concept on synthesising code from a CTA notation, this project is opting for one that is more easily automatable.

Emulation The structure proposed in this project could be described as more of an attempt to completely emulate the behaviour of the CTA, instead of interpreting it and implementing the semantics soundly. The hope is that by emulating it, no detail is left out and the programs produced completely encapsulate the semantics of the CTA model.

2.3.2 Clocks

The clocks of each automaton are going to be run in the `main` function of the program. The reason for this is so that they will progress synchronously. Each automaton goroutine will require its own clock, and each will need to be able to have the capability to reset it.

3 Version 1

This project has produced quite a few programs, some of them being more of a success than others. Version 1 is the first capable of producing working examples. Version 1 can be found at `https://github.com/thecathe/From-Timed-Automata-to-Timed-Programs/tree/Version_1`. Version 1 takes automata strings as arguments. It asks the user what labels should be used for each of the automata, and what to name the file outputted.

The first attempt at automatically generating executable `Go` programs has some limitations. There are 2 files that deal with generating the `Go` code; **CTA Loader**, which deals with reading in the CTA notation, and **Golang Generator**, which goes through the loaded CTA and generates the code.

3.1 Method

Initially, the method of solving this problem is implemented manually, and then a program is written to do this automatically. This is tested on several complementary automata. Complementary CTA are used because they reflect each other and thus clearly exhibit whether the `Go` program generated reflects this relationship.

3.1.1 Complement Example

The model shown in Figure 3 can be implemented as Go program. It is very important with timed or concurrent systems that are being modelled, that the semantics are maintained and truthful through the conversion. The full code used in this example can be found in Section 5.2.1.

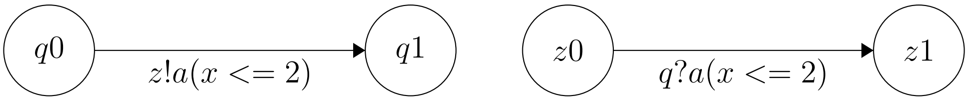


Figure 3: A complementary CTA model.

All code examples used can be found and run at: `https://repl.it/@cathe/test-templates`

In Figure 4 the global variables needed in this implementation are shown. The speed of the system can be controlled by adjusting the interval between increments, or the size of the increments. Each automata clock is global so that they can be run synchronously, from `main` shown in Figure 5, and the automata can still reset them. There are also global

Boolean variables that allow for the program to stop running when all goroutines have reached an end state.

```go
// speed of system
const clock_speed = 1
const clock_increment = 1
```

```go
// set up clocks
var q_clock int = 0
var z_clock int = 0
```

```go
// set up notifiers
var q_fin bool = false
var z_fin bool = false
```

Figure 4: Global variables used in this program.

The `main` function is where all of the channels will be initialised, goroutines started and clocks run. The decision to have the clocks run remotely from the automata itself is to ensure that they always progress synchronously.

```go
func main() {
  // set up channels
  channel_q_z_string := make(chan string, 2)

  // run goroutines
  go automata_q(channel_q_z_string)
  go automata_z(channel_q_z_string)

  // run clocks
  for {
    time.Sleep(time.Second * clock_speed)
    // increment clocks
    q_clock += clock_increment
    z_clock += clock_increment
    // check if goroutines have ended
    if q_fin && z_fin {
      break
    }
  }
  fmt.Println("CTA finished running.")
}
```

Figure 5: The `main` function of the implemented example CTA.

The automata cannot start until the clock system does. For a CTA model to be robustly implemented, the timings of each automata need to be strictly managed. The implementation in Figure 5 is able to do this effectively. The clocks will run until all of the goroutines have finished running, and then the program will stop.

The functions that represent and implement each automaton in the CTA shown in Figure 3 will take all of the channels that they can use as parameters. The alternative would be to have them all be global variables. Besides this being bad practice, it also makes the code less interpretable by the user. The method that this project develops needs to be a robust and user friendly scaffold.

```go
func automata_q(channel_q_z_string chan string) {
  // initial state
  current_state := "q0"
  // set up clock
  x := q_clock
  fmt.Printf("automata_q: %s: %v: Starting...\n", current_state, x)
  // repeat until end state reached
  for {
    // update clock
    if x != q_clock {
      x = q_clock
      switch current_state {
      case "q0":
        if x <= 2 {
          fmt.Printf("automata_q: %s: %v: Checking q0.\n", current_state, x)
          // send
          channel_q_z_string <- "a"
          // next state
          current_state = "q1"

          fmt.Printf("automata_q: %s: %v: Left q0.\n", current_state, x)
          continue
        }
      }
    }
    // check if an end state has been reached
    switch current_state {
      case "q1":
        fmt.Printf("automata_q: %s: %v: End state reached.\n", current_state, x)
        q_fin = true
        break
    }
  }
  q_fin = true
}
```

Figure 6: Goroutine implementing the transitions of automata Q.

A Go implementation of automaton Q, described in Figure 3, is shown in Figure 6. This is scaffold code that implements the asynchronous specification of Q within this model. This function implements the specification given by the CTA model for automaton Q, and allows the user to implement what data or information actually takes place at each state. This example is very simple: it sends the letter ”a” to Z, along the channel specified, and then ends.

This process could be simplified and written far more succinctly. However, this would either require a higher level of abstraction, which could lead to making assumptions on the semantics of the notation, or by having significant expertise in writing Go code, which the researcher of this project does not. By having a method that works to *emulate* the behaviour of the automata, the risk of misinterpretation is removed. This program works to act exactly as the automata should, not as an efficient program.

```go
case "z0":
  if x <= 2 {
    fmt.Printf("automata_z: %s: %v: Checking z0.\n", current_state, x)
    // check if receive is available
    select {
    case receive, ok := <-channel_q_z_string:
      if ok {
        // received
        _ = receive
        // next state
        current_state = "z1"

        fmt.Printf("automata_z: %s: %v: Left z0.\n", current_state, x)
        continue
      } else {
        // channel closed
        fmt.Printf("automata_z: %s: %v: ERROR: channel not open.\n", current_state, x)
      }
    default:
      // nothing in channel
      fmt.Printf("automata_z: %s: %v: Waiting to receive.\n", current_state, x)
    }
  }
```

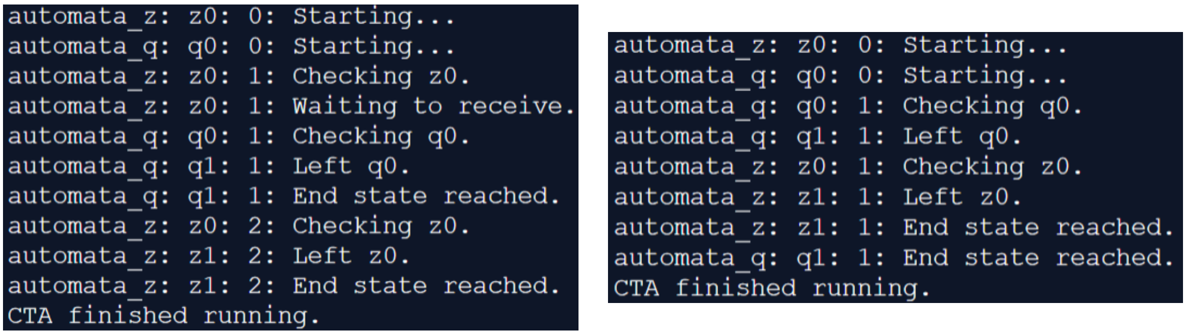
Figure 7: Goroutine implementing the transitions of automata *Z*.

Following the CTA model in Figure 3, automata *Z* represents *Q*‘, shown in Figure 7. This state is clearly larger than its equivalent, *q0*. This is because this is a receive transition. Receives are, by default, blocking in `Go`. When receiving, the program waits, and the only way to leave this point in the program would be by having a timeout. For some applications this is okay, but in the case of this project, the program needs to be able to check for the first available transition. If a state had multiple transitions, and one of them was a receive, it would get stuck at the receive transition until something arrived on that channel. Sending something on a channel is not like this, instead its more along the lines of ’send and forget’, or non-blocking. Aside from the special requirements of checking if a channel is not empty when receiving, this state transition is very similar to a send. The user is able to decide what actually happens in this system, at the point *receive*.

3.1.2 Complement Test

The implementation of the CTA shown in Figure 3 has printouts to help with debugging. It helps show the flow of control as it runs and can indicate whether it is truthful to the original CTA semantics. Figure 8 shows the output of the program when run two

different times. When run many times every output falls into one of these two patterns. The format of all of the traces produced by this program is the following:
automata name: automata state: automata clock: message.



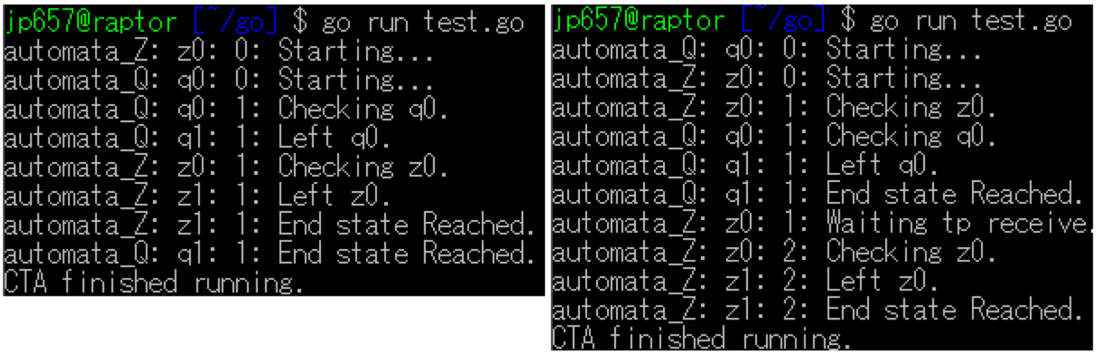
```
automata_z: z0: 0: Starting...
automata_q: q0: 0: Starting...
automata_z: z0: 1: Checking z0.
automata_z: z0: 1: Waiting to receive.
automata_q: q0: 1: Checking q0.
automata_q: q1: 1: Left q0.
automata_q: q1: 1: End state reached.
automata_z: z0: 2: Checking z0.
automata_z: z1: 2: Left z0.
automata_z: z1: 2: End state reached.
CTA finished running.
```

```
automata_z: z0: 0: Starting...
automata_q: q0: 0: Starting...
automata_q: q0: 1: Checking q0.
automata_q: q1: 1: Left q0.
automata_z: z0: 1: Checking z0.
automata_z: z1: 1: Left z0.
automata_z: z1: 1: End state reached.
automata_q: q1: 1: End state reached.
CTA finished running.
```

Figure 8: Output of the `Go` implementation of Figure 3.

One hypothesis is that out of the two goroutines that implement each automata, one of them has to be first to check its transitions. There is no way for them to check at the same time. Some of the time, Z checks first, and has to wait until the next time progression. Q is able to carry its transition out and finishes. Then the clock progresses and Z is then able to finish. The other option is that Q goes first, and when this happens, both of the automata are able to finish before the clocks reach *2*.

One possibility is that this is caused by being run on a system with only one processor, as this means the goroutines would have to occupy the same processor and inconsistencies over which one starts first may arise. However, when run on raptor, a system with multiple processors, the same potential issue arises as shown in Figure 9.



```
jp657@raptor [~/go] $ go run test.go
automata_Z: z0: 0: Starting...
automata_Q: q0: 0: Starting...
automata_Q: q0: 1: Checking q0.
automata_Q: q1: 1: Left q0.
automata_Z: z0: 1: Checking z0.
automata_Z: z1: 1: Left z0.
automata_Z: z1: 1: End state Reached.
automata_Q: q1: 1: End state Reached.
CTA finished running.
```

```
jp657@raptor [~/go] $ go run test.go
automata_Q: q0: 0: Starting...
automata_Z: z0: 0: Starting...
automata_Z: z0: 1: Checking z0.
automata_Q: q0: 1: Checking q0.
automata_Q: q1: 1: Left q0.
automata_Q: q1: 1: End state Reached.
automata_Z: z0: 1: Waiting tp receive.
automata_Z: z0: 2: Checking z0.
automata_Z: z1: 2: Left z0.
automata_Z: z1: 2: End state Reached.
CTA finished running.
```

Figure 9: Raptor trace.

3.2 Code Synthesis

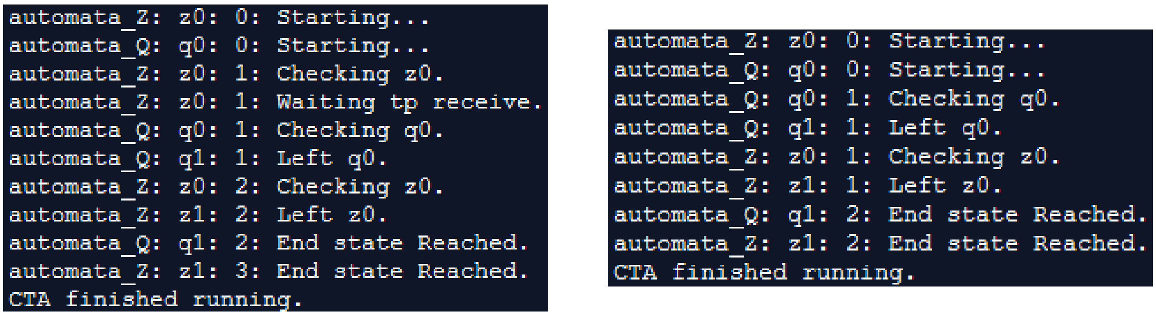
Version 1 implements the method devised and tested manually, to automatically generate `Go` programs when given CTA notation as arguments.

3.2.1 Complement Case 1

```
Cta Q = Init q0;q0 z!a(x <= 2) q1;
Cta Z = Init z0;z0 q?a(x <= 2) z1;
```

Figure 10: Notation of the CTA shown in Figure 3.

There are some notable differences in the trace produced by the generated program and the one that was written manually, though both implement the same CTA model. Figure 11 shows traces of the program and clearly shows a similar pattern to Figure 8 but with some differences.



```
automata_Z: z0: 0: Starting...
automata_Q: q0: 0: Starting...
automata_Z: z0: 1: Checking z0.
automata_Z: z0: 1: Waiting tp receive.
automata_Q: q0: 1: Checking q0.
automata_Q: q1: 1: Left q0.
automata_Z: z0: 2: Checking z0.
automata_Z: z1: 2: Left z0.
automata_Q: q1: 2: End state Reached.
automata_Z: z1: 3: End state Reached.
CTA finished running.
```

```
automata_Z: z0: 0: Starting...
automata_Q: q0: 0: Starting...
automata_Q: q0: 1: Checking q0.
automata_Q: q1: 1: Left q0.
automata_Z: z0: 1: Checking z0.
automata_Z: z1: 1: Left z0.
automata_Q: q1: 2: End state Reached.
automata_Z: z1: 2: End state Reached.
CTA finished running.
```

Figure 11: Output of the `Go` implementation of Figure 3.

An interesting point to make is that the automata Z ends up finishing at time 3. This might seem to contradict what is specified in the notation shown in Figure 10. However, the trace indicates that the change from state *z0* to *z1* occurred at time 2.

After being run extensively, these are the only two variations in its trace. There doesn't seem to be anything else this particular example's trace can show us.

3.2.2 Ping Pong Example

The next CTA example that will be tested contains loops. It is a model of two systems that ping and pong each other as much as they can within 5 time steps.

In this system, shown in Figure 12, it is Q that counts to 5. Z is just a slave system, always responding to whatever Q does. An important thing to remember is that the data

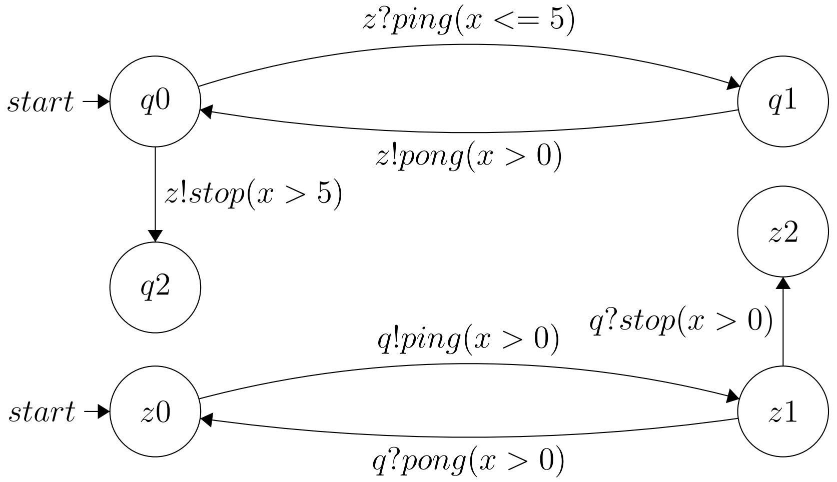


Figure 12: CTA model of ping-pong.

being communicated here are just meaningless labels. What actually controls the flow of this system is the timing constraints. The entire program that has been generated with Version 1 can be seen in Section 5.3.2.

```
Cta Q = Init q0;q0 z?ping(x <= 5) q1;q0 z!stop(x > 5) q2;q1 z!pong(x > 0) q0;
 Cta Z = Init z0;z0 q!ping(x > 0) z1;z1 q?stop(x > 0) z2;z1 q?pong(x > 0) z0;
```

Figure 13: CTA notation of Figure 12.

The corresponding CTA notation for the CTA shown in Figure 12, is shown in Figure 13. Version 1 is also able to generate executable code for this CTA too. It's traces are shown in Figure 16, which can be found in Section 5.1, as the traces are too large to fit here. Once again it is interesting to note the variation between the two example traces. It never appears to amount to much, and in all of the tests that have been carried out, the CTA is always able to finish executing without any trouble. There was even an instance where the clock Z reached a time step of 10.

3.2.3 Sever Client Example

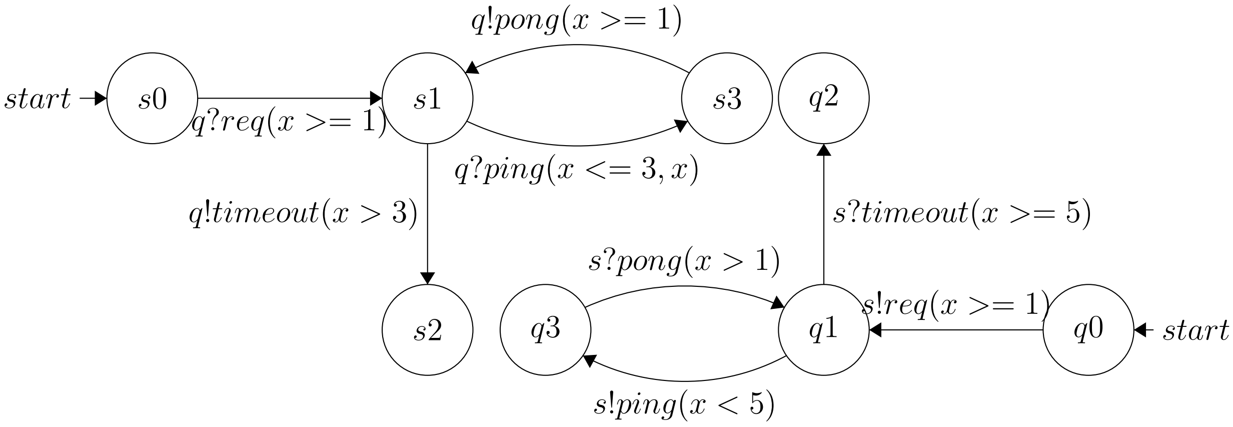


Figure 14: Another CTA representing a client, Q, pinging a server, S.

```
Cta S = Init s0;s0 q?req(x >= 1) s1;s1 q!timeout(x > 3) s2;s1 q?ping(x <= 3,{x}) s3;s3
                                  q!pong(x >= 1) s1;
   Cta Q = Init q0;q0 s!req(x >= 1) q1;q1 s?timeout(x >= 5) q2;q1 s!ping(x < 5) q3;q3
                                  s?pong(x > 1) q1;
```

Figure 15: CTA notation of Figure 14.

This example once again uses the idea of two systems pinging each other, CTA model shown in Figure 14. This time it is a client, Q, pinging a server, S. In this example, the client only plans on sending as many pings as possible within 5 time steps, and then waits until the server responds with a time out. The server only keeps the connection alive as long as it has received something in the last 3 time steps, otherwise it will send a timeout. The CTA notation is shown in Figure 15. The traces that show the flow of this CTA implementation can be found in the appendix in Figure 17.

3.3 Limitations & Issues

The main limitation with this design is that outward transitions from the same state should not have overlapping time constraints. If this happens, it will take the first one possible and not consider the others. It would have already changed state and continued to the next iteration of the loop.

- A potential issue is that the program actively waits for a transition to be come available. On any scale this is essentially a waste of resources. On a large scale this could slow the entire system down, if there were many automata doing this. Although in this program it is mitigated to once every time step.

- Manually running the clocks in `main` is perhaps redundant as `Go` has it's own system for doing this. This should be looked into when refactoring this program.

Issues There are some precise details on how the tool needs to be run, all of the limitations or issues surrounding the method itself are still present.

- 'Cta' and 'Init' need to be in this grammatical case.
- The timing condition needs to be one that `Go` can resolve. The tool does no checking or translating.
- The spacing of the notation must be as follows:
  `Cta Q = Init q0;q0 z!a(x <= 2) q1;`
- If a reset is needed in a transition, it must go exactly in the following position, with no spaces: `q0 z!a(x <= 2,{x}) q1;`

An issue this program runs into is knowing what data type to assign the channels. The only place in the notation where a potential data type could be, for example in Figure 3, would be *'a'*. However, this location in the notation is more of a label rather than meaning anything in particular, and although a data type such as 'string' or 'int' may be used in some models, it cannot be relied upon. The only method of reliably determining the data type would be by interacting with the user during code synthesis. For now, everything is treated as a string. All of the channels generated will be strings, and the data sent between will be strings.

Another limitation with this method of implementation, is that it doesn't support the conditions of the transition being something the program computes. Currently, anything provided in the brackets is considered a time restraint, but the program could possibly support the user further and allow for other types of conditions, aiding the user in not only implementing *how* the system works, but how *what* it does affects this.

3.4 Evaluation

Despite the issues and limitations of this program, the core aims of this project have been achieved. This version of the program is functional and will generate a working `Go` program, providing the user operates it correctly. There are certainly improvements that can be made to make it more successful.

It is important to keep in mind that a CTA model only specifies *how* the different systems interact with each other asynchronously, *what* the system actually does, and the data that is actually communicated between them is outside the scope of a CTA model.

An interesting point to note is the inconsistency of the implementations when using `Go`, shown in the traces. When run, the whole execution of the program can be offset a time step for each time the receiving state happens to goes first. If a CTA model is structured soundly enough this doesn't yield any errors, but it will cause inconsistencies over many executions.

3.5 Future Work

There is certainly more that can be done to aid the user and create more helpful implementations of the CTA model. To do this, the program would require far more user interaction, as the notation doesn't have the capabilities to describe the information needed. This information would include the data types used in communication and channels. To be able to do this would require more user interaction, or expanding on the notation.

If this project relied on more user interaction during the generation of the program, it would become far less automatic. This could take away the main purpose of trying to automate this process in the first place. The only benefit that would remain is that it would be a guided experience; the user wouldn't be able to interfere with the method, only supplement it. The user provides all of the information needed as the program runs, and with this abundant source of information the programs generated could be far more advanced and useful.

Improving the efficiency of the program produced is important. Removing the active waiting and manual clocks are revisions that should be made in future development of this project.

3.5.1 Tool Chain

Currently, this program could still be compatible with the proposed toolchain. It is able to take the same CTA notation as the previous parts and as long as the CTA models themselves match the criteria of this program, it could work, even in its current state.

4 Version 2

Version 2 is an improvement on Version 1 and focuses a lot more on the transitions and how they are chosen. It allows for overlapping transition conditions and allows the user to set a preference on which *sending* transitions should have a higher priority over others. Receive priorities remain passive as only the sender has any control over them. The method would remain largely the same as Version 1, but the program that automates them would have a lot more user interaction.

Unfortunately, no fully working Version 2 has been able to be produced. There have been many attempts, all built on Version 1 or on a refactoring of it, but the time constraints of the course and the disruptions of the COVID-19 pandemic have certainly stunted its development.

The work on Version 2 that has been made can be found in the corpus.

References

[1] Mokrushin L. Pettersson P. Yi W. Amnell T., Fersman E. Times: A tool for schedulability analysisand code generation of real-time systems. In *Formal Modeling and Analysis of Timed Systems.*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2003.

[2] Salem A. Sheirah M. Ayoub A., Wahba A. Code synthesis for timed automata: A comparison using case study. In *Abstract State Machines, Alloy, B and Z.*, volume 5977 of *Lecture Notes in Computer Science*, pages 403–403. Springer, 2010.

[3] Massimo Bartoletti, Laura Bocchi, and Maurizio Murgia. Progress-preserving refinements of CTA. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 40:1–40:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

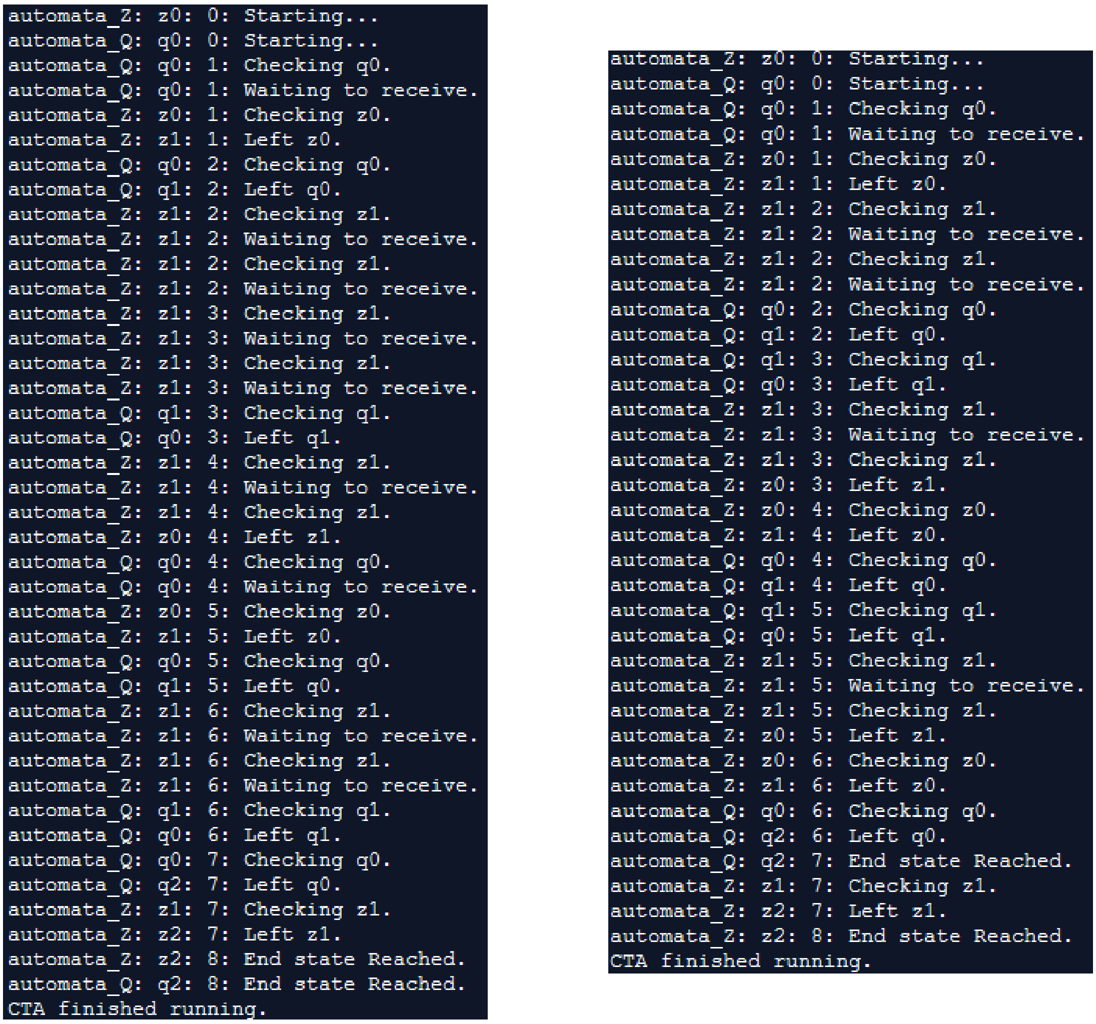
[4] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995. http://www.uppaal.org/.

[5] Simon Wimmer and Peter Lammich. Verified model checking of timed automata. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2018.

5 Appendix

This section contains all of the programs written in this project, except for the tool which can be found in the corpus. These programs are either generated using the tool, or written implementing the method. All of these implement a CTA. These are here as they cannot be shown in full, earlier in the report. All of the code shown in this report can be viewed and run online at `https://repl.it/@cathe/test-templates`.

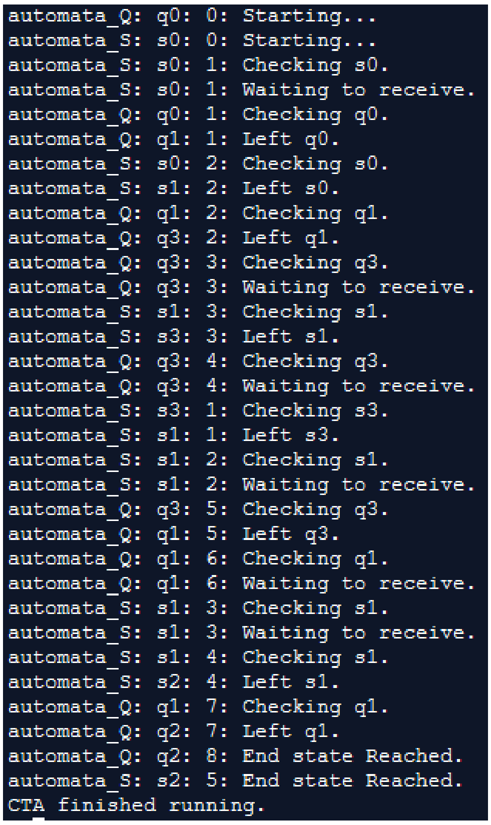
5.1 Traces



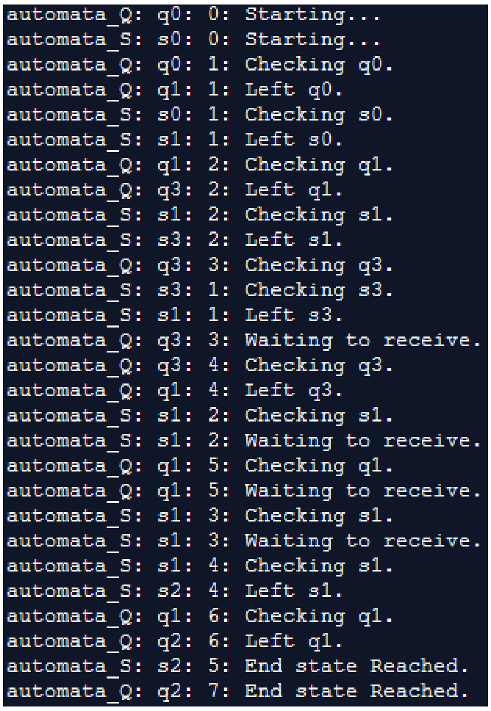
```
automata_Z: z0: 0: Starting...
automata_Q: q0: 0: Starting...
automata_Q: q0: 1: Checking q0.
automata_Q: q0: 1: Waiting to receive.
automata_Z: z0: 1: Checking z0.
automata_Z: z1: 1: Left z0.
automata_Q: q0: 2: Checking q0.
automata_Q: q1: 2: Left q0.
automata_Z: z1: 2: Checking z1.
automata_Z: z1: 2: Waiting to receive.
automata_Z: z1: 2: Checking z1.
automata_Z: z1: 2: Waiting to receive.
automata_Z: z1: 3: Checking z1.
automata_Z: z1: 3: Waiting to receive.
automata_Z: z1: 3: Checking z1.
automata_Z: z1: 3: Waiting to receive.
automata_Q: q1: 3: Checking q1.
automata_Q: q0: 3: Left q1.
automata_Z: z1: 4: Checking z1.
automata_Z: z1: 4: Waiting to receive.
automata_Z: z1: 4: Checking z1.
automata_Z: z0: 4: Left z1.
automata_Q: q0: 4: Checking q0.
automata_Q: q0: 4: Waiting to receive.
automata_Z: z0: 5: Checking z0.
automata_Z: z1: 5: Left z0.
automata_Q: q0: 5: Checking q0.
automata_Q: q1: 5: Left q0.
automata_Z: z1: 6: Checking z1.
automata_Z: z1: 6: Waiting to receive.
automata_Z: z1: 6: Checking z1.
automata_Z: z1: 6: Waiting to receive.
automata_Q: q1: 6: Checking q1.
automata_Q: q0: 6: Left q1.
automata_Q: q0: 7: Checking q0.
automata_Q: q2: 7: Left q0.
automata_Z: z1: 7: Checking z1.
automata_Z: z2: 7: Left z1.
automata_Z: z2: 8: End state Reached.
automata_Q: q2: 8: End state Reached.
CTA finished running.
```

```
automata_Z: z0: 0: Starting...
automata_Q: q0: 0: Starting...
automata_Q: q0: 1: Checking q0.
automata_Q: q0: 1: Waiting to receive.
automata_Z: z0: 1: Checking z0.
automata_Z: z1: 1: Left z0.
automata_Z: z1: 2: Checking z1.
automata_Z: z1: 2: Waiting to receive.
automata_Z: z1: 2: Checking z1.
automata_Z: z1: 2: Waiting to receive.
automata_Q: q0: 2: Checking q0.
automata_Q: q1: 2: Left q0.
automata_Q: q1: 3: Checking q1.
automata_Q: q0: 3: Left q1.
automata_Z: z1: 3: Checking z1.
automata_Z: z1: 3: Waiting to receive.
automata_Z: z1: 3: Checking z1.
automata_Z: z0: 3: Left z1.
automata_Z: z0: 4: Checking z0.
automata_Z: z1: 4: Left z0.
automata_Q: q0: 4: Checking q0.
automata_Q: q1: 4: Left q0.
automata_Q: q1: 5: Checking q1.
automata_Q: q0: 5: Left q1.
automata_Z: z1: 5: Checking z1.
automata_Z: z1: 5: Waiting to receive.
automata_Z: z1: 5: Checking z1.
automata_Z: z0: 5: Left z1.
automata_Z: z0: 6: Checking z0.
automata_Z: z1: 6: Left z0.
automata_Q: q0: 6: Checking q0.
automata_Q: q2: 6: Left q0.
automata_Q: q2: 7: End state Reached.
automata_Z: z1: 7: Checking z1.
automata_Z: z2: 7: Left z1.
automata_Z: z2: 8: End state Reached.
CTA finished running.
```

Figure 16: Trace of the `Go` implementation of Figure 12, ping pong.



```
automata_Q: q0: 0: Starting...
automata_S: s0: 0: Starting...
automata_S: s0: 1: Checking s0.
automata_S: s0: 1: Waiting to receive.
automata_Q: q0: 1: Checking q0.
automata_Q: q1: 1: Left q0.
automata_S: s0: 2: Checking s0.
automata_S: s1: 2: Left s0.
automata_Q: q1: 2: Checking q1.
automata_Q: q3: 2: Left q1.
automata_Q: q3: 3: Checking q3.
automata_Q: q3: 3: Waiting to receive.
automata_S: s1: 3: Checking s1.
automata_S: s3: 3: Left s1.
automata_Q: q3: 4: Checking q3.
automata_Q: q3: 4: Waiting to receive.
automata_S: s3: 1: Checking s3.
automata_S: s1: 1: Left s3.
automata_S: s1: 2: Checking s1.
automata_S: s1: 2: Waiting to receive.
automata_Q: q3: 5: Checking q3.
automata_Q: q1: 5: Left q3.
automata_Q: q1: 6: Checking q1.
automata_Q: q1: 6: Waiting to receive.
automata_S: s1: 3: Checking s1.
automata_S: s1: 3: Waiting to receive.
automata_S: s1: 4: Checking s1.
automata_S: s2: 4: Left s1.
automata_Q: q1: 7: Checking q1.
automata_Q: q2: 7: Left q1.
automata_Q: q2: 8: End state Reached.
automata_S: s2: 5: End state Reached.
CTA finished running.
```



```
automata_Q: q0: 0: Starting...
automata_S: s0: 0: Starting...
automata_Q: q0: 1: Checking q0.
automata_Q: q1: 1: Left q0.
automata_S: s0: 1: Checking s0.
automata_S: s1: 1: Left s0.
automata_Q: q1: 2: Checking q1.
automata_Q: q3: 2: Left q1.
automata_S: s1: 2: Checking s1.
automata_S: s3: 2: Left s1.
automata_Q: q3: 3: Checking q3.
automata_S: s3: 1: Checking s3.
automata_S: s1: 1: Left s3.
automata_Q: q3: 3: Waiting to receive.
automata_Q: q3: 4: Checking q3.
automata_Q: q1: 4: Left q3.
automata_S: s1: 2: Checking s1.
automata_S: s1: 2: Waiting to receive.
automata_Q: q1: 5: Checking q1.
automata_Q: q1: 5: Waiting to receive.
automata_S: s1: 3: Checking s1.
automata_S: s1: 3: Waiting to receive.
automata_S: s1: 4: Checking s1.
automata_S: s2: 4: Left s1.
automata_Q: q1: 6: Checking q1.
automata_Q: q2: 6: Left q1.
automata_S: s2: 5: End state Reached.
automata_Q: q2: 7: End state Reached.
```

Figure 17: Trace of the Go implementation of Figure 14, server client.

5.2 Manual

5.2.1 Complementary Example

This is the manually written program implementing the method. The CTA this implements is shown in Figure 3

```go
package main

import (
  "fmt"
  "time"
) //"math/rand"

// speed of system
const clock_speed = 1
const clock_increment = 1

// set up clocks
var q_clock int = 0
var z_clock int = 0

```

```go
// set up notifiers
var q_fin bool = false
var z_fin bool = false

func main() {
  // set up channels
  channel_q_z_string := make(chan string, 2)

  // run goroutines
  go automata_q(channel_q_z_string)
  go automata_z(channel_q_z_string)

  // run clocks
  for {
    time.Sleep(time.Second * clock_speed)
    // increment clocks
    q_clock += clock_increment
    z_clock += clock_increment
    // check if goroutines have ended
    if q_fin && z_fin {
      break
    }
  }

  fmt.Println("CTA finished running.")
}

func automata_q(channel_q_z_string chan string) {
  // initial state
  current_state := "q0"
  // set up clock
  x := q_clock
  fmt.Printf("automata_q: %s: %v: Starting...\n", current_state, x)
  // repeat until end state reached
  for {
    // update clock
    if x != q_clock {
      x = q_clock
      switch current_state {
      case "q0":
        if x <= 2 {
          fmt.Printf("automata_q: %s: %v: Checking q0.\n", current_state, x)
          // send
          channel_q_z_string <- "a"
          // next state
          current_state = "q1"

          fmt.Printf("automata_q: %s: %v: Left q0.\n", current_state, x)
          continue
        }
      }
    }

    // check if an end state has been reached
    if current_state == "q1" {
        fmt.Printf("automata_q: %s: %v: End state reached.\n", current_state, x)
        q_fin = true
        break
    }
  }
}

// q‘
func automata_z(channel_q_z_string chan string) {
  // initial state
  current_state := "z0"
  // setup clock
  x := z_clock
  fmt.Printf("automata_z: %s: %v: Starting...\n", current_state, x)
  // repeat until end state reached
  for {
    // update clock
    if x != z_clock {
      x = z_clock
      switch current_state {
      case "z0":
        if x <= 2 {
          fmt.Printf("automata_z: %s: %v: Checking z0.\n", current_state, x)
          // check if receive is available
          select {
          case receive, ok := <-channel_q_z_string:
            if ok {
              // received
              _ = receive
              // next state
              current_state = "z1"

              fmt.Printf("automata_z: %s: %v: Left z0.\n", current_state, x)
              continue
            } else {
              // channel closed
              fmt.Printf("automata_z: %s: %v: ERROR: channel not open.\n", current_state, x)
            }
          default:
            // nothing in channel
            fmt.Printf("automata_z: %s: %v: Waiting to receive.\n", current_state, x)
          }
        }
      }
    }

    // check if an end state has been reached
    if current_state == "z1" {
      fmt.Printf("automata_z: %s: %v: End state reached.\n", current_state, x)
      z_fin = true
      break
    }
  }
}
```

5.3 Synthesised

5.3.1 Complementary Case 1

This is a product of code synthesis, it has been automatically generated from the CTA notation shown in Figure 10. The CTA model this implements is shown in Figure 3, and uses the notation shown in Figure 10.

```
package main

import (
  "time"
  "fmt"
)

/* for the automata:
  Cta Q = Init q0;q0 z!a(x <= 2) q1;
  Cta Z = Init z0;z0 q?a(x <= 2) z1;
*/

// speed of system
const clock_speed = 1
const clock_increment = 1

// set up clocks
var Q_clock int = 0
var Z_clock int = 0

// set up notifiers
var Q_fin bool = false
var Z_fin bool = false

func main() {
  // set up channels
  channel_Q_Z_a := make(chan string, 2)

  // run goroutines
  go automata_Q(channel_Q_Z_a)
  go automata_Z(channel_Q_Z_a)
```

```go

  // run clocks
  for {
    time.Sleep(time.Second * clock_speed)
    //increment clocks
    Q_clock += clock_increment
    Z_clock += clock_increment
    // check if goroutines have ended
    if Q_fin && Z_fin {
      break
    }
  }

  fmt.Println("CTA finished running.")
}

// Cta Q = Init q0;q0 z!a(x <= 2) q1;
func automata_Q(channel_Q_Z_a chan string) {
  // initial state
  current_state := "q0"
  // set up clock
  x := Q_clock
  fmt.Printf("automata_Q: %s: %v: Starting...\n", current_state, x)
  // repeat until end state reached
  for {
    // update clock
    if x != Q_clock {
      x = Q_clock
      switch current_state {
      case "q0":
        if x <= 2 {
          fmt.Printf("automata_Q: %s: %v: Checking q0.\n", current_state, x)
          // send
          channel_Q_Z_a <- "a"
          // next state
          current_state = "q1"
          fmt.Printf("automata_Q: %s: %v: Left q0.\n", current_state, x)
          continue
        }
      }
      // check if end state has been reached
      if current_state == "q1" {
        fmt.Printf("automata_Q: %s: %v: End state Reached.\n", current_state, x)
        Q_fin = true
        break
      }
    }
  }
}

// Cta Z = Init z0;z0 q?a(x <= 2) z1;
```

```go
func automata_Z(channel_Q_Z_a chan string) {
  // initial state
  current_state := "z0"
  // set up clock
  x := Z_clock
  fmt.Printf("automata_Z: %s: %v: Starting...\n", current_state, x)
  // repeat until end state reached
  for {
    // update clock
    if x != Z_clock {
      x = Z_clock
      switch current_state {
      case "z0":
        if x <= 2 {
            fmt.Printf("automata_Z: %s: %v: Checking z0.\n", current_state, x)
            // check if receive is available
            select {
            case receive, ok := <-channel_Q_Z_a:
              if ok {
                // received
                _ = receive
                // next state
                current_state = "z1"

                fmt.Printf("automata_Z: %s: %v: Left z0.\n", current_state, x)
                continue
              } else {
                // channel closed
                fmt.Printf("automata_Z: %s: %v: ERROR: channel not open.\n",
    current_state, x)
              }
            default:
              // nothing in channel
              fmt.Printf("automata_Z: %s: %v: Waiting tp receive.\n", current_state, x)
            }
          }
        }
      // check if end state has been reached
      if current_state == "z1" {
        fmt.Printf("automata_Z: %s: %v: End state Reached.\n", current_state, x)
        Z_fin = true
        break
      }
    }
  }
}
```

5.3.2 Ping Pong

This is an automatically generated from the CTA notation shown in Figure 13, implementing the CTA model shown in Figure 12.

```go
package main

import (
  "time"
  "fmt"
)

/* for the automata:
  Cta Q = Init q0;q0 z?ping(x <= 5) q1;q0 z!stop(x > 5) q2;q1 z!pong(true) q0;
  Cta Z = Init z0;z0 q!ping(true) z1;z1 q?stop(true) z2;z1 q?pong(true) z0;
*/

// speed of system
const clock_speed = 1
const clock_increment = 1

// set up clocks
var Q_clock int = 0
var Z_clock int = 0

// set up notifiers
var Q_fin bool = false
var Z_fin bool = false

func main() {
  // set up channels
  channel_Z_Q_ping := make(chan string, 2)
  channel_Q_Z_stop := make(chan string, 2)
  channel_Q_Z_pong := make(chan string, 2)

  // run goroutines
  go automata_Q(channel_Z_Q_ping, channel_Q_Z_stop, channel_Q_Z_pong)
  go automata_Z(channel_Z_Q_ping, channel_Q_Z_stop, channel_Q_Z_pong)

  // run clocks
  for {
    time.Sleep(time.Second * clock_speed)
    //increment clocks
    Q_clock += clock_increment
    Z_clock += clock_increment
    // check if goroutines have ended
    if Q_fin && Z_fin {
      break
    }
  }

  fmt.Println("CTA finished running.")
}

// Cta Q = Init q0;q0 z?ping(x <= 5) q1;q0 z!stop(x > 5) q2;q1 z!pong(true) q0;
func automata_Q(channel_Z_Q_ping chan string, channel_Q_Z_stop chan string, channel_Q_Z_pong chan string) {
  // initial state
  current_state := "q0"
  // set up clock
  x := Q_clock
  fmt.Printf("automata_Q: %s: %v: Starting...\n", current_state, x)
  // repeat until end state reached
  for {
    // update clock
    if x != Q_clock {
      x = Q_clock
      switch current_state {
      case "q0":
        if x <= 5 {
            fmt.Printf("automata_Q: %s: %v: Checking q0.\n", current_state, x)
            // check if receive is available
            select {
            case receive, ok := <-channel_Z_Q_ping:
              if ok {
                // received
                _ = receive
                // next state
                current_state = "q1"

                fmt.Printf("automata_Q: %s: %v: Left q0.\n", current_state, x)
                continue
              } else {
                // channel closed
                fmt.Printf("automata_Q: %s: %v: ERROR: channel not open.\n", current_state, x)
              }
            default:
              // nothing in channel
              fmt.Printf("automata_Q: %s: %v: Waiting to receive.\n", current_state, x)
            }
        }
        if x > 5 {
          fmt.Printf("automata_Q: %s: %v: Checking q0.\n", current_state, x)
          // send
          channel_Q_Z_stop <- "stop"
          // next state
          current_state = "q2"
          fmt.Printf("automata_Q: %s: %v: Left q0.\n", current_state, x)
          continue
        }
      case "q1":
        if true {
          fmt.Printf("automata_Q: %s: %v: Checking q1.\n", current_state, x)
          // send
          channel_Q_Z_pong <- "pong"
          // next state
          current_state = "q0"
          fmt.Printf("automata_Q: %s: %v: Left q1.\n", current_state, x)
          continue
        }
      }
      // check if end state has been reached
      if current_state == "q2" {
        fmt.Printf("automata_Q: %s: %v: End state Reached.\n", current_state, x)
        Q_fin = true
        break
      }
    }
  }
}

// Cta Z = Init z0;z0 q!ping(true) z1;z1 q?stop(true) z2;z1 q?pong(true) z0;
func automata_Z(channel_Z_Q_ping chan string, channel_Q_Z_stop chan string, channel_Q_Z_pong chan string) {
  // initial state
  current_state := "z0"
  // set up clock
  x := Z_clock
  fmt.Printf("automata_Z: %s: %v: Starting...\n", current_state, x)
  // repeat until end state reached
  for {
    // update clock
    if x != Z_clock {
      x = Z_clock
      switch current_state {
      case "z0":
        if true {
          fmt.Printf("automata_Z: %s: %v: Checking z0.\n", current_state, x)
          // send
          channel_Z_Q_ping <- "ping"
          // next state
          current_state = "z1"
          fmt.Printf("automata_Z: %s: %v: Left z0.\n", current_state, x)
          continue
        }
      case "z1":
        if true {
            fmt.Printf("automata_Z: %s: %v: Checking z1.\n", current_state, x)
            // check if receive is available
            select {
            case receive, ok := <-channel_Q_Z_stop:
              if ok {
```

```
        // received
        _ = receive
        // next state
        current_state = "z2"

        fmt.Printf("automata_Z: %s: %v: Left z1.\n", current_state, x)
        continue
       } else {
        // channel closed
        fmt.Printf("automata_Z: %s: %v: ERROR: channel not open.\n", current_state, x)
       }
      default:
       // nothing in channel
       fmt.Printf("automata_Z: %s: %v: Waiting to receive.\n", current_state, x)
      }
    }
    if true {
      fmt.Printf("automata_Z: %s: %v: Checking z1.\n", current_state, x)
      // check if receive is available
      select {
      case receive, ok := <-channel_Q_Z_pong:
       if ok {
        // received
        _ = receive
        // next state
        current_state = "z0"

        fmt.Printf("automata_Z: %s: %v: Left z1.\n", current_state, x)
        continue
       } else {
        // channel closed
        fmt.Printf("automata_Z: %s: %v: ERROR: channel not open.\n", current_state, x)
       }
      default:
       // nothing in channel
       fmt.Printf("automata_Z: %s: %v: Waiting to receive.\n", current_state, x)
      }
    }
   }
   // check if end state has been reached
   if current_state == "z2" {
    fmt.Printf("automata_Z: %s: %v: End state Reached.\n", current_state, x)
    Z_fin = true
    break
   }
  }
 }
}
```

5.3.3 Server Client

This is an automatically generated from the CTA notation shown in Figure 15, implementing the CTA model shown in Figure 14.

```go
package main

import (
  "time"
  "fmt"
)

/* for the automata:
  Cta S = Init s0;s0 q?req(x >= 1) s1;s1 q!timeout(x > 3) s2;s1 q?ping(x <= 3,{x}) s3;s3 q!pong(x >= 1) s1;
  Cta Q = Init q0;q0 s!req(x >= 1) q1;q1 s?timeout(x >= 5) q2;q1 s!ping(x < 5,) q3;q3 s?pong(x > 1) q1;
*/

// speed of system
const clock_speed = 1
const clock_increment = 1

// set up clocks
var S_clock int = 0
var Q_clock int = 0

// set up notifiers
var S_fin bool = false
var Q_fin bool = false

func main() {
  // set up channels
  channel_Q_S_req := make(chan string, 2)
  channel_S_Q_timeout := make(chan string, 2)
  channel_Q_S_ping := make(chan string, 2)
  channel_S_Q_pong := make(chan string, 2)

  // run goroutines
  go automata_S(channel_Q_S_req, channel_S_Q_timeout, channel_Q_S_ping, channel_S_Q_pong)
  go automata_Q(channel_Q_S_req, channel_S_Q_timeout, channel_Q_S_ping, channel_S_Q_pong)

  // run clocks
  for {
    time.Sleep(time.Second * clock_speed)
    //increment clocks
    S_clock += clock_increment
    Q_clock += clock_increment
    // check if goroutines have ended
    if S_fin && Q_fin {
      break
    }
  }

  fmt.Println("CTA finished running.")
}

// Cta S = Init s0;s0 q?req(x >= 1) s1;s1 q!timeout(x > 3) s2;s1 q?ping(x <= 3,{x}) s3;s3 q!pong(x >= 1) s1;
func automata_S(channel_Q_S_req chan string, channel_S_Q_timeout chan string, channel_Q_S_ping chan string, channel_S_Q_pong chan string) {
  // initial state
  current_state := "s0"
  // set up clock
  x := S_clock
  fmt.Printf("automata_S: %s: %v: Starting...\n", current_state, x)
  // repeat until end state reached
  for {
    // update clock
    if x != S_clock {
      x = S_clock
      switch current_state {
      case "s0":
        if x >= 1 {
            fmt.Printf("automata_S: %s: %v: Checking s0.\n", current_state, x)
            // check if receive is available
            select {
            case receive, ok := <-channel_Q_S_req:
              if ok {
                // received
                _ = receive
                // next state
                current_state = "s1"

                fmt.Printf("automata_S: %s: %v: Left s0.\n", current_state, x)
                continue
              } else {
                // channel closed
                fmt.Printf("automata_S: %s: %v: ERROR: channel not open.\n", current_state, x)
              }
            default:
              // nothing in channel
              fmt.Printf("automata_S: %s: %v: Waiting to receive.\n", current_state, x)
            }
        }
      case "s1":
        if x > 3 {
          fmt.Printf("automata_S: %s: %v: Checking s1.\n", current_state, x)
          // send
          channel_S_Q_timeout <- "timeout"
          // next state
          current_state = "s2"
          fmt.Printf("automata_S: %s: %v: Left s1.\n", current_state, x)
          continue
        }
        if x <= 3 {
            fmt.Printf("automata_S: %s: %v: Checking s1.\n", current_state, x)
            // check if receive is available
            select {
            case receive, ok := <-channel_Q_S_ping:
              if ok {
                // received
                _ = receive
                // next state
                current_state = "s3"
                // reset x
                S_clock = 0

                fmt.Printf("automata_S: %s: %v: Left s1.\n", current_state, x)
                continue
              } else {
                // channel closed
                fmt.Printf("automata_S: %s: %v: ERROR: channel not open.\n", current_state, x)
              }
            default:
              // nothing in channel
              fmt.Printf("automata_S: %s: %v: Waiting to receive.\n", current_state, x)
            }
        }
      case "s3":
        if x >= 1 {
          fmt.Printf("automata_S: %s: %v: Checking s3.\n", current_state, x)
          // send
          channel_S_Q_pong <- "pong"
          // next state
          current_state = "s1"
          fmt.Printf("automata_S: %s: %v: Left s3.\n", current_state, x)
          continue
        }
      }
      // check if end state has been reached
      if current_state == "s2" {
        fmt.Printf("automata_S: %s: %v: End state Reached.\n", current_state, x)
        S_fin = true
        break
      }
    }
  }
}

// Cta Q = Init q0;q0 s!req(x >= 1) q1;q1 s?timeout(x >= 5) q2;q1 s!ping(x < 5,) q3;q3 s?
```

```
  pong(x > 1) q1;
func automata_Q(channel_Q_S_req chan string, channel_S_Q_timeout chan string,
  channel_Q_S_ping chan string, channel_S_Q_pong chan string) {
 // initial state
 current_state := "q0"
 // set up clock
 x := Q_clock
 fmt.Printf("automata_Q: %s: %v: Starting...\n", current_state, x)
 // repeat until end state reached
 for {
  // update clock
  if x != Q_clock {
   x = Q_clock
   switch current_state {
   case "q0":
    if x >= 1 {
     fmt.Printf("automata_Q: %s: %v: Checking q0.\n", current_state, x)
     // send
     channel_Q_S_req <- "req"
     // next state
     current_state = "q1"
     fmt.Printf("automata_Q: %s: %v: Left q0.\n", current_state, x)
     continue
    }
   case "q1":
    if x >= 5 {
      fmt.Printf("automata_Q: %s: %v: Checking q1.\n", current_state, x)
      // check if receive is available
      select {
      case receive, ok := <-channel_S_Q_timeout:
       if ok {
        // received
        _ = receive
        // next state
        current_state = "q2"

        fmt.Printf("automata_Q: %s: %v: Left q1.\n", current_state, x)
        continue
       } else {
        // channel closed
        fmt.Printf("automata_Q: %s: %v: ERROR: channel not open.\n",
  current_state, x)
       }
      default:
       // nothing in channel
       fmt.Printf("automata_Q: %s: %v: Waiting to receive.\n", current_state, x)
      }
    }
    if x < 5 {
     fmt.Printf("automata_Q: %s: %v: Checking q1.\n", current_state, x)
     // send
     channel_Q_S_ping <- "ping"
     // next state
     current_state = "q3"
     fmt.Printf("automata_Q: %s: %v: Left q1.\n", current_state, x)
     continue
    }
   case "q3":
    if x > 1 {
      fmt.Printf("automata_Q: %s: %v: Checking q3.\n", current_state, x)
      // check if receive is available
      select {
      case receive, ok := <-channel_S_Q_pong:
       if ok {
        // received
        _ = receive
        // next state
        current_state = "q1"

        fmt.Printf("automata_Q: %s: %v: Left q3.\n", current_state, x)
        continue
       } else {
        // channel closed
        fmt.Printf("automata_Q: %s: %v: ERROR: channel not open.\n", current_state, x)
       }
      default:
       // nothing in channel
       fmt.Printf("automata_Q: %s: %v: Waiting to receive.\n", current_state, x)
      }
    }
   }
   // check if end state has been reached
   if current_state == "q2" {
    fmt.Printf("automata_Q: %s: %v: End state Reached.\n", current_state, x)
    Q_fin = true
    break
   }
  }
 }
}
```

6 Acknowledgements

This project was supervised by Laura Bocchi, who has been helpful and supportive throughout. Even if my work isn't ultimately used, I hope it helps towards to development of the toolchain. This project has expanded the scope of what I have been able to learn this final year.