Cyber-Physical Computation

Last assignment

Maria Gabriela Jordão Oliveira[1*], pg50599
Miguel Caçador Peixoto[1†], pg50657

June 25, 2023

*mgabijo@gmail.com
†miguelpeixoto457@gmail.com
[1] Master in Physics Engineering, University of Minho, Gualtar, 4710-057 Braga, Portugal.

1 First task (managing shared resources with Uppaal)

For this first task we consider a small private airfield used by **2 planes**, which can be either **flying, parked, landing, or taking off**. The landing field is a resource shared by the two planes and the following requirements must be met:

1. Only 1 plane can use the field at a time.
2. A Controller component receives requests to land or to take off, and replies with a wait signal when the field is not available.
3. Each plane sends requests to the Controller to land or to take off, and sends notifications when the field becomes free.
4. The Controller has 5-time units to notify a plane to wait.
5. After 5-time units from requesting access to the field and with no wait signal, the planes take another 5-time units to reach the field.
6. Each plane takes non-deterministically between 1-3 time units to take off, and between 4-6 time units to land and park.
7. After taking off and parking, the planes notify the Controller with a gone signal.
8. If a plane is told to wait, it will take between 5-7 time units to reach the field.

Having this in consideration the following two templates for the Plane and Controller were designed:

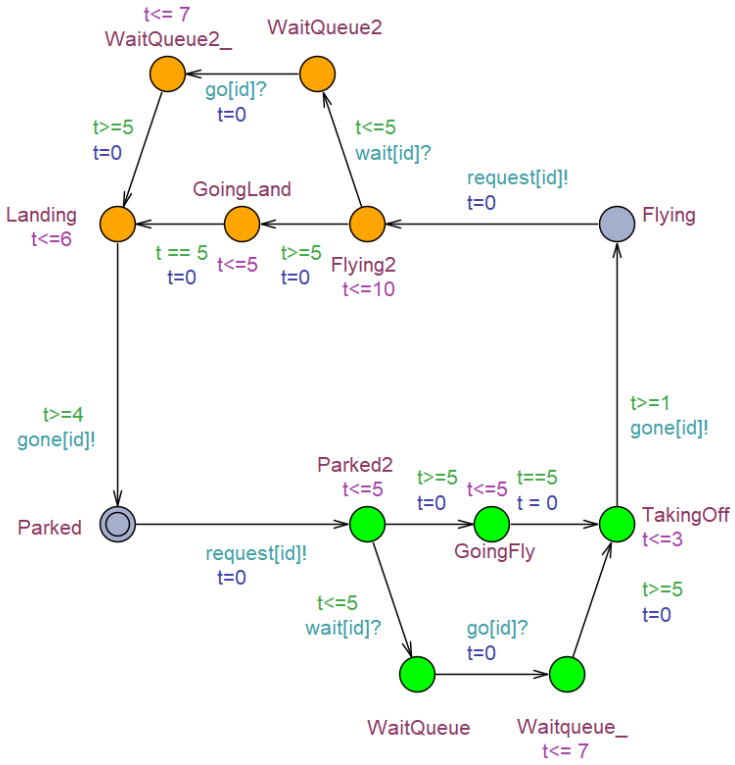

Figure 1: Airplane template architecture.

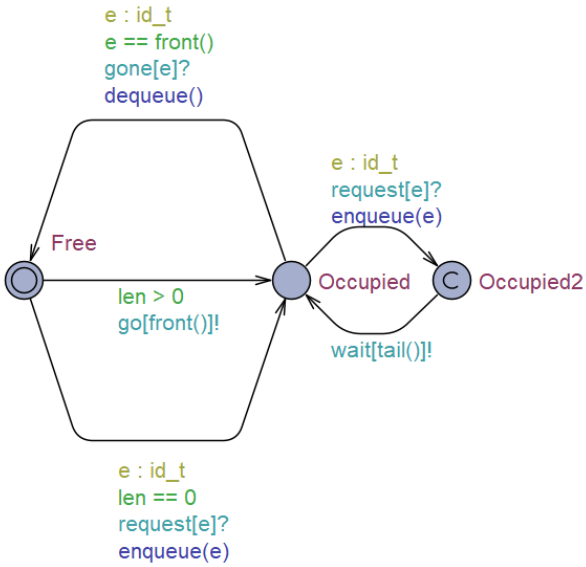

Figure 2: Controller template architecture.

To support the implementation of these templates, global variables were declared using the following code:

```
const int N = 5;
typedef int[0,N-1] id_t;
chan request[N], wait[N], gone[N];
urgent chan go[N];
```

Listing 1: Global Declarations.

The constant N represents the size of an array, for our case the number of planes. id_t is an integer array with elements ranging from 0 to $N-1$, representing the id of each plane. The channels "request," "wait," "gone," and "go" are additionally declared to facilitate communication between planes and the runway. These channels allow planes to request runway usage, confirm runway clearance, and receive instructions from the controller regarding whether they should wait or proceed.

Furthermore, the Controller system requires additional functions and variables, which are defined as follows:

```
// Declaration of an array 'list' of size N+1 to store elements
id_t list[N+1];

// Declaration of an integer array 'len' of size N+1 to track
// the length of the queue

int[0,N] len;
```

```
// Function to add an element to the end of the queue
void enqueue(id_t element)
{
// Assign the 'element' to the current length index of 'list' and
// increment 'len'
list[len++] = element;
}

// Function to remove the front element of the queue
void dequeue()
{
    // Declare and initialize a variable 'i' to 0
    int i = 0;
    // Decrement 'len' to reflect the removal of the front element
    len -= 1;

    // Shift each element towards the front by one position
    while (i < len)
    {
            list[i] = list[i + 1];
            i++;
    }

    // Set the last element of 'list' to 0
    list[i] = 0;
    }

// Function to return the front element of the queue
id_t front()
{
return list[0];
}

// Function to return the last element of the queue
id_t tail()
{
return list[len - 1];
}
```

Listing 2: Controller definitions.

In the above code, the array list and integer array len are used to implement a queue data structure. The function enqueue adds an element to the end of the queue, dequeue removes the front element of the queue, front returns the front element, and tail returns the last element. Note that the Plane system only declares the variable t, representing the

clock.

1.1 How it works

Each airplane is assigned a unique identifier (id) to ensure easy identification by the controller. Let's explore the functioning of each template individually.

1.1.1 Airplane

An airplane operates has two main locations: 'Parked' and 'Flying', where it can stay for an indeterminate time. In addition, there are "transitional" locations represented by colors (refer to Figure 1). These "transitional" locations determine whether the runway is available, enabling the airplane to switch between 'Parked' and 'Flying' locations. The transition from 'Parked' to 'Flying' involves passing through the 'Landing' location, while the transition from 'Flying' to 'Parked' involves passing through the 'TakingOff' location. Note that although they are different locations in our model, in the physical system they share the same landing field.

Since both transitioning systems work similarly, with only slight differences in timing, we will explain the process involved in taking off for simplicity.

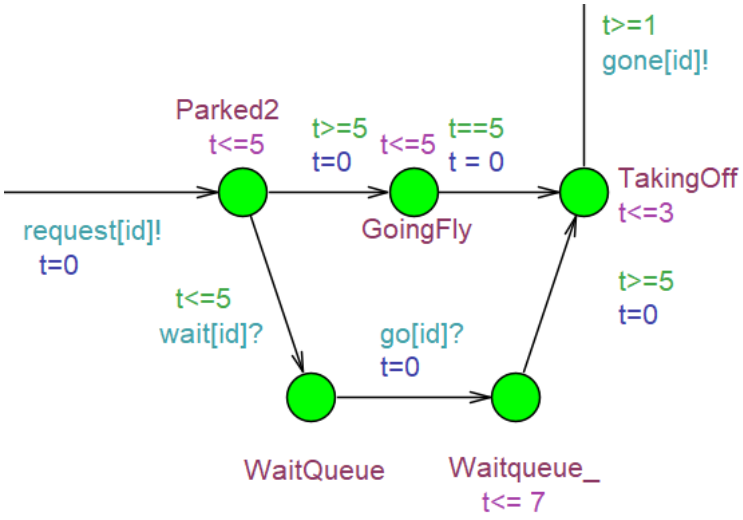

Figure 3: Taking off subsystem.

When an airplane wants to transition from the 'Parked' to the 'Flying' location, it should request clearance from the controller to take off, initiating the subsystem shown in Figure 3 with a 'request[id]' - a request tagged with the airplane's ID.

Next, one of two things can happen, as specified in the exercise:

1. The controller does not respond to the request, indicating that the runway is free. In this case, after 5 time units with no response, the airplane starts moving towards the airfield, taking an additional 5 time units, as depicted in the top row of nodes in Figure 3.

2. The controller sends a 'Wait' signal, moving the airplane to the 'WaitQueue' where it will wait until the controller sends the 'Go' signal. After receiving the signal, the airplane takes between 5-7 time units to reach the airfield, as shown in the lower row of nodes in Figure 3.

Once the airplane reaches the airfield, it enters the 'TakeOff' location, where it takes a non-deterministic amount of time, between 1-3 time units, to take off[1]. After taking off, the airplane transitions to the 'Flying' location and emit the 'gone' signal to the controller, indicating that it has cleared the runway from its perspective.

1.1.2 Controller

The controller, illustrated in Figure 2, operates based on two primary locations: "Free" and "Occupied," indicating whether the runway is available or in use, respectively. The objective of the controller is to handle the requests from planes. If the runway is occupied, it responds with a "wait" signal, placing the planes in a wait queue, as explained in the previous section. Once the runway is clear, the controller releases the planes from the queue by sending a "go" signal. To achieve this, the controller utilizes the functions described in Listing 2, which allow for enqueuing and dequeuing elements from a queue. Additionally, there are auxiliary functions available, such as "front" (returning the first element), "tail" (returning the last element), and "len" (providing the length of the list). It's important to note that the queue operates on a first-in, first-out (FIFO) basis.

Let's examine the step-by-step operation of the controller. Initially, the runway is free and in the "Free" location, prepared to receive liftoff/landing requests. If a request, denoted as "request[e]?", is received with an empty queue (i.e., "len == 0"), the controller transitions to the "Occupied" location and enqueues the plane's ID, effectively accepting the runway usage request. Since no further response is generated by the controller, the plane assumes it is clear to take off after 5-time units and proceeds accordingly, as described in the previous section.

Once the plane takes off and transitions to the "Flying" location, it sends a "Gone[e]" signal. If the plane's ID was the last one inserted in the queue (i.e., "e == front()"), the controller removes the ID from the queue using the "dequeue()" function and returns to the "Free" location. The guard condition "e == front()" ensures that only planes currently using the runway can be dequeued when leaving.

[1] In the 'sister' subsystem, instead of taking off, the airplane would land, taking between 4-6 time units to land and park.

It's important to consider two additional scenarios. If a plane submits a request while the runway is in use (in the "Occupied" location), the controller will enqueue the plane's ID, placing it in the last position of the queue, and instruct it to wait using the "wait[tail()]!" signal (remember that "tail" returns the index of the last element inserted in the queue). When instructed to wait, the plane will subsequently require a "go" signal to leave the wait queue. This can occur once the runway is cleared, and the controller transitions to the "Free" location. If there are planes in the queue (i.e., "len ¿ 0"), the controller can transition to the "Occupied" location and synchronize the "go[front()]!" signal, allowing the plane at the front of the queue to use the runway.

Note that the 'e' variable used for indexing is originated from the select statement of the particular transition (e : id_t), where "e" represents a variable of type "id_t".

1.2 Properties Tested

For ensuring the airfield operates normally, some properties were verified in Uppal:

1. Mutual Exclusion: Only one plane can use the landing field at a time.

   A□ forall (i : id_t) forall (j : id_t) (i != j) imply not
   ((Plane(i).Landing | Plane(i).TakingOff) && (Plane(j).Landing | Plane(j).TakingOff))

2. No deadlocks: We can not observe any deadlocks in our system

   E♢ not deadlock

   Even though the system appears to have no deadlocks, it can still come to a halt. For example, it can stay indefinitely on the 'Parked' or 'Flying' positions and so the expressions:

   $$\text{Plane(1).Parked} --> \text{Plane(1).Flying}$$
   $$\text{Plane(1).Flying} --> \text{Plane(1).Parked}$$

   return as being False on UPPAL's verifier, which is expected due to the specifications of the program. There is also another edge case that should be tested. When the place is told to wait it remains indefinitely on the 'WaitQueue' location until is told it can proceed via 'go!'. One should verify if eventually it will always be told to go via the following expressions:

   $$\text{Plane(1).WaitQueue} --> \text{Plane(1).Flying}$$
   $$\text{Plane(1).WaitQueue2} --> \text{Plane(1).Parked}$$

returning as True in on UPPAL's verifier. This was also expected due to the FIFO nature of the queue, and so it will eventually be processed by the controller.

3. All locations are accessible at least once during a trace. Due to no deadlocks in our system, we can limit the search to the paths where are multiple possible outcomes:

$$\begin{gathered} \text{E}\Diamond \text{ Plane(1).GoingLand} \\ \text{E}\Diamond \text{ Plane(1).WaitQueue2} \\ \text{E}\Diamond \text{ Plane(1).GoingFly} \\ \text{E}\Diamond \text{ Plane(1).WaitQueue} \\ \text{E}\Diamond \text{ Controller.Occupied} \\ \text{E}\Diamond \text{ Controller.Occupied2} \end{gathered}$$

Note: $p --> p$ means $A\Box$ *(p imply* $A\Diamond$ *q)*

Further Testing

In this section, we will conduct additional testing from a theoretical perspective. Let's consider a scenario to determine if an airplane can get stuck in the 'WaitQueue' location. For simplicity, we will assume the presence of two airplanes: P_1 and P_2.

Let's run a simple simulation. Airplane P_1 requests to use the runway for takeoff. If it does not receive a signal from the controller, it will assume the runway is clear and proceed with takeoff, thereby putting the controller in the 'Occupied' location. As airplane P_2 also wants to take off and the runway is already in use, the controller will respond to its initial request with a 'Wait' signal.

At this point, the controller's queue will be as follows:

$$\text{queue} = [P_1, P_2]$$

Here, the first airplane, P_1, made the initial request and is currently taking off, so it occupies the first position in the queue. P_2 is in the waiting queue.

Since P_2 cannot leave the waiting queue without a 'go' signal, and the 'go' signal can only be emitted from the 'Free' location of the controller, the controller must wait for the 'gone' signal from P_1. This signal will eventually occur within a specific time range required for takeoff (1 to 3 time units). After receiving the 'gone' signal, the controller will transition to the 'Free' location, dequeue P_1 from the queue, and allow P_2 to proceed. Note this process can't be interrupted, since when the controller is in the 'Free' location it can only accept requests if the queue is empty, and when in the 'Occupied' location, if any request is received, the controller immediately sends a 'wait' signal.

With this we covered the possible paths the two planes could have taken, proving that the system can't be 'stuck' in the waiting queue for two planes. But what if for an arbitrary number of planes, N? If so, the queue would look like the following:

$$\text{queue} = [P_1, P_2, P_3, \ldots, P_N]$$

Here, P_1 represents the plane that is currently transitioning from the ground to the air (or vice versa) due to the inherent properties of the system discussed in previous sections. The remaining planes, P_2 to P_N, are in the waiting queue.

Given that the controller is currently in the 'Occupied' location, any new requests will simply add more planes to the queue. The only other feasible action for the controller is to release the previous plane, denoted as P_1, by dequeuing it from the queue. Subsequently, the controller will continue releasing planes from the front of the queue, which were initially on the waiting list, and wait for confirmation of their departure before repeating the process. If additional requests are received to use the runway, they will result in new airplanes entering the 'WaitQueue' location and being added to the bottom of the queue for later processing using the same process. This cycle will continue until the queue is empty.

This thought experiment serves as a concept proof that the planes in the 'WaitQueue' location cannot be indefinitely stuck in that location, being eventually moved to a new location via the 'go' signal from the controller.

Extra - n Planes

The number of planes was inherent and a parameter of the program shown above. Adjusting the number of planes to 6, all the above expressions were successfully verified in UPPAL verifier.

1.3 Airfield additional features

Although this system adheres to the requirements and specifications demanded, it's a very simple model that would work only for a few small airfields. To make a more realistic model, it would be necessary to include some other features. In particular,

- **More landing fields** - A more complex airfield may have more than one landing field, so, it's necessary to manage all of the landing fields. As far as the implementation is concerned, our first approach would be having a variable counting the number of requests and it would be possible to let use the field as many airplanes as number of fields. This variable would be incremented with any request and decremented with any 'gone' signal. Hence, the wait messages would only be sent when the number of airplanes reached the number of fields.

- **Priority flights** - In the airfield implemented all the airplanes had the same priority, although, in real scenarios, not all airplanes have the same priority in using the field. Thus, the first approach to include priority airplanes in the model would be to have only two different degrees of priority (normal and high). Then, for each level of priority would be a different queue of airplanes, and the planes in the normal queue

only receive a 'go' signal if the priority queue is empty. This means that high-priority airplanes would use the airfield with the desired priority. For this to work would be necessary to have an array with the high-priority airplanes' id, to en-queue the planes in the correct queue.

- **Not allowed flights** - In contrast with the previous point, for airfield security reasons, it would be advantageous to have a list of forbidden airplanes, and, if the controller received a request from one of these airplanes they would receive a response denying the field access. In terms of implementation, this would necessitate an array with the id of the forbidden airplanes and add some actions to our system. Namely, after sending the request, airplanes, in locations 'Parked2' and 'Flying2', can receive a 'back' signal that would send them back to the previous states, i.e. 'Parked' and 'Flying', respectively. Similarly, after each request controller should check the plane's id and, if it is a forbidden one, it would be necessary to send a 'back' signal.

- **Emergency protocol** - Again, for security reasons, having a security protocol would bring benefits. Simply, the security protocol could be having an 'emergency' action that would be emitted under specific conditions and all the airplanes not using the field would be sent back (to Parked or Flying locations depending on their actual location). To address this issue a third entity can be added - this entity should have only two locations 'regular' and 'emergency', the 'regular' location should have an invariant accordingly to the security protocol, and the 'emergency' should be an urgent one, only to be possible to send the airplanes back. In concordance, the airplane's template would need to include the 'emergency' action to be possible to send them back.

2 Second task (essay on using the right concepts in software development)

In this second task, one is required to write a small essay detailing the differences between modelling, verification, and programming. It is also demanded to discuss how they complement each other.

2.1 Introduction

Appearing from the synergies between computer science, engineering, and the physical world, cyber-physical computation is focused on the integration of digital systems and physical processes. Hence, it involves the modelling, verification, programming, and deployment of computational systems that interact with and/or control physical entities in the real world. These physical entities include vehicles, robots, industrial machinery, medical devices, energy systems, etc.

In a more general, such as happens for the particular case of cyber-physical computation, when developing software, three key concepts play distinct yet interconnected roles, i.e. modelling, verification, and programming. These three concepts take part in the process of developing software and each one has its own purpose and contributes to the overall objective. With this, in this essay, these three different concepts are explored (subsection 2.2, subsection 2.4 and subsection 2.5), then their interplaying and complementary are discussed in subsection 2.5, and, finally, in subsection 2.6 some running examples are given.

2.2 Modelling

The modelling process, in the context of software development, essentially consists of creating abstract representations of the system in study. During this process, the essential aspects, structures, and behaviours of the studied system are captured. The main purposes of this part are to gain sensibility and deeper understanding of the studied system, clarify the requirements of the system, and facilitate communication between intervening in the software development process.

Accordingly, serving as means of documentation and communication that enables the visualisation and analysis of the system (that may be complex), models can have different forms, such as diagrams, charts, automata, mathematical formulas, and so on. Inherently, modelling may have a crucial role in managing of problem's complexity by breaking it down into simpler components. Thus, it works like a foundation for other activities (verification and implementation).

In modelling a system process, one can stand out at least three main components to modelling. Videlicet, structural modelling - modelling the static structure of the system through the identification of the components, relationships, and dependencies between different elements of the system -, behavioural modelling - capturing the dynamic behaviour of the software system, focusing on how the system responds to various inputs, events, and/or stimuli, it also describes the sequence of actions, states, and interactions between different components -, and, data modelling - defining the structure, relationships, and constraints of the data that the software system will manipulate and store.

2.3 Verification

The second concept mentioned is verification, it is essentially focused on verifying if a system implementation (or design) meets the specified requirements and if it behaves as desired. Hence, this stage aims to identify defects or deviations from the expected behaviour or non-compliments with quality standards and regulations, to rectify them in the early development process. In other words, the verification process contributes to improving the overall quality of the system (software) and reducing the risks associated with software failures, such as undesired infinite loops.

The verification techniques usually used include formal methods - use of mathematical techniques (such as model checking or theorem proving) to formally verify the correctness or properties of software -, static analysis - analyse of the software's code without executing the program to detect potential defects, vulnerabilities, or violations of coding standards -, testing - executing the software with various inputs and assessing its outputs against expected results, techniques as unit testing, integration testing or system testing are commonly employed -, etc.

2.4 Programming

The last but not least concept is programming. When one thinks of software development this is probably the first concept that comes to mind and, as expected, it essentially consists of writing code to implement a desired functionality. From another perspective, programming is considered the process of translating the design and requirements captured through modelling into executable instructions. The development of robust and scalable systems is directly related to the programming practices adopted.

The programming stage encompasses several aspects. Namely, choosing a programming language according to the systems requirements and constraints and the developer's taste, writing the code itself (and most of the time with special attention to making it reusable and understandable), managing data (code may need to store, retrieve, transform, and process data according to the requirements), implement error handling mechanisms - to handle exceptions, validate input, and if possible recover from failures, such techniques help to improve code stability and reliability -, etc.

2.5 Modelling, verification and programming interplaying

So far, modelling, verification, and programming were presented as distinct activities in the process of software development with brief references to one another. Besides their natural distinction, they are closely interconnected and have complementary roles in several ways.

In the early stages of software development, modelling plays a major role by making the bridge between conceptual understanding and practical implementation, allowing developers to gain insights into the system's structure and behaviour before starting to code. Furthermore, as already referred, modelling reduces ambiguities and facilitates communication between the intervenings.

Following, the verification phase tries to ensure the system behaves as intended and meets the specified requirements through the application of several techniques, as already mentioned. The verification stage is more effective due to the representation of the system's structure, behaviour, and requirements made during modelling. In other words, the modelling stage generates models that serve as a basis for formal methods, testing, inspections, and reviews made during the verification process.

Finally, programming merges the models created and verification through the implementation of the system, i.e. the code produced during programming serves as the concrete realization of the models (that probably already passed through verification processes). The models give a high level of the system, guiding the developers in the process of programming. Besides the role in improving the model, verification also plays a crucial role in ensuring the correctness and quality of the implemented code, through the identification of defects, bugs, or deviations from the desired behaviour in code and supports the reliability, functionality, and performance of the implemented system.

To sum up, modelling, verification, and programming are distinct parts of software development that have interplaying roles. Together, they form a cohesive process with the one objective of creating a successful software solution.

2.6 Study cases

After presenting the concepts and their roles, let's look at some concrete examples. Well, as one may imagine this kind of process is present in the all of quotidian things, from an e-commerce website to a self-driving car system or a traffic signal system. Here, it was chosen to present the development of an autonomous drone system and a traffic signal system.

2.6.1 Autonomous drone system

Let's look now at an example of how to develop software for an autonomous drone system, with special emphasis on the modelling, verification, and programming stages.

As far as the modelling process is concerned, it is necessary to create a model representing different parts of the system. Namely, the physical components (e.g. helices), sensors (e.g. proximity, wind, etc), control algorithms, communication protocols (e.g. to communicate with the controller), and any other desired part. This model can then be used to simulate the drone's behaviour, analyse its performance, and/or refine the system's design.

After modelling, to verify the model and some desired properties, formal methods, and simulation-based techniques can be used, for example. In terms of properties to verify, verification can include, for example, checking the system against the predefined specifications and ensuring its safety by correctly responding to different environmental conditions or avoiding obstacles.

Finally, after designing the model as desired and making some verification on it, it's necessary to implement it through code writing. During the process, some other verification may need to be made in order to avoid code problems or bugs.

2.6.2 Traffic Signal System

As expected, the development of a traffic signal system also includes the three exposed concepts, i.e. modelling, verification and programming.

To create the model of a traffic signal system, it is necessary to include the traffic lights themselves, some sensors (such as radars or proximity ones) and time mechanisms to control, for example, the natural flow of light signals.

Then, as usual, it's necessary to use verification methods (e.g. formal methods) to ensure the system behaves as expected. As far as the behaviour of a traffic signal system is concerned, one wants to avoid car collisions, minimise congestion, and probably minimise energy consumption.

Finally, attending to the model and the outcomes of the verification process, it is necessary to write the code to implement this system. A correct programming stage and an efficient verification enable the implementation of traffic light software the ensure road safety.

3 Third task (unified program semantics)

In this task, it is asked to develop a probabilistic language in the same spirit as the languages developed in the previous lectures/assignments. The first step in this process is to present its semantics attending to the given grammar and then implement it in Haskell using the monad of probabilities. So, the starting point is the following grammar:

$$\text{Prog}(X) \ni x := t |\ p +_p q \mid p; q \mid \text{if } b \text{ then } p \text{ else } q \mid \text{while } b \text{ do } \{p\}$$

Accordingly to the semantics exposed during lectures, in this case, the semantic is the following one:

$$\frac{\langle p, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i \qquad \forall_{i\leq n}\langle q, \sigma_i\rangle \Downarrow \mu_i}{\langle p; q, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \mu_i}\ (\text{seq})$$

$$\frac{\langle t, \sigma\rangle \Downarrow r}{\langle x := t, \sigma\rangle \Downarrow \sigma[r/x]}\ (\text{asg})$$

$$\frac{\langle p, \sigma\rangle \Downarrow \mu \qquad \langle q, \sigma\rangle \Downarrow \mu'}{\langle p +_p q, \sigma\rangle \Downarrow \mu \cdot p + \mu' \cdot (1-p)}\ (\text{prob})$$

$$\frac{\langle b, \sigma\rangle \Downarrow tt \qquad \langle p, \sigma\rangle \Downarrow \mu}{\langle \text{if b then p else q}, \sigma\rangle \Downarrow \mu}\ (\text{if}_1) \qquad \frac{\langle b, \sigma\rangle \Downarrow ff \qquad \langle q, \sigma\rangle \Downarrow \mu'}{\langle \text{if b then p else q}, \sigma\rangle \Downarrow \mu'}\ (\text{if}_2)$$

$$\frac{\langle b, \sigma\rangle \Downarrow tt \qquad \langle p, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i \qquad \forall_{i\leq n}\langle \text{while b do p}, \sigma_i\rangle \Downarrow \mu_i'}{\langle \text{while b do p}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \mu_i}\ (\text{wh}_1)$$

$$\frac{\langle b, \sigma\rangle \Downarrow ff}{\langle \text{while b do p}, \sigma\rangle \Downarrow \sigma}\ (\text{wh}_2)$$

Where $\mu = \sum_i^n p_i \cdot \sigma_i$ is a distribution.

After defining the semantics, just as in the previous assignment, to implement the language and its semantics in Haskell its necessary to define three essential things:

- Types - It is necessary to define the possible types of programs in our language. In the implementation below the program's type was nominated as ProgTerm and admits the statements presented above.
- Semantic - It is also necessary to define the semantic rules, just as theoretically defined. This definition is implemented through the function psem.
- Change memory function - Since the assign statement enables changing the memory (environment) function, it's necessary to implement a function that does that (function chMem in the code). Since this function is the same used in the previous assignment and was already presented, although it is implemented in the code file submitted along with this assignment, its implementation it's not presented in this file.

Attending to these semantic rules, and using the code developed in the previous lectures/assignment for the semantics of Linear and Boolean terms, the implementation of the probabilistic terms is the following.

```
-- Defining the type of ProgTerms
-- ProgTerms can be either an Asg, a Seq, an Ife, a Wh or a Prob
-- Asg is the assignment of a LTerm to a variable
-- Seq is the sequential composition of two ProgTerms
-- Ife is the if-then-else statement
-- Wh is the while statement
-- Prob is the probabilistic statement
data ProgTerm = Asg Vars LTerm | Seq ProgTerm ProgTerm | Ife BTerm ProgTerm
    ProgTerm | Wh BTerm ProgTerm | Prob ProgTerm Float ProgTerm deriving (Show, Eq
    , Ord)

psem :: ProgTerm → (Vars → Double) → Dist (Vars→ Double)
-- Asg x t returns the memory function with the value of x changed to t
psem (Asg x t) m = return (chMem x (sem t m) m)

-- Seq p q returns the memory function resulting from executing q after p after p
     it is generated a distibution of memory functions, which is used to execute q
psem (Seq p q) m = do m' ← psem p m
                      psem q m'
```

```
-- Ife b p q returns the memory distribution resulting from executing p if the
    semantics of b is True, otherwise it returns the memory distribution resulting
     from executing q
psem (Ife b p q) m = let v = bsem b m in if v then psem p m else psem q m

-- Wh b p returns the memory distribution resulting from executing p and Wh b p,
    if the semantics of b is True, otherwise it returns the memory distribution m
psem (Wh b p) m = let v = bsem b m in if v then psem (Seq p (Wh b p)) m else
    return m

-- Prob p r q returns the memory function resulting from executing p with
    probability r and q with probability 1-r
psem (Prob p r q) m = do m' ← psem p m
                         m'' ← psem q m
                         choose r m' m''
```

Listing 3: Haskell code to implement the probabilistic language studied.

As one may see, this implementation is self-explained by its comments and the theoretical semantics presented.

Examples

Now, after the implementation stage, it's time to expose some examples that show that the implementation behaves as expected. The examples below are also in the code file. First, let's clarify that there are three different vars implemented in the code file, although it is possible to implement as many vars as you wish, it's only necessary to change the declaration and the sigma function. The memory function defined is the following:

```
ghci> sigma
X = 0.0, Y = 1.0, Z = -100.0
```

Note that the sigma function is shown in this readable way because it was implemented a function that shows things of type $Vars \rightarrow Double$.

Now defining x, y and z as these vars, accordingly to the semantic rules:

```
x = Leaf (Left X)
y = Leaf (Left Y)
z = Leaf (Left Z)
```

The test concerning the semantics of Linear and Boolean terms was already presented in the previous assignment, thus it's not necessary nor important to repeat them here. The first example concerning the probabilistic language doesn't include the prob statement itself, so the goal is to compute *while* $x \leq x$ *do* $x=x+y$; $y=y+1$. To do this it's necessary to implement this function accordingly to the defined notation and then compute its semantic attending to sigma, this whole process is implemented and commented below.

```
-- wh x ≤ x do x=x+y; y=y+1
-- x≤x -- BTerm
lexx = Leq x x
-- x=x+y -- ProgTerm
xMaisy = (Asg X (Plus x y))
-- y=y+1
yMaisUm = Asg Y (Plus y (Leaf (Right 1)))
-- wh x ≤ x do x=x+y; y=y+1
teste = Wh lexx (Seq xMaisy yMaisUm)

ghci> psem teste sigma
```

As you can see, since x is always equal to itself the while doesn't finish, and so our evaluation doesn't finish too. To exposed the probabilistic effect let's use it with the terms *xMaisy* and *yMaisUm* defined above, i.e.

```
-- Prob xMaisy 0.5 yMaisUM
teste_p = Prob xMaisy 0.5 yMaisUm

ghci> psem teste_p sigma
X = 0.0, Y = 2.0, Z = -100.0  50.0%
X = 1.0, Y = 1.0, Z = -100.0  50.0%
```

As one may trivially see, the result is the expected, since it's obtained the addition of 1 to Y with 50% of probability and, with the same probability, the addition of Y to X. To finish, let's look at a more complex example that uses both if and probabilistic statements. In other words, we want to compute: *if* $x \leq y$ *then* $(x = x + 1) +_{0.2} (x = 30)$ *else* $(y = y + 1) +_{0.6} (y = 40)$.

```
-- if x ≤ y then Prob x = x + 1 0.2 x = 30 else Prob y = y + 1 0.6 y = 40

teste2 = Ife (Leq x y) (Prob (Asg X (Plus x (Leaf (Right 1)))) 0.2 (Asg X (Leaf (
    Right 30)))) (Prob (Asg Y (Plus y (Leaf (Right 1)))) 0.6 (Asg Y (Leaf (Right
    40))))

ghci> psem teste2 sigma
X = 30.0, Y = 1.0, Z = -100.0  80.0%
X = 1.0, Y = 1.0, Z = -100.0  20.0%
```

It was obtained that the value of X is changed to 30 with a probability of 80% and to 1 with a 20% probability. This result is the one desired since X is less than Y, and so, it was computed $(x = x + 1) +_{0.2} (x = 30)$, that it is exactly the output obtained.

3.1 Extra

For extending the semantics further, the functionality of handling exceptions was added to *psem* in accordance to the following semantic. It should be created an exception if the

probability used in a probability statement is less than zero or more than 1.

$$\frac{\langle t, \sigma\rangle \Downarrow r}{\langle x := t, \sigma\rangle \Downarrow \sigma[r/x]}\ (\text{asg})$$

$$\frac{\langle p, \sigma\rangle \Downarrow \sum_i^n p_i \cdot (\sigma_i + \{e\}) \qquad \forall_{i \le n} \langle q, \sigma_i\rangle \Downarrow \sum_j^n p'_j \cdot (\sigma_{ij} + \{e\})}{\langle p; q, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sum_j^n p'_j \cdot (\sigma_{ij} + \{e\})}\ (\text{seq})$$

$$\frac{\langle b, \sigma\rangle \Downarrow tt \qquad \langle p, \sigma\rangle \Downarrow \sum_i^n p_i \cdot (\sigma_i + \{e\})}{\langle \text{if b then p else q}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot (\sigma_i + \{e\})}\ (\text{if}_1)$$

$$\frac{\langle b, \sigma\rangle \Downarrow ff \qquad \langle q, \sigma\rangle \Downarrow \sum_i^n p'_i \cdot (\sigma'_i + \{e\})}{\langle \text{if b then p else q}, \sigma\rangle \Downarrow \sum_i^n p'_i \cdot (\sigma'_i + \{e\})}\ (\text{if}_2)$$

$$\frac{\langle b, \sigma\rangle \Downarrow tt \qquad \langle p, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i + \{e\} \qquad \forall_{i \le n} \langle \text{while b do p}, \sigma_i\rangle \Downarrow \mu'_i + \{e\}}{\langle \text{while b do p}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \mu_i + \{e\}}\ (\text{wh}_1)$$

$$\frac{\langle b, \sigma\rangle \Downarrow ff}{\langle \text{while b do p}, \sigma\rangle \Downarrow \sigma}\ (\text{wh}_2)$$

$$\frac{0 \le r \le 1 \qquad \langle p, \sigma\rangle \Downarrow \sum_i^n p_i \cdot (\sigma_i + \{e\}) \qquad \langle q, \sigma\rangle \Downarrow \sum_i^n p'_i \cdot (\sigma'_i + \{e\})}{\langle p +_r q, \sigma\rangle \Downarrow \sum_i^n p_i \cdot (\sigma_i + \{e\}) \cdot p + \sum_j^n p'_j \cdot \Big(\sigma'_j + \{e\}\Big) \cdot (1 - p)}\ (\text{prob}_1)$$

$$\frac{r < 0 \vee r > 1 \qquad \langle p, \sigma\rangle \Downarrow \mu' \qquad \langle q, \sigma\rangle \Downarrow \mu''}{\langle \langle p +_r q, \sigma\rangle \Downarrow \{e\}}\ (\text{prob}_2)$$

In this program, the addition of exception handling was added via The **Maybe** monad, which can either return a value wrapped in **Just** or indicate an exceptional case using **Nothing** (the exception).

The `psem` function, like in the previous section, takes a `ProgTerm` expression, a function `(Vars -> Double)` representing the memory, and returns a distribution (Dist monad) of values that can either be memory functions or exceptions (Maybe monad).

To evaluate an assignment statement (`Asg x t`), the function first evaluates the term `t` using the current memory state `m` by invoking `sem t m`. Then it updates the memory with the result by calling `chMem x (sem t m) m`. The updated memory is wrapped in `Just` and returned.

For a sequence of programs (`Seq p q`), the function recursively evaluates the first program `p` using the current memory state `m` by invoking `psem p m`. If the result is `Just m''`, indicating a successful evaluation, the function proceeds to evaluate the second program `q` with the updated memory state `m''` by invoking `psem q m''`. The final result is the result of evaluating `q`. If a `Nothing` is encountered during the process the exception is propagated.

When encountering an if-else statement (`Ife b p q`), the function evaluates the condition `b` by invoking `Just (bsem b m)` and stores the result in `v`. If `v` is `Just True`, it evaluates the first program `p` using the current memory state `m` by invoking `psem p m`. If `v` is `Just False`, it evaluates the second program `q` with the current memory state `m`. In case `v` is `Nothing`, indicating a failure during the evaluation of the condition, the function returns `Nothing`.

For a while loop (`Wh b p`), the function evaluates the condition `b` and stores the result wrapped by `Just` in `v`. If `v` is `Just True`, it evaluates the program `p` using the current memory state `m` by invoking `psem p m`. If the evaluation of `p` is `Just m''`, indicating a successful evaluation, the function continues the loop by recursively calling `psem` with the loop condition `b` and the program `Wh b p`, using the updated memory state `m''`. If `v` is `Just False`, the loop ends, and the function returns `Just m` without any modifications. If `v` is `Nothing`, indicating a exception during the evaluation of the condition, the function returns `Nothing`. An exception found during the evaluation of p is also propagated.

In the case of a probabilistic choice statement (`Prob p r q`), the function first checks if the probability `r` is within the valid range of 0 to 1. If it is not, the function returns `Nothing`. Otherwise, it evaluates both the true branch `p` and the false branch `q` using the current memory state `m` by invoking `psem` on both expressions. If both evaluations result in `Just` values (`Just mp'` and `Just mq'`), the function calls the `choose` function with the probability `r` and the two memory states `mp'` and `mq'`. The `choose` function selects one of the memory states based on the given probability and returns it. If either `mp` or `mq` is `Nothing`, the function returns `Nothing`.

The new code for the expanded semantics is below.

```
-- Define the semantics of ProgTerms using the MaybeT and Dist monads
psem :: ProgTerm → (Vars → Double) → Dist (Maybe (Vars → Double))

-- Evaluates an assignment statement by updating the memory with the result of
    evaluating the term.
psem (Asg x t) m = return (Just (chMem x (sem t m) m))

-- Evaluates a sequence of programs by executing the first program, updating the
    memory,
-- and then executing the second program with the updated memory.
psem (Seq p q) m = do
  m' ← psem p m
  case m' of
    Just m'' → psem q m''
    Nothing → return Nothing

-- Evaluates an if-else statement by evaluating the condition, and then executing
-- either the first program or the second program based on the condition's result
    .
psem (Ife b p q) m = do
```

```
  v ← return (Just (bsem b m))
  case v of
    Just True → psem p m
    Just False → psem q m
    Nothing → return Nothing

-- Evaluates a while loop by evaluating the condition, and if it's true,
-- executes the program inside the loop and repeats the process.
-- If the condition is false or the evaluation fails, the loop ends.
psem (Wh b p) m = do
  v ← return (Just (bsem b m))
  case v of
    Just True → do
      m' ← psem p m
      case m' of
        Just m'' → psem (Wh b p) m''
        Nothing → return Nothing
    Just False → return (Just m)
    Nothing → return Nothing

-- Evaluates a probabilistic choice statement by evaluating both the true and
-- false branches and choosing one based on the given probability.
-- The memory is passed to both branches.
psem (Prob p r q) m
  | r < 0 || r > 1 = return Nothing
  | otherwise = do
      mp ← psem p m
      mq ← psem q m
      case (mp, mq) of
        (Just mp', Just mq') → choose r mp mq
        _ → return Nothing
```

Examples

As before, some example code was tested to show that the implementation behaves as expected.

For exemplifying the newly implemented exception handling feature, a division function, 'div2', was created. This function outputs an exception every time a division 0 is encountered.

```
div2 :: Double → Double → Dist (Maybe Double)
div2 _ 0 = return Nothing
div2 x y = return (Just (x / y))
```

Accordingly, a function was also made to test it's behaviour:

```
testDiv :: Dist (Maybe Double)
```

```
testDiv = do
  x ← uniform [1, 2, 3]
  y ← uniform [0, 1, 2, 3]
  div2 x y
```

The expected behaviour is that 25% of the time we obtain Nothing since the y uniform distribution has a probability of $\frac{1}{4} = 0.25$ to obtain 0 (and thus we will try to divide by 0 25% of the time).

```
ghci> testDiv
                Nothing  25.0%
               Just 1.0  25.0%
Just 0.3333333333333333   8.3%
               Just 0.5   8.3%
Just 0.6666666666666666   8.3%
               Just 1.5   8.3%
               Just 2.0   8.3%
               Just 3.0   8.3%
```

Another functions were devised to test the other instances of the semantics. For example, testing the 'Prob':

```
testProbChoice :: ProgTerm
testProbChoice = Prob (Asg X (Leaf (Right 1))) 0.2 (Asg X (Leaf (Right 2)))
```

This yields assigning 1 with a probability of 0.2 and 2 with a probability of $(1 - 0.2)$ to X, as expected:

```
ghci> psem  testProbChoice sigma
Just  X = 2.0, Y = 1.0, Z = -100.0  80.0%
Just  X = 1.0, Y = 1.0, Z = -100.0  20.0%
```

The 'if' statement was also tested:

```
testIfElse :: ProgTerm
testIfElse = Ife  (Leq  (Leaf  (Right  10)) (Leaf  (Left X)) ) (Asg  Y (Leaf  (
    Right  1))) (Asg  Y (Leaf  (Right  0)))
```

This program should assign 1 to Y if $X > 10$, and 0 otherwise. Since $X \leq 10$, we should expect $Y = 0$:

```
ghci> psem testIfElse sigma
Just X = 0.0, Y = 1.0, Z = -100.0  100.0λ%
```

Since $Y = 1.0$, the code works as expected.

For testing the while loop, the following code was written:

```
testWhile :: ProgTerm
testWhile = Wh (Leq (Leaf (Left X)) (Leaf (Right 10)) ) (Asg X (Plus (Leaf (Left
    X)) (Leaf (Right 1))))
```

This function should increment X until it reaches 10 and then increment one more before failing evaluation of *(Leq (Leaf (Left X)) (Leaf (Right 10)))*. By testing it we obtain:

```
ghci> sigma
X = 0.0, Y = 1.0, Z = -100.0
ghci> psem testWhile sigma
Just X = 11.0, Y = 1.0, Z = -100.0 100.0%
```

As expected, $X = 11$ by the end of computation.