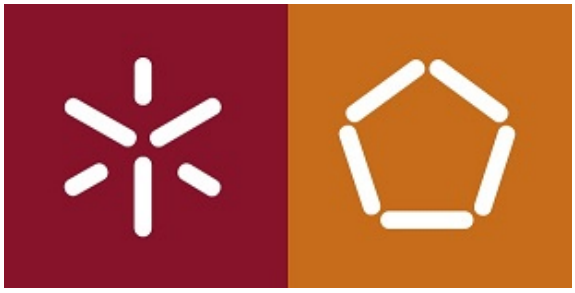

University of Minho
Informatics Department

Practical Assignment

Cyber-Physical Programming

June 26, 2023

Beatriz Rodrigues
(*pg50240*)

Gonçalo Rodrigues
(*pg50667*)

Contents

1. Initial overview

In the field of cyber-physical programming, the seamless integration of software and physical systems is becoming increasingly important. As part of our project in this field, we set off to complete three different tasks. First, we participated in the UPPAAL modelling exercise, where we modeled the operation of a small private airport in detail.

We then worked on an article that aimed to clarify the intrinsic differences between modelling and verification, while making a comparison between programming and the aforementioned aspects. By contrasting the distinct but interdependent roles of modelling, verification and programming, we gain insight into their importance in system development.

Finally, we turn our attention to an exercise that focuses on implementing a probabilistic language in the Haskell programming language. Using the probability monad, we build a powerful framework for expressing and evaluating probabilistic computations.

2. Managing shared resources with UPPAAL

In this particular task of the practical assignment, our objective was to undertake the implementation of an `UPPAAL` model that accurately represents the operations of a private airfield, managed by a controller. We strived to create an accurate representation of the airfield's operational behavior, encompassing the crucial restriction of guaranteeing single-plane occupancy during landing or take off, as well as other important restrictions made clear in the practical assignment guide.

2.1 Modelling

We will delve into the modelling process of a plane and a controller, explain their respective constraints and elucidate how they were implemented in the model itself. Furthermore, we elaborate on the way we ensured the ability to dynamically run the model with n planes, simultaneously.

2.1.1 Global variables

To start, the global variables defined for our model are presented below.

```
//No.  of planes
const int N = 4;

//Generate plane ids
typedef int [0, N-1] planeID;

// Used for indexes in appr matrix
const int TAKEOFF = 0;
const int LAND = 1;

// Generates channels
chan wait[N], gone, appr[2][N];
urgent chan go[N];
```

To account for N planes at the same time, we implemented a variable that can be adjusted to the desired number of planes. Furthermore, we specify that each plane is assigned an index in the range 0 to N-1.

To facilitate communication between the controller and the individual planes, we also define `wait` and `go` as arrays of N channels, where the second aforementioned one is urgent, in order to force both edges to be taken as soon as they are ready. However, it should be noted that the `gone` channel can be declared as a single channel in our implementation, as its specific details are explained in the following sections. Lastly, we implemented a matrix of channels, `appr`, since the controller will handle *take off* and *land* requests in the same way. This way, `appr[TAKEOFF][planeID]` will represent the *take_off* channel between the controller and a plane with *planeID*, while `appr[LAND][planeID]` will represent the *land* channel between the controller and that same plane.

2.1.2 Controller

In order to implement the controller, we needed to provide functions to manage a N sized list, that represents the queue of planes interested in using the field, as well as a variable that stores the state of the aforementioned resource.

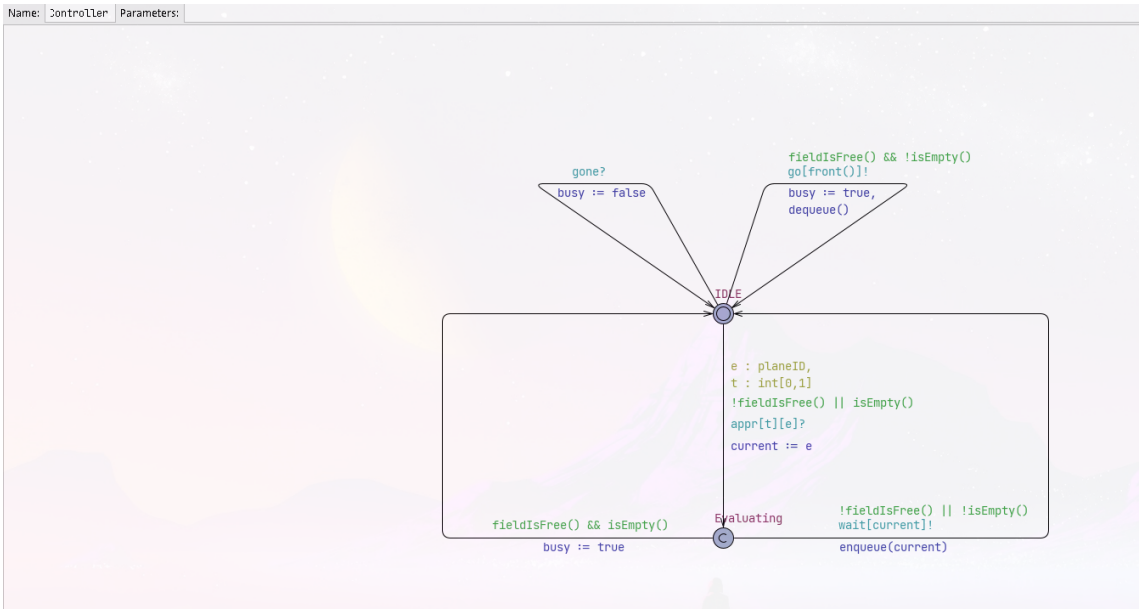

Figure 2.1: Controller

The controller is able to take an `appr` signal from one of the existing planes, moving to state `Evaluating` and deciding, in the next instant, if it should send back a wait signal or not. This decision is influenced by the field being occupied or there being planes waiting for their turn to use it. However, this transition can only be selected if the field is busy or there are no planes queued, in order to guarantee that it's impossible for all N planes to be in the queue at the same time.

When a plane notifies that it has left the landing field, with a `gone` signal, we set the field as free, updating the *busy* variable to *false*.

Finally, if the field is free and there are planes waiting to take off or land, we send a `go` signal to the first in line, so that he can go ahead and take on the desired action, removing it from the queue and setting the field as busy, again.

Local variables

```
// Current plane being handled
planeID current;

// Represents the state of the field as busy or not
bool busy = false;

// List that represents a N sized queue (at most), with planes interested in
    using the field
planeID queue[N];

// Queue size
int[0,N] len = 0;
```

```
// Returns if the field is free
bool fieldIsFree() {
    return busy == false;
}
// Adds a planeID to the queue
void enqueue(planeID id) {
    queue[len++] = id;
}
// Removes a plane from the queue
void dequeue() {
     int i = 0;
     len -= 1;
     while (i < len)
     {
         queue[i] = queue[i + 1];
         i++;
     }
}
// Returns if the queue is empty
bool isEmpty() {
    return len == 0;
}
// Returns the plane that has the next turn
planeID front() {
    return queue[0];
}
```

2.1.3 Plane

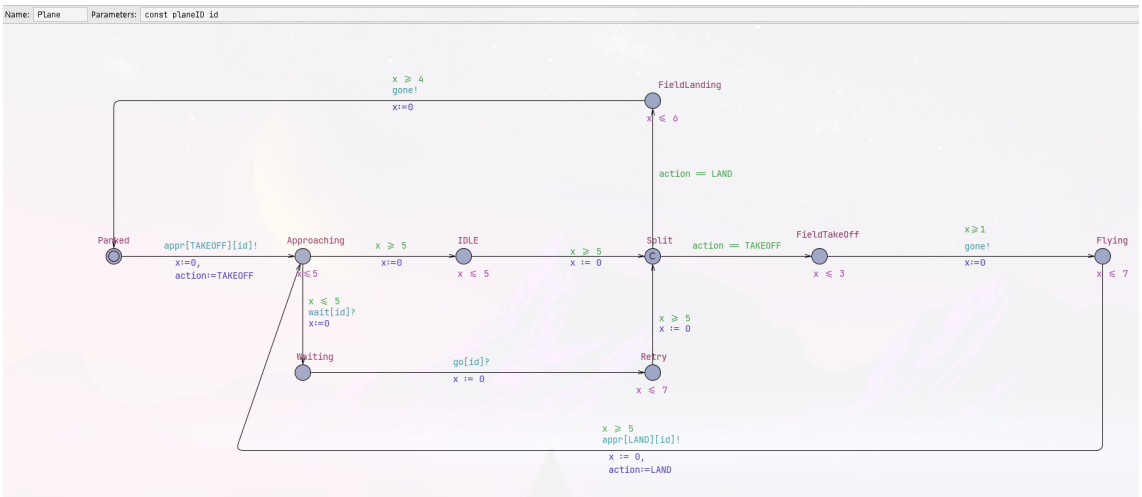

Figure 2.2: Plane

When a plane is `Parked`, it sends the signal `appr[TAKEOFF][id]!`, where `id` is its own integer identifier. It's essential so that the controller knows where to send any eventual reply.

In the next state, it waits a maximum of 5 time units. If a *wait* signal is received within these 5 time units, it will stay in the `Waiting` state until it is given a signal to proceed, taking 5-7 time units to start the take off process afterwards. Otherwise, it moves to `IDLE` and waits another 5 time units before moving to `FieldTakeOff`. This is achieved with an invariant for the `Approaching` state and guards for the transitions mentioned before.

We also certified that a plane takes between 1 and 3 time units to take off by placing an invariant `x <= 3` in the `FieldTakeOff` state and a guard `x >= 1` in the transition to next state, `Flying`. It's relevant to clarify as well that it'll send a `gone` signal to the controller, making clear that it has left the landing field. The limitation of a plane staying in the following state for 5-7 time units was executed with the exact same approach.

After this time, it will move back to the `Approaching` state, where a repetition of the validation process that occurs during take off is performed once more. However, after this stage, it'll move to the `FieldLanding` state, where it stays for 4-6 time units, before finally parking again.

Important general details include the fact that `x` is a local variable, so each plane will have their own individual clock. The same holds true for the integer `action`, used to manage the controller validation that is common to taking off and landing.

Local variables

```
// Internal clock
clock x;
// Action, can be TAKEOFF or LAND
int[0,1] action;
```

2.2 Verification

Having built a working model, we set off to validate that the given constraints were met. For this purpose, we proceeded to define reachability, safety and liveness properties in the `UPPAAL` verifier section. These are essential to verify the fundamental behavior, reliability and integrity of the model.

2.2.1 Reachability Properties

There will be a state where plane 1 is using the field and all other planes will be waiting for their turn.

```
E<> (Plane(1).FieldTakeOff || Plane(1).FieldLanding) && forall (i:planeID)
 i != 1 imply Plane(i).Waiting
```

Plane 1 is able to fly.

```
E<> Plane(1).Flying
```

2.2.2 Safety Properties

There can never be N planes in the queue.

```
A[] Controller.len < N
```

The model should have no deadlocks.

```
A[] not deadlock
```

The field can only be accessed by one plane at a time.

```
A[] forall (i:planeID) forall (j:planeID) (Plane(i).FieldLanding || Plane(
i).FieldTakeOff) && (Plane(j).FieldLanding || Plane(j).FieldTakeOff) imply i
== j
```

It's possible for plane 1 to never leave the Parked state.

```
E[] Plane(1).Parked
```

2.2.3 Liveness Properties

If plane 1 is waiting, its turn will eventually arrive.

```
Plane(1).Waiting --> (Plane(1).Flying || Plane(1).Parked)
```

If a plane leaves the field to fly, it will eventually attempt to return.

```
Plane(1).Flying --> Plane(1).Approaching
```

3. Using the right concepts in software development

Programming is a familiar task for software developers. However, as developers strive to create more efficient and complex applications, it is clear that advanced modelling and verification techniques are needed to ensure reliability and correctness.

Modelling, especially with timed automata, allows us to model real-world situations with precise timing and synchronization requirements. By visualizing all states and transitions, we can gain a comprehensive understanding of the behavior and evolution of the system over time. However, we can identify other types of modelling that use programming or specification languages like Alloy or TLA+.

Having built a model, we can express key requirements that the model should comply with, using temporal logic, such as computational tree logic or linear temporal logic. This way, we can detect potential problems such as deadlocks, livelocks and other violations that can be difficult to detect. This is known as verification.

Modelling is also seen along with verification when working, for example, with `UPPAAL`, an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata. Interesting examples to study can be the modelling of complex algorithms like Fisher's or Lamport's algorithm.

Conversely, programming is done at a lower level of abstraction, where code is written to perform specific tasks and define algorithms. It ends up being the translation of logic to instructions that the computer must execute. Identifying bugs often requires intensive testing, and the existence of bugs cannot be proven until they are found. Therefore, there is no way to guarantee the absences of any errors with this alone.

However, our implementations can be facilitated by mathematics, especially when working with functional programming with effects. Lambda calculus, for instance, can work as a formal system that is the basis of functional programming. Complementing this with the computational power of monads, that allows controlled and composable side-effect handling, error handling, state management, and concurrency, sets the developer up for success.

Although clearly very different from each other, modelling and verification end up complementing programming by refining the testing process. Certainly, analyzing traces and counterexamples is more effective than mindlessly testing trying to understand where a mistake is coming from. With the results we obtain, we can refine the model used and achieve the desired outcome. We must keep in mind, however, that verification is evidently limited by the computational capacity available. Nevertheless, as mentioned before, programming languages for modelling and verification do exist, which only offers more evidence that these can work as allies.

In conclusion, modelling, verification and programming are distinct concepts, that work well together. While modelling and verification end up focusing on capturing the system behavior and checking properties that represent the key requirements it must obey, programming is concrete and practical, implementing the system through code. Using them together, developers can build their implementations with more confidence, since all of them have their own importance in the software development lifecycle.

4. Unified program semantics

On the third part of this assignment, our objective was to develop a probabilistic language utilizing the probabilistic monad. The language was defined by the following grammar:

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

Although most aspects are familiar, there is a novel language construct, $p +_p q$, which entails running p and q with probabilities `p` and `1-p`, respectively.

Thus, our next step involved deriving the semantics of this language by following the example rule for sequential composition, prior to its implementation in Haskell.

4.1 Semantics

$$\frac{\langle p, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i \quad \forall i \leq n. \langle q, \sigma_i\rangle \Downarrow \mu_i}{\langle p; q, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \mu_i} \tag{seq}$$

$$\frac{\langle t, \sigma\rangle \Downarrow r}{\langle x := t, \sigma\rangle \Downarrow 1 \cdot \sigma[r/x]} \tag{asg}$$

$$\frac{\langle p, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i \quad \langle q, \sigma\rangle \Downarrow \sum_i^m q_i \cdot \sigma'_i}{\langle p +_p q, \sigma\rangle \Downarrow \sum_i^n (p_i * p) \cdot \sigma_i + \sum_i^m (q_i * (1-p)) \cdot \sigma'_i} \tag{cho}$$

$$\frac{\langle b, \sigma\rangle \Downarrow tt \quad \langle p, \sigma\rangle \Downarrow \sigma'}{\langle \text{if b then p else q}, \sigma\rangle \Downarrow 1 \cdot \sigma'} \tag{if1}$$

$$\frac{\langle b, \sigma\rangle \Downarrow ff \quad \langle q, \sigma\rangle \Downarrow \sigma'}{\langle \text{if b then p else q}, \sigma\rangle \Downarrow 1 \cdot \sigma'} \tag{if2}$$

$$\frac{\langle b, \sigma\rangle \Downarrow tt \quad \langle p, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \sigma_i \quad \forall i \leq n. \langle \text{while b do } \{p\}, \sigma_i\rangle \Downarrow \mu_i}{\langle \text{while b do } \{p\}, \sigma\rangle \Downarrow \sum_i^n p_i \cdot \mu_i} \tag{wh1}$$

$$\frac{\langle b, \sigma\rangle \Downarrow ff}{\langle \text{while b do } \{p\}, \sigma\rangle \Downarrow 1 \cdot \sigma} \tag{wh2}$$

4.2 Haskell Implementation

4.2.1 Dependencies

In order to achieve the desired implementation for the extended language, we leveraged the libraries at our disposal, which encompassed code pertaining to the probabilistic monad, linear terms, and boolean terms.

4.2.2 Language module definition

To commence, it was imperative for us to transform the grammar into an Haskell data type, yielding the following representation:

```
data ProgTerm = Asg Vars LTerm
              | Choice ProbRep ProgTerm ProgTerm
              | Seq ProgTerm ProgTerm
              | Ife BTerm ProgTerm ProgTerm
              | Wh BTerm ProgTerm
                deriving Show
```

Subsequently, it was necessary to define a function that meticulously evaluates each term within this language. We denoted the aforementioned function as `wsem`. It shall accept a term and a variable-to-double mapping function. The resulting output will be a probabilistic monad encapsulating the variable-to-double function.

```
wsem :: ProgTerm -> (Vars -> Double) -> Dist (Vars -> Double)
wsem (Asg x t) m = return $ chMem x (sem t m) m
wsem (Choice pb p q) m = do pD <- wsem p m ; qD <- wsem q m ; choose pb pD qD
wsem (Seq p q) m = wsem p m >>= wsem q
wsem (Ife b p q) m | bsem b m = wsem p m
                   | otherwise = wsem q m
wsem (Wh b p) m | bsem b m = wsem p m >>= wsem (Wh b p)
                | otherwise = return m
```

To understand the choices made more thoroughly, we shall proceed with a more comprehensive term-by-term analysis.

Regarding the assignment, we employed a `chMem` function that will reassign the variable to the calculated value derived from a linear term.

On the other hand, for the choice term, we recursively evaluate the values of each term, subsequently utilizing the `choose` function, available in the `Probability` library, to apply the desired probability.

To handle the sequence option, we use the bind operator, `>>=`, to pass the result of the execution of p downstream in order to execute q afterwards.

Furthermore, the if-else term is characterized by its straightforward nature. If the evaluation of the boolean term yields a true value, we proceed by utilizing the output obtained from executing `p`. Conversely, if the boolean evaluation yields a false value, we utilize the output derived from executing `q`."

To conclude, the evaluation of the while term closely resembles that of the sequence term. When the evaluation of the boolean term results in a true value, we proceed by applying the execution of the loop to the outcome obtained from evaluating `p`. Conversely, when the boolean evaluation yields a false value, we return the result as it is.

4.3 Testing

In order to facilitate more efficient testing, we implemented a function called `showWsem`, defined below.

```
showWsem :: Dist (Vars -> Double) -> Vars -> Dist (Double)
showWsem m x = do v <- m ; return $ v x
```

By doing so, we were able to utilize the existing `Show` implementation for `Dist(Double)` provided by the `Probability` library.

5. Final Thoughts

In this report, we aimed to give a comprehensive overview of our project and to shed light on the complexity of UPPAAL modelling, the symbiotic relationship between modelling and verification, and the fascinating field of probabilistic programming in Haskell. By taking on these tasks, we not only increased our knowledge and practical skills, but also gained valuable insights into the interdisciplinary nature of cyber-physical programming and its importance in real-world applications.