

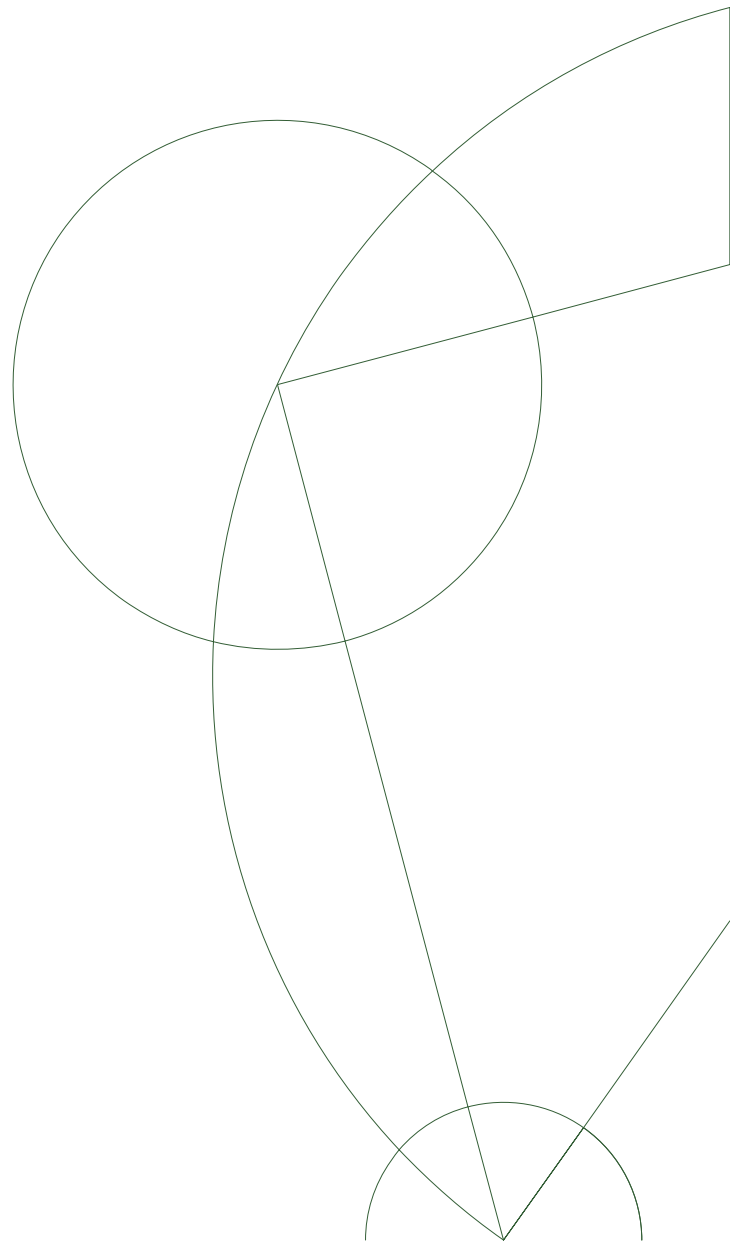


# Reactive & Event-Based Systems

## Assignment 1: Process Models & Event-Based Systems

Frederik Ingwersen  
Rasmus Winther

December 6, 2019



# Model descriptions

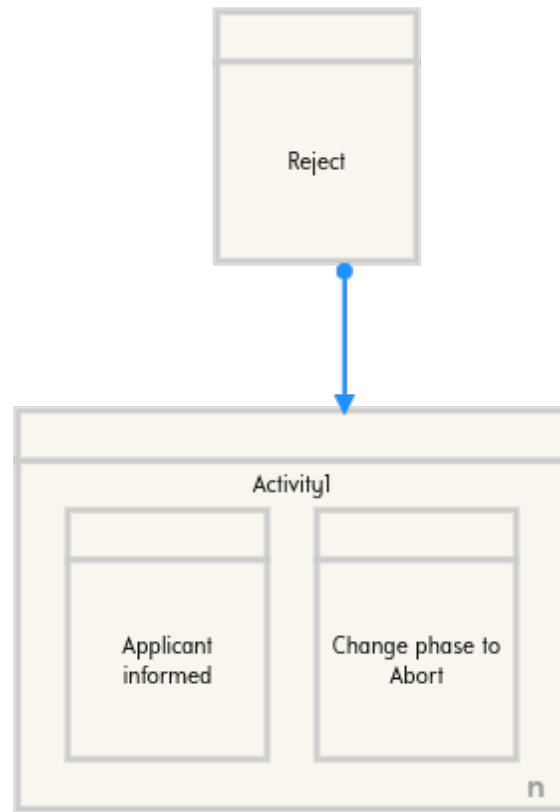
## Pattern 1

With this model, we had to find all the activities in the Dreyers log, which we simply did by making a small Python script. We added an additional activity called "Case" to avoid having a condition arrow pointing to each and every one of the activities.



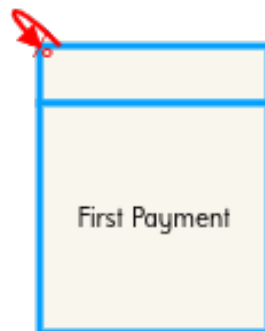
## Pattern 2

Again, we nested the two within an outer activity to only draw one pending arrow. In the assignment text, however, it is stated that both Applicant informed and Change phase to Abort should happen after Reject, eventually. We were a bit in doubt whether this should mean that Applicant informed should happen after Reject and then Change phase to Abort, or if the order is irrelevant. We chose the latter.



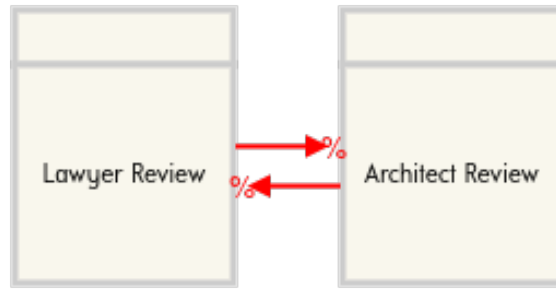
## Pattern 3

This excludes itself after it has been executed.



## Pattern 4

These exclude each other after execution, making sure only one of them is ever executed.



## Engine

We decided to write our own DCR engine with Go. The reason for this was largely due to the fact that we wanted to learn more about the intricacies of working with the DCR graphs to deepen our understanding, and, of course, to challenge ourselves a bit.

### XML Format

First off, the input format; we created our own DCR graph input format, very much inspired by the XML output you get from downloading a graph from *dcrgraphs.net*. It is a simpler format where we start with an outer graph of type *dcr* and move in from there. Beneath, a simple overview:

- **dcr**
  - **events**
    - **event 1**
  - **constraints**
    - **conditions**
    - **responses**
      - **response 1**
    - **correspondences**
    - **excludes**
    - **includes**
    - **milestones**
    - **spawns**
  - **markings**
    - **executed**
    - **included**
      - **event 1**
  - **pending**

Now, we never implemented spawns, but they are within our format. One thing to note is that this format does not support nesting. Looking at the actual XML provided should make entries such as the response entry clear.

## Engine overview

The engine itself is fairly simple, consisting of three main files which make up the actual engine, two files to implement some helper functions for data handling and XML and a single file to execute. We will go over the three main files and omit the rest.

**DCR\_structs.go** is a short file where we declare the structs we want to use. Some these structs are actually used to directly pick out information from the XML format we created ourselves.

**DCR\_interface.go** is where most of our engine is implemented. We define an interface called DCR and the methods it should implement. In Go, this works a bit differently than in OOP languages like C++ or Java; so, here, we just declare functions like so: *func (struct) Method name return type*. The struct here is the struct we want to add this method to, once a struct has all methods of an interface, the interface is said to be implemented by the struct. The implementation is very much our own interpretation of the code shown in the slides, so we won't go into detail.

**DCR\_functions.go** holds the helper functions, which did not make sense to put onto the struct of DCR\_graph. These functions are ones which act on something within the DCR\_graph struct, but not the entire data structure. They are used throughout the interface implementations, as well as in the **main.go** file to create the graphs from our XML files.

## Conformance Checker

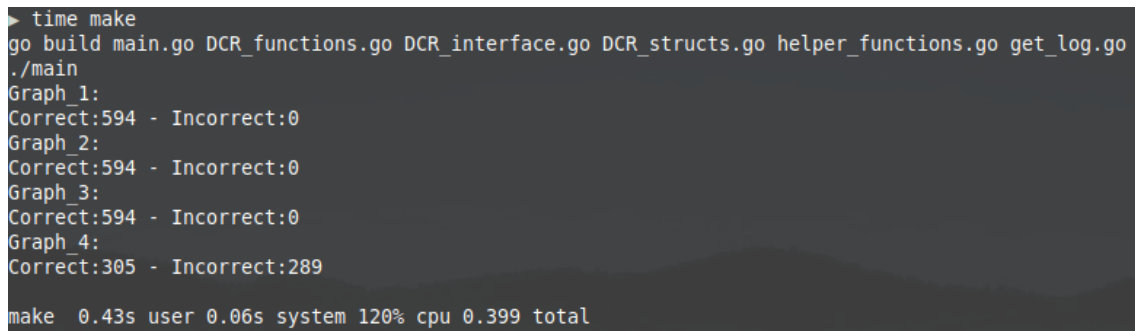
To check how a graph satisfies a trace we compile a set of all legal events with the current marking of the graph. If the event is included in the set of legal events the event is executed and the marking is updated. This process is repeated every event in the trace. If any event is not in legal events is the trace not satisfied. If every event is executed the set of pending events is checked and if no events are pending the trace the graph is concluded to satisfy the trace.

## Tests

We tested the result by hardcoding the patterns and check if we get the same count of right and wrong traces. The specifics of how each pattern is made can be seen in the file test.go. With the testing we found a mishandled cases in the way we handle pending relations. We found that at an event could be multiple times in the pending set of the marking and only one was removed when that event was executed.

## Display of results

Below, an image of our program compiling and running. We put a time such as to show, that different images of program run state would not be relevant.



```
> time make
go build main.go DCR_functions.go DCR_interface.go DCR_structs.go helper_functions.go get_log.go
./main
Graph_1:
Correct:594 - Incorrect:0
Graph_2:
Correct:594 - Incorrect:0
Graph_3:
Correct:594 - Incorrect:0
Graph_4:
Correct:305 - Incorrect:289
make 0.43s user 0.06s system 120% cpu 0.399 total
```

Below, a table showcasing those same results.

Pattern	Traces satisfied	Traces unsatisfied
Pattern 1	594	0
Pattern 2	594	0
Pattern 3	594	0
Pattern 4	305	289