Rebs Exam 2021

Christian Påbøl Jacobsen(wbr220)

University of Copenhagen – DIKU

28/01 – 2021

Introduction

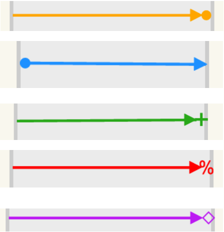
Three different assignments to talk about

DCR Graphs - How and why

introduction

- Dynamic Condition Response
- Modelling using DCR Graphs
- "The Analysis of a Real Life Declarative Process" - Slaats & Debois

DCR Connections used



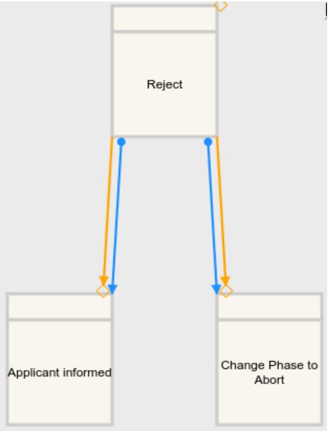
Condition

Response

Include

Exclude

Milestone



The Assignment

Part 1

We then model four patterns in the assignment.

Pattern 1

Fill out Application

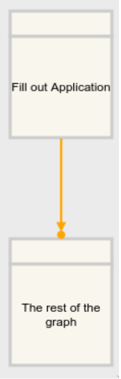


Figure: Fill out application must come before the rest of the graph

Pattern 2

Reject should always eventually be followed by "Applicant informed" and "Change phase to Abort"

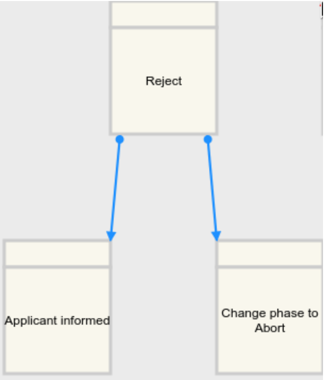


Figure: Using the Response relation

Pattern 3

First Payment must only occur once

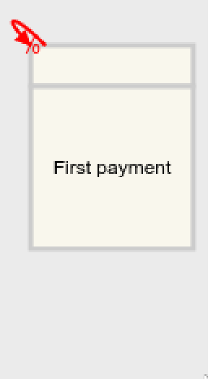


Figure: Excluding the sender

Pattern 4

Only one of the reviews must occur at the same time

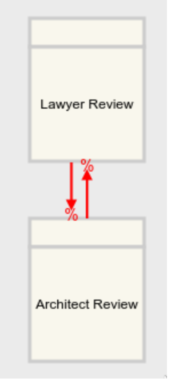
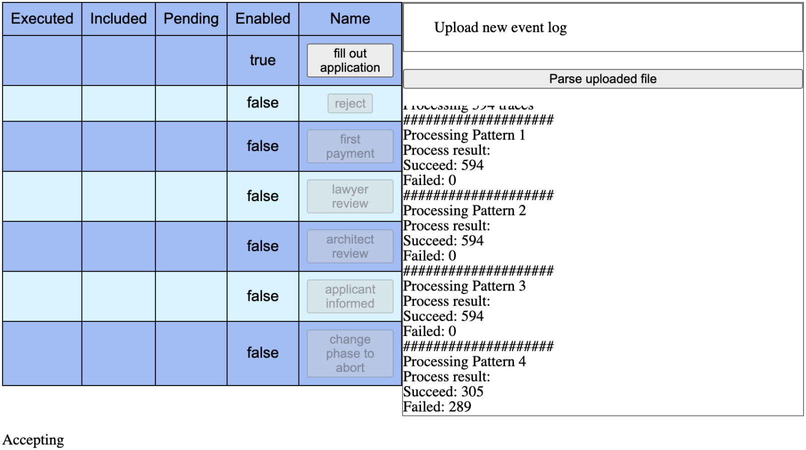


Figure: Excluding another activity

Conformance Checking

Additions To a handed-out DCR Implementation



Executed	Included	Pending	Enabled	Name
			true	fill out application
			false	reject
			false	first payment
			false	lawyer review
			false	architect review
			false	applicant informed
			false	change phase to abort

```
Processing 594 traces
####################
Processing Pattern 1
Process result:
Succeed: 594
Failed: 0
####################
Processing Pattern 2
Process result:
Succeed: 594
Failed: 0
####################
Processing Pattern 3
Process result:
Succeed: 594
Failed: 0
####################
Processing Pattern 4
Process result:
Succeed: 305
Failed: 289
```

Accepting

This Slide

Intentionally left blank

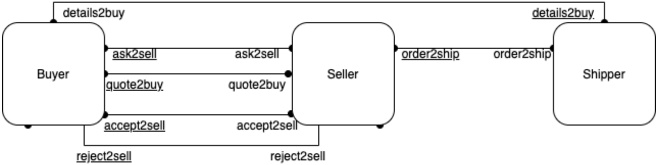
Part 2 of the Course - CCS Choreographies and Jolie

introduction

- Changing focus a bit
- Choreographies and Jolie

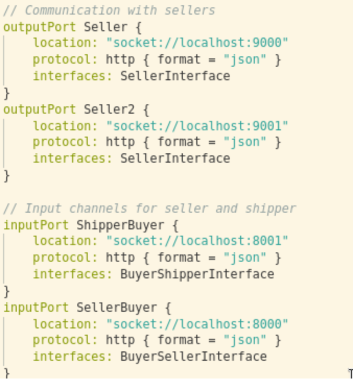
Modelling Buyers, Sellers, and Shippers

The interface Diagram



Buyer

Input and outputs

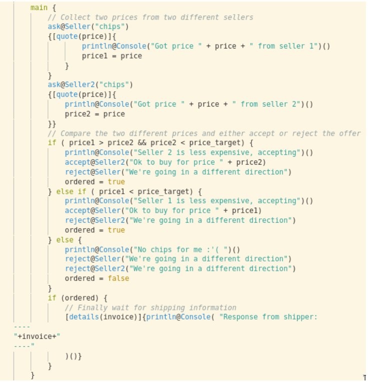


```
// Communication with sellers
outputPort Seller {
    location: "socket://localhost:9000"
    protocol: http { format = "json" }
    interfaces: SellerInterface
}
outputPort Seller2 {
    location: "socket://localhost:9001"
    protocol: http { format = "json" }
    interfaces: SellerInterface
}

// Input channels for seller and shipper
inputPort ShipperBuyer {
    location: "socket://localhost:8001"
    protocol: http { format = "json" }
    interfaces: BuyerShipperInterface
}
inputPort SellerBuyer {
    location: "socket://localhost:8000"
    protocol: http { format = "json" }
    interfaces: BuyerSellerInterface
}
```

Figure: Outputs and Inputs, is this O/I

Buyer

Business Logic

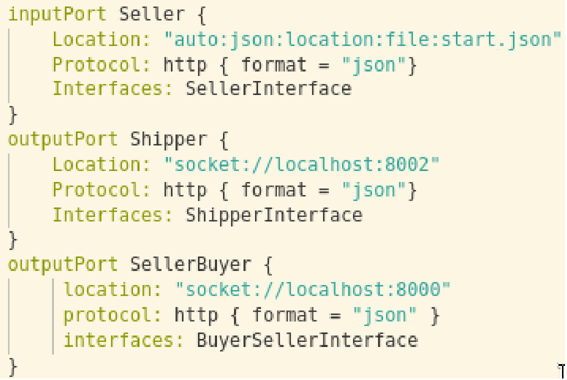


```
    main {
        // Collect two prices from two different sellers
        ask@Seller("chips")
        {[quote(price)]{
                println@Console("Got price " + price + " from seller 1")()
                price1 = price
            }
        }
        ask@Seller2("chips")
        {[quote(price)]{
            println@Console("Got price " + price + " from seller 2")()
            price2 = price
        }}
        // Compare the two different prices and either accept or reject the offer
        if ( price1 > price2 && price2 < price_target) {
            println@Console("Seller 2 is less expensive, accepting")()
            accept@Seller2("Ok to buy for price " + price2)
            reject@Seller("We're going in a different direction")
            ordered = true
        } else if ( price1 < price_target) {
            println@Console("Seller 1 is less expensive, accepting")()
            accept@Seller("Ok to buy for price " + price1)
            reject@Seller2("We're going in a different direction")
            ordered = true
        } else {
            println@Console("No chips for me :'( ")()
            reject@Seller("We're going in a different direction")
            reject@Seller2("We're going in a different direction")
            ordered = false
        }
        if (ordered) {
            // Finally wait for shipping information
            [details(invoice)]{println@Console( "Response from shipper:
----
"+invoice+"
----"
            )()}
        }
    }
```

Figure: This isn't the smartest solution *nor* the dumbest

Seller

Input and outputs

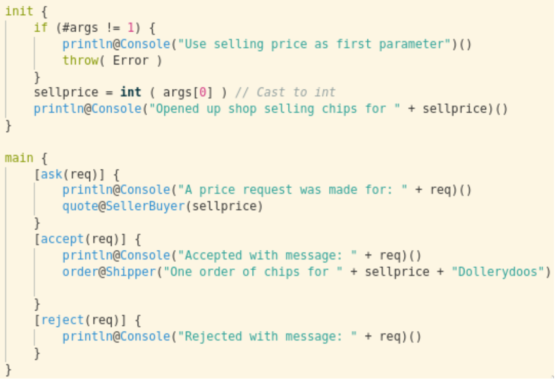


```
inputPort Seller {
    Location: "auto:json:location:file:start.json"
    Protocol: http { format = "json"}
    Interfaces: SellerInterface
}
outputPort Shipper {
    Location: "socket://localhost:8002"
    Protocol: http { format = "json"}
    Interfaces: ShipperInterface
}
outputPort SellerBuyer {
     location: "socket://localhost:8000"
     protocol: http { format = "json" }
     interfaces: BuyerSellerInterface
}
```

Figure: Notice the dynamic location

Seller

Business Logic

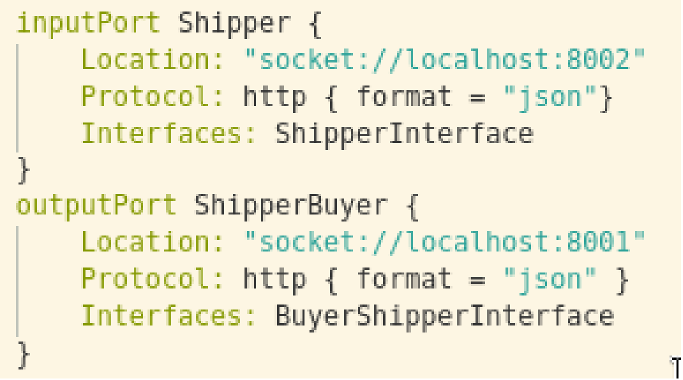


```
init {
    if (#args != 1) {
        println@Console("Use selling price as first parameter")()
        throw( Error )
    }
    sellprice = int ( args[0] ) // Cast to int
    println@Console("Opened up shop selling chips for " + sellprice)()
}

main {
    [ask(req)] {
        println@Console("A price request was made for: " + req)()
        quote@SellerBuyer(sellprice)
    }
    [accept(req)] {
        println@Console("Accepted with message: " + req)()
        order@Shipper("One order of chips for " + sellprice + "Dollerydoos")

    }
    [reject(req)] {
        println@Console("Rejected with message: " + req)()
    }
}
```

Figure: Main and init is quite simple to write despite the logic involved

Shipper

Input and outputs

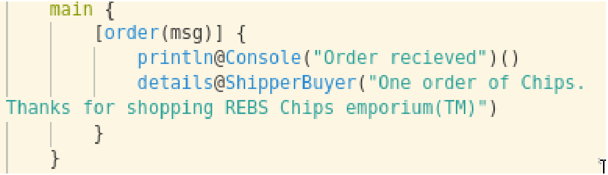


```
inputPort Shipper {
    Location: "socket://localhost:8002"
    Protocol: http { format = "json"}
    Interfaces: ShipperInterface
}
outputPort ShipperBuyer {
    Location: "socket://localhost:8001"
    Protocol: http { format = "json" }
    Interfaces: BuyerShipperInterface
}
```

Figure: Doesn't need to know sellers location

Shipper

Business Logic



```
    main {
        [order(msg)] {
            println@Console("Order recieved")()
            details@ShipperBuyer("One order of Chips.
Thanks for shopping REBS Chips emporium(TM)")
        }
    }
```

Figure: We don’t actually ship anything

This Slide

Intentionally left blank

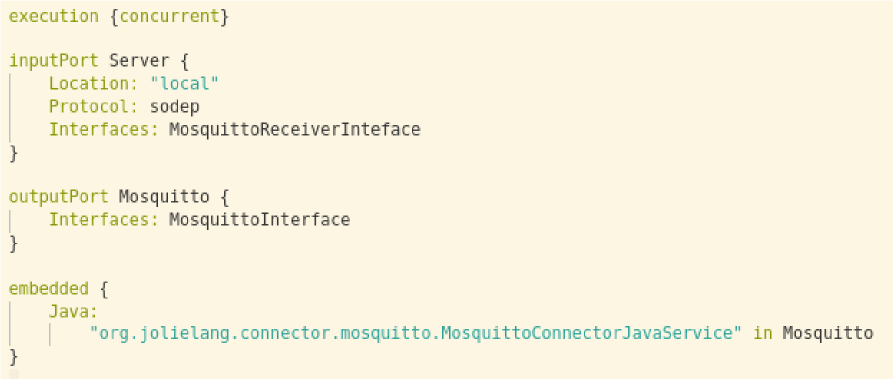
Jolie, MQTT and how to tie them together

introduction

- Message Queuing Telemetry Transport (MQTT)
- A subscriber-based communications protocol

A Filtered Subscriber

Input and outputs

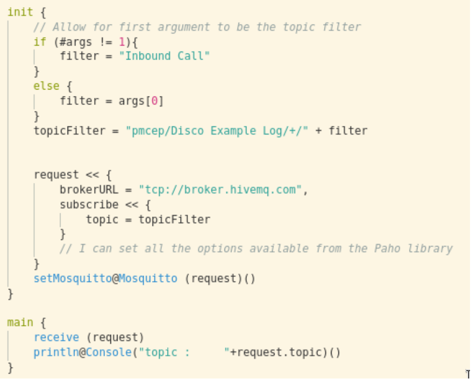


```
execution {concurrent}

inputPort Server {
    Location: "local"
    Protocol: sodep
    Interfaces: MosquittoReceiverInteface
}

outputPort Mosquitto {
    Interfaces: MosquittoInterface
}

embedded {
    Java:
        "org.jolielang.connector.mosquitto.MosquittoConnectorJavaService" in Mosquitto
}
```

Figure: Notice the mosquitto interfaces

A Filtered Subscriber

Business Logic



```
init {
    // Allow for first argument to be the topic filter
    if (#args != 1){
        filter = "Inbound Call"
    }
    else {
        filter = args[0]
    }
    topicFilter = "pmcep/Disco Example Log/+/" + filter


    request << {
        brokerURL = "tcp://broker.hivemq.com",
        subscribe << {
            topic = topicFilter
        }
        // I can set all the options available from the Paho library
    }
    setMosquitto@Mosquitto (request)()
}

main {
    receive (request)
    println@Console("topic :     "+request.topic)()
}
```

Figure: Takes filters from the command line

Making the whole thing a bit more interesting

Parsing the MQTT response

What did we do:

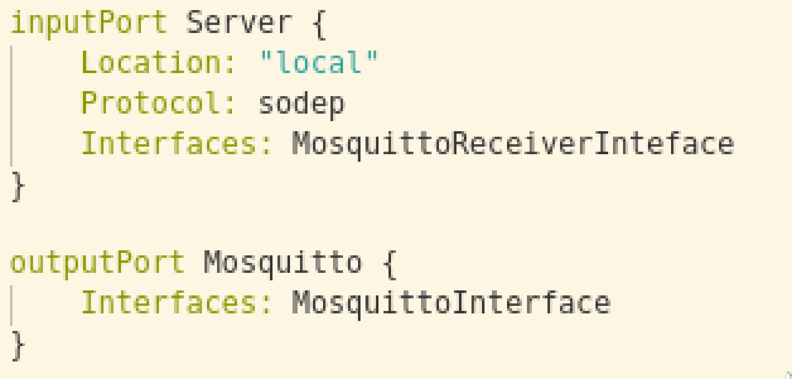
- Subscribe to a specific topic
- Notify the stdout everytime an activity comes in
- Show off our different wildcards

Whats next

- Responses are in json
- Jolie has global variables
- Jolie has File IO

A Counting Subscriber

Input and outputs

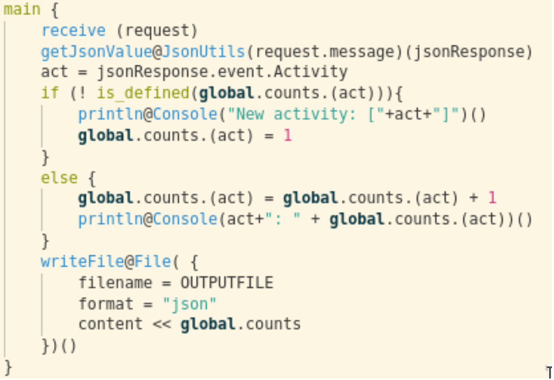


```
inputPort Server {
    Location: "local"
    Protocol: sodep
    Interfaces: MosquittoReceiverInteface
}

outputPort Mosquitto {
    Interfaces: MosquittoInterface
}
```

Figure: Same as previous subscriber

A Counting Subscriber

Business Logic

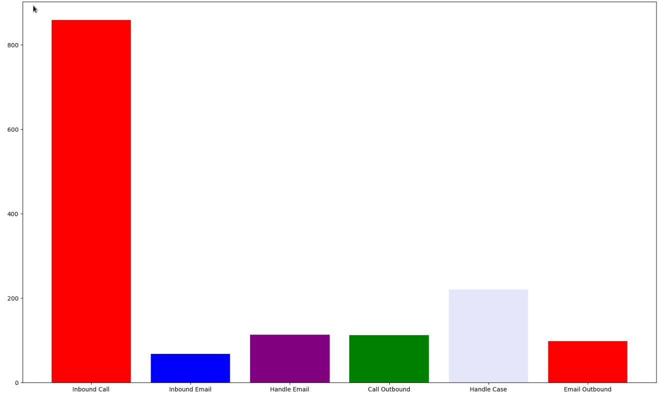


```
main {
    receive (request)
    getJsonValue@JsonUtils(request.message)(jsonResponse)
    act = jsonResponse.event.Activity
    if (! is_defined(global.counts.(act))){
        println@Console("New activity: ["+act+"]")()
        global.counts.(act) = 1
    }
    else {
        global.counts.(act) = global.counts.(act) + 1
        println@Console(act+": " + global.counts.(act))()
    }
    writeFile@File( {
        filename = OUTPUTFILE
        format = "json"
        content << global.counts
    })()
}
```

Figure: Our main function has tripled in size

Plotting the results

People Call more than they email



This Slide

Intentionally left blank