UNIVERSITY OF NANTES - SCIENCES AND TECHNOLOGY

Capstone Project

Model-driven engineering for Industry 4.0 and Manufacturing

Students:

Antoine MAGNIN
Nathan SALAÜN

Supervisors:

Pascal ANDRÉ
Olivier CARDIN

2018 - 2019

Contents

Introduction

Management of production lines is a complex matter, with a myriad of use cases in the industry field and raising interrogations in the research field. The end goal would be to produce a software able to drive a production line from a computer, optimising both operation time and costs.

One of the answers to that problem is to use operational research algorithms in order to get an optimal solution, but this approach is sorely lacking in flexibility: even if the suggested solutions are almost perfect, they cannot react accordingly to breakdowns and other possible failures.

Thus, a few years ago, Francisco Gamboa defined a multi-agent system architecture named SoHMS (**S**ervice-**o**riented **H**olonic **M**anufacturing **S**ystem) aiming to answer the problem while keeping in mind these new flexibility requirements. Over the following years, this project was iterated on several times by multiple students, each working on a different aspect of the system.

In this document, we will first explain what is an SoHMS system and its operation. Then, we will describe what we achieved within the project. Finally, we will talk about the future of the project, where we left off and what is left to do.

1 SoHMS and its operation

1.1 The concept of holons and holarchy

A holon is a philosophical concept meaning to represent something that is both a part and a whole (e.g : fractals, or "seeds and trees" as a tree contains seeds but is also contained in a seed). In our case, a holon is an agent that can be composed of other agents. The connections between holons are called a holarchy, which is a multi-agent system where all agents are holons.

1.2 The SoHMS architecture

The SoHMS architecture is deeply connected whit the very concept of holons. Five different types of holons currently exist and communicate with each others.

1.2.1 The whole architecture

In order to specify what the production area is made of, i.e what machines are there and what services are offered, a scenario is written in JSON (see Appendix for an example). It is then fed to the program that initialises the services, resources, products, order, and fills the directory facilitator with tuples made of a machine and the services that it offers along with the parameters, for instance : M1, "painting", "blue".

The order holon (OH) is then split into several resource order holons (ROH), one for each product in the list of order. They may not all be initialised at the beginning however, as the ROH pool size is limited by the maximum number of parallel units (e.g if we have 5 products to make and the max parallel units is 3, only 3 ROH will be initialised at the beginning and a new ROH will be initialised once a slot is freed).

1.2.2 Order Holon

The order holon (OH) is in charge of each order, an order being a product that has to be made by being processed by several machines, and is called a resource order holon (ROH). It is also in charge of negotiating access to services for its ROHs, meaning it will communicate a lot with RHs in order to schedule meetings between the products and the machines.

1.2.3 Product Holon, Resource Holon

The product holon (PH) is a recipe for a product, meaning there are as much PHs as there are recipes. The recipe is simply the services the product will have to go through. For instance, in order to make a final product P1, the product will have to

through S1, S2, and S3 meaning the PH representing P1 will be {S1, S2, S3}. Thus, each product is able to ask a PH where it has to go next. An instance of said product is called a product order holon (POH).

The resource holon (RH) represents a resource (machines and forklifts). Each RH indicates if a resource is available for processing a product or if it is already taken by another one. As mentioned above, an RH will talk with the OH in order to plan a meeting between the resource RH and the future product ROH.

1.2.4 Resource Order Holon, Product Order Holon

The resource order holon (ROH) is what makes up the initial OH, and said OH is then split in ROHs. For instance if the OH is to make 5x finished product P1, then we will end up with 5 ROHs, each one corresponding to a P1 in the making. Each ROH is also linked to a product order holon (POH) in order to track the progress of the ROH.

The POH is useful as it allows the ROH to know where it is located in the production process. For example, let's say a product P1 is made by having an initial product go through the services S1, S2, and S3 in that order. The ROH will ask the POH which service it has to use at the beginning and the POH will answer S1. Then once the ROH has been processed through S1, it will again ask the POH what to do and the POH will answer with the next service (here S2).

POHs and ROHs are deeply linked together, as an ROH will communicate with only one POH and a POH will be associated with only one ROH.

1.2.5 Directory Facilitator

The Directory Facilitator (DF) is a tool used to maintain record of what machine offers what service with what parameters. It is initialised at the very beginning of the program, and is a list of tuples in the form of (resource, service, parameters). It is essential that it is filled properly because it is the only way for an ROH to know what machines it can use to benefit from a specific service.

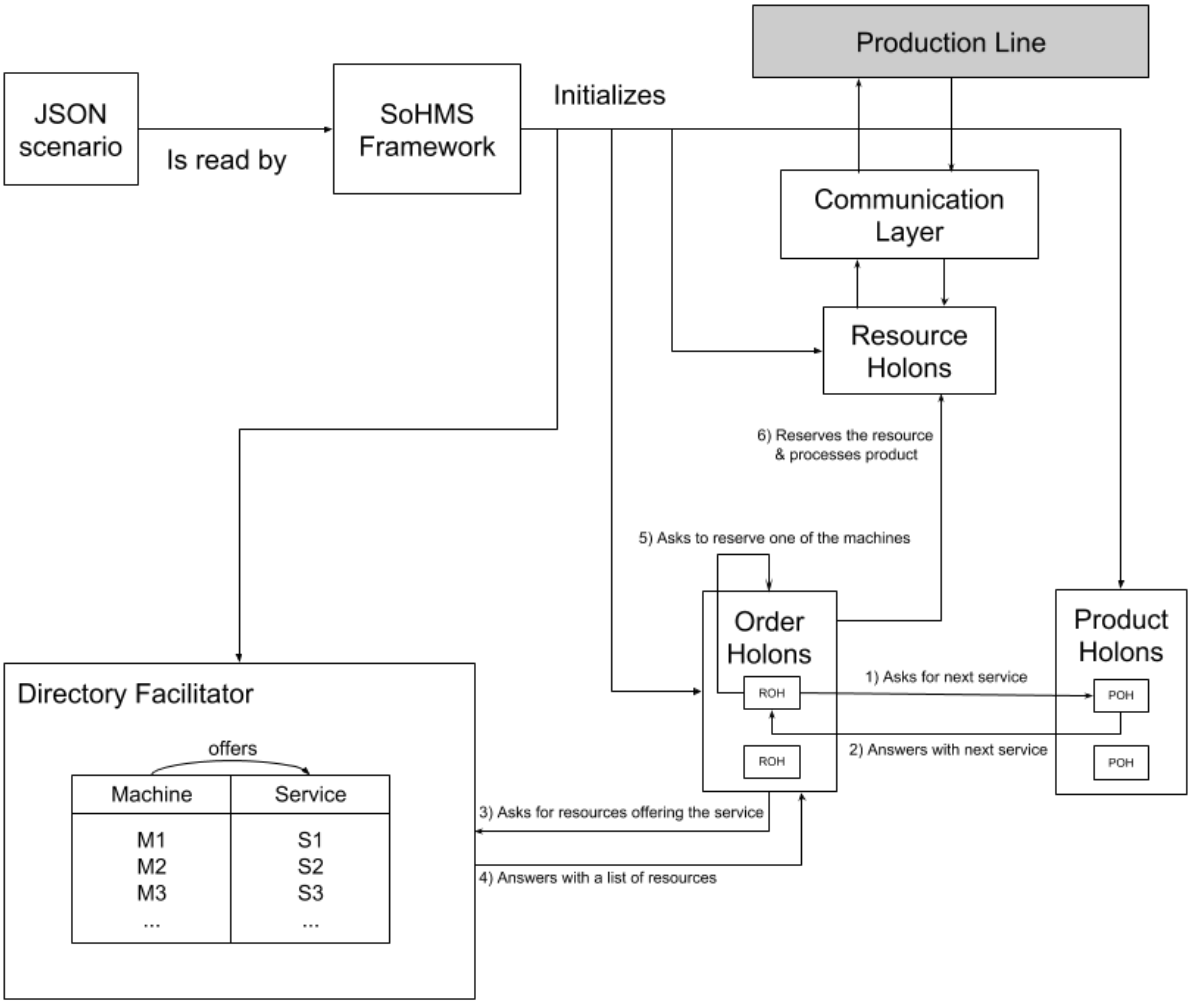

Figure 1.1: The architecture of SoHMS

The operation of the SoHMS system is illustrated here in figure 1.1.

1.3 How does it actually work ?

In order to understand how the SoHMS system is able to deliver finished products, all we have to do is understand what the life cycle of an ROH is.

The ROH is obtained by decomposing the initial order, each ROH being a future finished product. The ROH will then execute the following steps until no more services are needed to go through:

- The ROH asks his POH which service is the next one
- The POH answers with a service. If the ROH went through all the services needed, it is instead moved to the SINK
- The ROH asks the DF which resources provide said service
- The DF answers with a list of resources
- The ROH asks the OH to reserve one of the resources
- The ROH is moved to the reserved machine and is processed there
- Once the process is done, the ROH needs the next service so it goes back at the beginning

Each ROH goes through this cycle, and once each ROH has completed its products then the order is done.

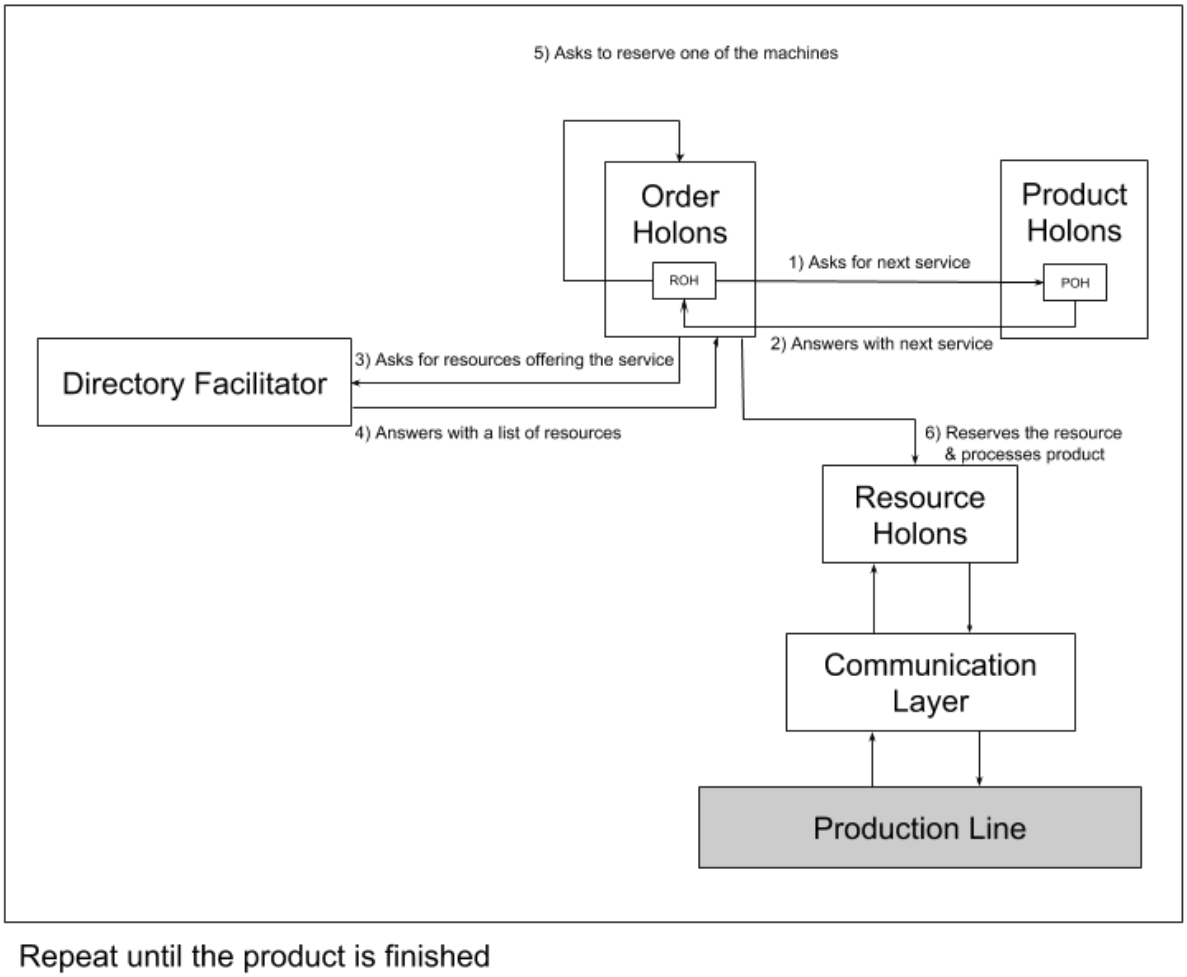

Figure 1.2: Lifecycle of an order

The lifecycle of an order is illustrated here in figure 1.2.

2 Our contribution: a working proof of concept

One of the main goal of our work was to produce a working proof of concept of the SoHMS system on a simulated production line.

The Java code resulting from this work can be found in our GitHub fork of the SoHMS system : `https://github.com/natinusala/SoHMS_Framework`. We recommend IntelliJ IDEA to build and run the project, as the repository already contains IntelliJ project files.

2.1 Our production line: a FlexSim Model

Olivier Cardin, one of the supervisors of the project, made a model of a simple production line within a software called FlexSim. The model is very simple and has very few components:

- A forklift which can move products between two points in the model (called `transporter` in the system)
- Three machines (`M1`, `M2` and `M3`) than can take an input product, process it and output another product
- A source point to take products from (`SOURCE`)
- A sink point to dispose of finished products (`SINK`)

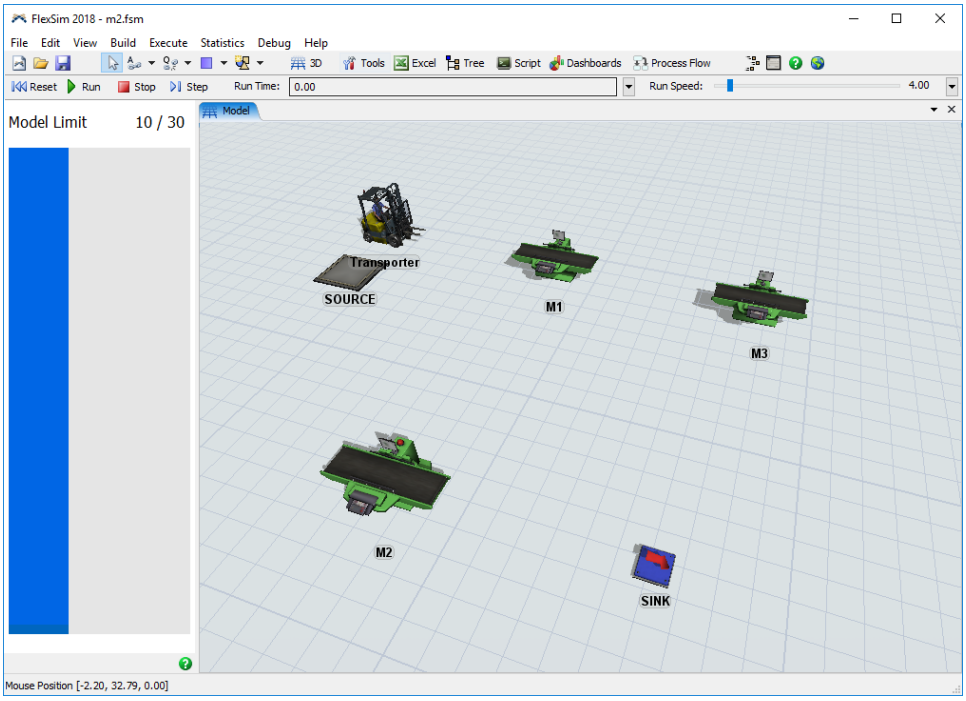

Figure 2.1: The model as seen in FlexSim

Note that the model doesn't have any notion of services. Therefore, it is assumed that each machine can do one and only one service, without any parameters. This is further explained in section 2.2.

The model can be driven using socket communication: once the simulation is started in FlexSim, the model will try to connect to a local server on a known port. The FlexSim model is **client**, and the driving software is **server**. Once the connection is established, the server can send commands to the model and "make it move" as it sees fit.

For now, there are two commands that are recognised by the model: `MOVE` and `PROCESS`.

`MOVE` is used to make the forklift move from a starting point to a destination point, taking the product of the starting point (if any) and putting it on the destination point. For instance, `MOVE SOURCE SINK` will have the forklift move from its current position to the source, take a product there, move to the sink and put the product there.

`PROCESS` will make a machine process their current product (if any) for a given amount of time. For example, sending `PROCESS M1 10` will make the machine `M1` process its product for ten seconds.

The model can receive commands, but also gives feedback to the server, when the forklift starts or stops moving, or when a machine finishes its processing, among others. For a detailed documentation of the model protocol, see appendix.

The model is abstract, and doesn't reflect exactly a production line as seen by the SoHMS system. For instance, "processing" is an abstraction term specific to the model; the SoHMS system sees "processing" as "execute service `x` on machine `y`", which is then translated to a process command on machine `y`. The same can be said for the forklift, which should be a resource on its own with a transport service, and not a standalone move command. This is part of the improvements that need to be made to the system (see section 3).

2.2 Splitting the SoHMS system in two

When he wrote his thesis, Francisco Gamboa also made a first version of the SoHMS system, written in Java. It worked for the production line he used, but everything was hard-coded into the system. This means that to make it work on another production line (different layout, different resources, different mechanisms...), he had to dig into the code and rewrite parts of the system.

This was the motivation behind Mohammed Tebib's work. Mohammed was an ALMA student back in 2017 (the school year before ours) - he spent his Master internship working on this project. His task was, among others, to split the SoHMS system in two :

- The SoHMS system itself in the form of a Java framework (a `JAR` library), implementing everything generic to all production lines and nothing more
- A Java program for each different production line, using the SoHMS system as a dependency, implementing everything specific to that one production line

Mohammed started to separate the SoHMS system, remove everything specific to Francisco's production line and write a Java program to make it work on the FlexSim model described earlier. However, he didn't have enough time to finish it before the end of his internship.

Finishing this work was a prerequisite to make our proof of concept, as it is based on the framework version of the system. Figure 2.2 explains our approach to the matter.

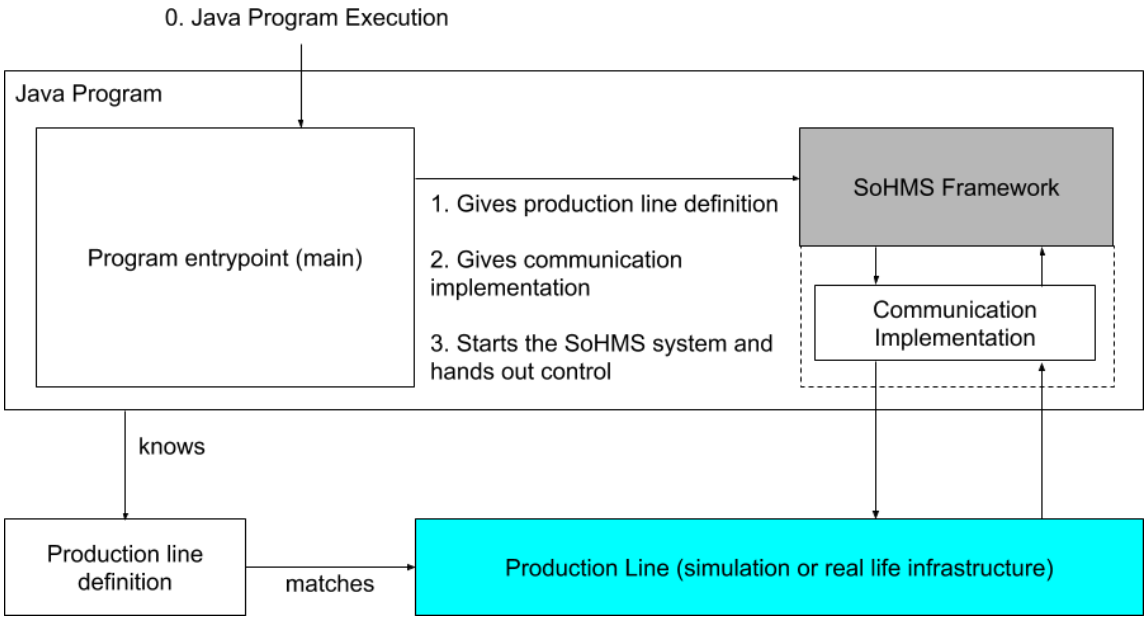

Figure 2.2: Separation between the SoHMS framework and the Java program

Each white rectangle represents what needs to be written for a particular production line : the Java program, the communication implementation and the production line definition (also called "scenario" or "scenario file"). The Java program is very simple: its goal is to invoke the SoHMS framework, giving the path to the production line definition file and the communication implementation (responsible for translating and sending service executions commands over). Once everything is loaded, the SoHMS system is started and takes over the program.

The scenario contains the following data about the production line:

- all services with parameters and attributes for each one
- all resources with a list of offered services for each one
- all products with the steps required to make each one (a list of services)
- a list of orders to be executed when the system starts (each order is a product and the number of units to make)

We then implemented this for our simulated production line (FlexSim model). The product line definition we used can be found in the appendix. The communication implementation is a socket server, connected to the FlexSim model. We wrote two versions of the Java program: one which directly loads our test scenario and starts the system immediately, and another with no hard-coded scenario and a graphical user interface for showcasing and distribution purposes.

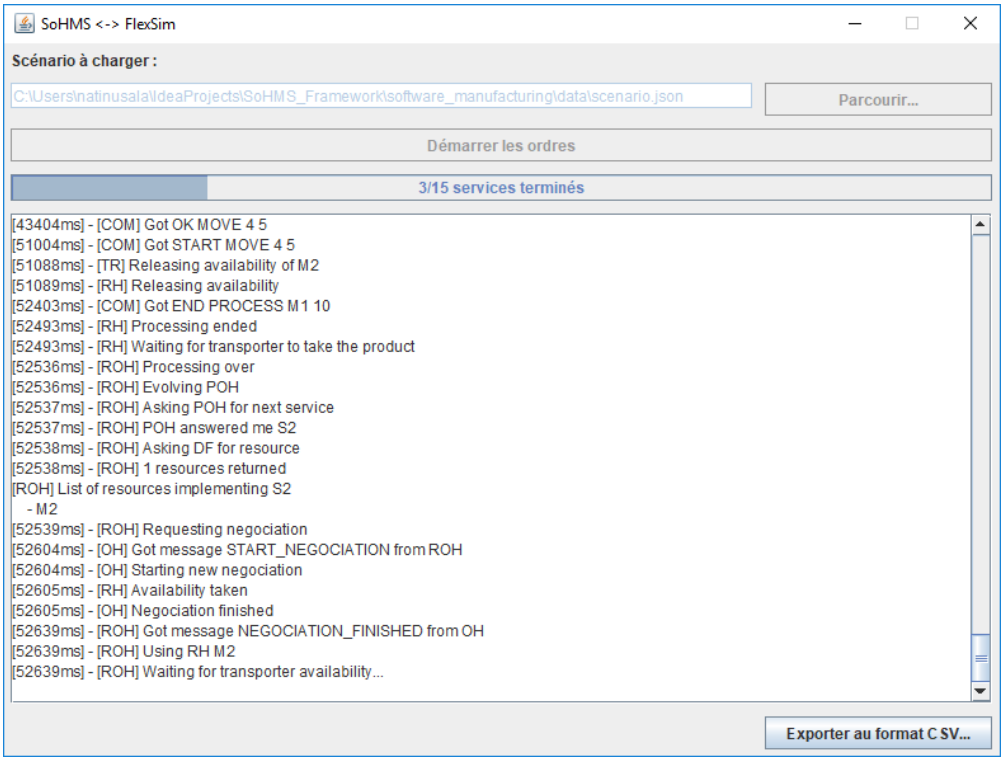

Figure 2.3: Screenshot of the GUI

As you can see on figure 2.3, we also added logging features to the system - the system logs every action and decision it does, each one being associated to a timestamp (in milliseconds). This allows us to gather data such as negotiation time or communication delays. Those logs can be exported to a CSV file for further analysis.

2.3 Making the model move: finishing the framework

Once the system was split in two, we needed to actually write what was needed to make it work on our FlexSim simulation. The code base was still full of code specific to Francisco's production line, so we had to carve that out and make a system which could work on every production line (defined by the scenario).

First, we decided to sanitise both the scenario and the code parsing it. We used the Gson model-based parsing library to have an efficient and easy to read and maintain parsing method. We also removed from the scenario everything which was specific to Francisco's production line and didn't make sense for our use case anymore. Then, we rewrote all the initialisation part to make use of that newly created scenario file and its new model. Note that class diagrams and some sequence diagrams are included in the appendix.

This reduced the Java application to a four lines program (not counting the FlexSim communication layer implementation): create the communication layer object, initialise the system with our scenario file and start it.

```java
public class Application implements Runnable {
    public Application() {
        ComInterface comFlexsim = new ComFlexsim();
        Init.initializeSystems(comFlexsim, "data/scenario.json", this);
        Init.launchAllOrders();
    }

    // Fired when all orders are finished (quick listener pattern)
    @Override
    public void run() {
        HistoryManager.saveCsvToFile("logs/history.csv");
    }
}
```

Note that `Application` implements the `Runnable` interface, hence the `this` in `initializeSystems`. The `run()` method is fired to notify when all orders are completed. Here, it just saves logs to a CSV file.

When the SoHMS system is started through the `launchAllOrders` method, each Order Holon's thread is started (called `OrderManager` in the code). We only have one in our scenario. When running, this thread creates one ROH per parallel product to make (in our case it's three) and starts another ROH each time an order is completed (if needed). It is also responsible of "booking" resources for the products (called negotiation in the code).

Communication between threads needs to be non-blocking, what is the point of using threads otherwise? To achieve both ways asynchronous communication, we used two thread-safe queues: one from thread A to thread B, and another from thread B to thread A. When two threads want to communicate, they create and share a `ThreadCommunicationChannel` object. It's assumed that programmers know in advance which thread will be thread A, and which is thread B.

To send a message, each thread needs to write to the corresponding queue (by using one of the `sendTo` methods). To "receive" a message, each thread needs to poll the corresponding queue (by using one of the `read` methods). If no message is to be received, the method will just return `null` in order not to block the calling thread. It's very similar to a mailbox system.

The ROH thread is the brain of the system. Its code is responsible for making the system move forward. Here is its logic, all in a loop :

1. Ask its associated POH for the next service to be executed on the product, along with its parameters and attributes
2. If no service is to be executed :
   (a) Wait for the forklift to be ready
   (b) Reserve the forklift and move the product to the sink
   (c) Break the loop and stop the thread
3. Ask the Directory Facilitator for a list of resources offering the service with the corresponding parameters and attributes
4. Ask the Order Holon to negociate for us and reserve one resource among all those offering the service
5. Wait for the Order Holon's response and remember the chosen resource
6. Wait for the forklift to be ready
7. Reserve the forklift and move the product to the chosen resource (we assume that the resource is already reserved and therefore not busy)
8. Wait for the forklift to move and put the product on the resource
9. Ask the resource to execute the service
10. Wait until the service is fully executed
11. Notify the POH that the service has been executed
12. Loop back to step 1

As you can see, each ROH can be executed in parallel and will move at its own pace. This allows us to have multiple products being processed at once.

The Order Holon (or Order Manager), on its own thread, manages each ROH it created. It checks if their product is located in the sink to see if it is finished, and launches another ROH in its place if needed. When all ROHs are finished, the whole order is considered finished and the Order Manager thread stops there.

The current negotiation protocol is naive: it just iterates over all resources until one is available and reserves it immediately. If it cannot find one, it just tries again at the next Order Manager thread loop iteration. This needs to be replaced by a planning (see section 3).

The Directory Facilitator implementation is a cache holding all services associated to all resources (with all the parameters and attributes). The cache needs to be rebuilt each time a new resource is added, or each time a service is made available or is disabled for a resource. On the other hand, interrogating the Directory Facilitator is fast and works reliably.

The forklift is currently implemented as a singleton for the sake of simplicity: there is only one instance of it for the whole system, and the transporter definition in the scenario file is ignored. While it works for our current production line, it will obviously not be suitable for workshops with more than one forklift. It needs to be turned into a resource on its own with a transport service and, of course, a planning. See section 3 for more details.

3 Future prospects

While we reached the goal of the Capstone Project, there is still some work to be done on the SoHMS system to improve it and make it work on other production lines.

3.1 Planning and running ahead

For now, the system works one step at a time: each ROH minds its own business and moves at its own pace. It doesn't *plan* anything, everything is done only when needed and for the current step.

To work efficiently with multiple resources, products and forklifts, we need to change how the system behaves. Each resource will have a planning of services execution, including the transporter. This will allow the system to plan everything by filling in the planning of every resource in advance instead of running the current step only.

We decided that the most appropriate way of representing such a planning in Java would be to have a list of events (each event being a service execution start or end), and another complementary list of gaps. So, each time you'd need to add something to the planning, you just need to look for the gaps table to see where your new event fits in the planning.

We have yet to implement and integrate such a data structure.

3.2 Fix the Directory Facilitator service matching

The current implementation of the Directory Facilitator don't take into account the parameters and attributes of the profiles. It will match a resource to a service even if the attributes don't match. The code responsible for this part is actually present in the code Francisco wrote, but for some reason it doesn't work so we stubbed it out.

3.3 Make the forklift a regular resource

As stated above, the forklift is currently represented as a singleton in the code. This needs to be changed to a resource on its own, with a transport service. The forklift specific move methods will need to be removed, and a generic "use the transport service on the forklift resource" command needs to be used in its place.

This makes the forklift more consistent with Francisco's original design, but more importantly it allows having multiple forklifts in a production line. It is also necessary for forklifts to have a proper planning.

We emitted the idea of interleaving transport service executions between each service in all products recipes. Instead of having S1, S2 then S3, we would have S1, Transport, S2, Transport and S3. This would be the best way of integrating the new forklift mechanism without having to change the whole SoHMS system.

3.4 Have more abstraction on the communication level

The current system can send move and processing commands. If you recall, these commands are tied to our model, and are not generic to all production lines. Once the forklift is made as an autonomous resource, these commands can be removed from the system and replaced by a "execute service `x` on resource `y`" command instead. Our FlexSim communication layer will then translate those back into `MOVE` and `PROCESS` commands, and the communication layer of other production lines will do what they want.

3.5 Remove everything useless and redundant from the code, then clean it up

The current code has a lot of residue from Francisco's production line and SoHMS system implementation. There is a lot of not working, unused, dead or useless code. To improve legibility and maintainability, it would be better to remove all those remnants. This is why we couldn't generate legible class and sequence diagrams out of the code, it's too large.

As we've been told, Francisco did not know object oriented programming when he created the system, he learned as he wrote it. This results in some not-so-elegant code that needs to be cleaned up and improved.

3.6 Use Petri nets instead of lists for products recipes

Francisco originally designed products recipes as being Petri nets instead of simple lists. The code has a "product recipe" interface, in which we implemented our simple list-based recipe class. One could make a Petri net out of this interface, as recipes were originally intended to be.

Petri nets would allow more complicated and flexible recipes: each unit to be produced can take a different path in the recipe depending on the available resources, leading to the same product in the end.

Conclusion

To put it in a nutshell, this project has been a great opportunity as it allowed us to join an already large project and add our contribution to it. We did encounter some difficulties because having to understand a code that several people modified over the recent years while not following a consistent standard was not easy, but we are confident that it will help as this is far from an uncommon situation in a professional setting.

Furthermore, we did our best to write a consistent code that will help with legibility and maintainability. Finally, the context and the future potential applications of this project are very exciting and we are glad we were able to be part of it.

We'd like to thank our supervisors Pascal André and Olivier Cardin for their help and guidance, and Mohammed Tebib for answering all of our questions regarding the SoHMS system and its Java implementation. Finally, we'd like to wish Fawzi Azzi best of luck for his thesis.

Appendix

FlexSim model protocol

Communication

The FlexSim model uses a socket to communicate with the SoHMS system. In the protocol, the model is **client** and the system is **server**. The client will try to connect using the port **1234**, so the server must listen to this same port.

Messages

Each message should be sent by writing it to the socket. Each message must be followed by a newline character (`\n`) to mark them off.

Commands

The **server** issues commands to the **client** by writing a message to the socket. Once the command is received, the **client** will reply back with another message containing the command prefixed by `OK` or `KO` to indicate if the command has been received and understood.

For instance, if the server sends `MOVE SOURCE SINK`, the client will respond with `OK MOVE SOURCE SINK`. However, if `MOOVE SOARKE SYNCH` is received, it will respond with `KO MOOVE SOARKE SYNCH` because the command is invalid (and it will not be executed).

The **client** can also send feedback on a previously issued command. To do this, it sends the command back, prefixed by the appropriate feedback, depending on the command type. All feedback types are documented in the commands specification below.

For instance, when receiving `PROCESS M1 10`, the client will first answer with `OK PROCESS M1 10` to indicate that the command has been received and understood, and then send a `START` feedback like so: `START PROCESS M1 10`. Once processing is done, it will send one last feedback for the command: `END PROCESS M1 10`.

Commands specification

1. MOVE

The `MOVE` command makes the forklift move from point A to point B. If a product is present on point A, it will be taken and put on point B.

Syntax: `MOVE <A> <B>`
Parameters:

- `<A>`: starting point
- `<B>`: destination point
- Points list:
  - `SOURCE` for source point
  - `SINK` for sink point
  - `3` for `M1`
  - `4` for `M2`
  - `5` for `M3`

Feedback:

- `START` when the forklift is gone from point A (and has taken the product if any)
- `END` when the forklift has reached point B (and has put the product if any)

2. PROCESS

The `PROCESS` command makes a given machine process its product for a given amount of time. The meaning is equivalent to executing its only service on the product.

Syntax: `PROCESS <MACHINE> <TIME>`
Parameters:

- `<MACHINE>`: machine which is to process its product
- `<TIME>`: processing time in seconds
- Machines list:
  - `M1`
  - `M2`
  - `M3`

Feedback:

- `START` when the machine has started processing the product
- `END` when processing is over

FlexSim model scenario file

This is the JSON file corresponding to our scenario for the simulated production line. We can see the forklift, source, sink and all three machines, each one having one and only one service. Each product is made by running each service in order. The production order is simple: make five units of the product, without making more than three at once.

```
{
  "servicesOntologies":[
    {
      "name":"Assembling",
      "services": {
        "S1": {
          "id": 1,
          "ontology": "ontology",
          "category": "paint",
          "taxonomy": "sometax",
          "description": "paint a product",
          "parameters": [],
          "attributes": []
        },
        "S2": {
          "id": 2,
          "ontology": "ontology",
          "category": "solder",
          "taxonomy": "sometax",
          "description": "do something",
          "parameters": [],
          "attributes": []
        },
        "S3": {
          "id": 3,
          "ontology": "ontology",
          "category": "assemble",
          "taxonomy": "sometax",
          "description": "do something",
          "parameters": [],
          "attributes": []
        }
      }
    },
    {
      "name":"Transport",
      "services": {
        "S4": {
          "id": 4,
          "name": "Transporter",
          "ontology": "Transport",
          "category": "Transport",
          "taxonomy": "",
          "description": "Takes a product from an Initial Port to a Final Port in the
          ↪  System's Layout Map",
          "parameters": [
            {
              "dataType": "String",
              "name": "StartPort"
            },
```

```
            {

              "dataType": "String",
              "name": "FinalPort"
            }
          ],
          "attributes": []
        }
      }
    }
  ],
  "resources":[
    {
      "type":"SOURCE",
      "name":"SOURCE",
      "technology":"T1",
      "category":"Nothing",
      "description":"do something",
      "inputPorts":[
        {
          "value":"Port1.1"
        }
      ],
      "outputPorts":[
        {
          "value":"Port1.1"
        }
      ],
      "offeredServices":[]
    },
    {
      "type":"RH",
      "name":"M1",
      "technology":"T1",
      "category":"Painter",
      "description":"do something",
      "position": "3",
      "inputPorts":[
        {
          "value":"Port1.IN"
        }
      ],
      "outputPorts":[
        {
          "value":"Port1.IN"
        }
      ],
      "offeredServices":[
        {
          "service":"S1",
          "parameters":[],
          "attributes":[],
          "methods":[
            {
              "processType":"Atomic",
              "id":11,
              "setupId":1
            }
          ],
          "averageCost":20
        }
```

```
        ]
      },
      {
        "type":"RH",
        "name":"M2",
        "technology":"T1",
        "category":"Solderer",
        "description":"do something",
        "position": "4",
        "inputPorts":[
          {
            "value":"Port2.IN"
          }
        ],
        "outputPorts":[
          {
            "value":"Port2.IN"
          }
        ],
        "offeredServices":[
          {
            "service":"S2",
            "parameters":[],
            "attributes":[],
            "methods":[
              {
                "processType":"Atomic",
                "id":11,
                "setupId":1
              }
            ],
            "averageCost":20
          }
        ]
      },
      {
        "type":"RH",
        "name":"M3",
        "technology":"T1",
        "category":"Assembler",
        "description":"do something",
        "position": "5",
        "inputPorts":[
          {
            "value":"Port3.IN"
          }
        ],
        "outputPorts":[
          {
            "value":"Port3.IN"
          }
        ],
        "offeredServices":[
          {
            "service":"S3",
            "parameters":[],
            "attributes":[],
            "methods":[
              {
                "processType":"Atomic",
                "id":11,
```

```
            "setupId":1
          }
        ],
        "averageCost":20
      }
    ]
  },
  {
    "type":"Transporter",
    "name":"AGV",
    "category":"Transport",
    "technology":"Conveyor",
    "description":"An AGV",
    "inputPorts":[
      {
        "value":"Port1.2"
      }
    ],
    "outputPorts":[
      {
        "value":"Port1.2"
      }
    ],
    "offeredServices":[
      {
        "service": "S4",
        "parameters":[
          {
            "dataType":"String",
            "name":"StartPort"
          },
          {
            "dataType":"String",
            "name":"FinalPort"
          }
        ],
        "attributes":[
          {
            "dataType":"Double",
            "name":"Cmax"
          },
          {
            "dataType":"String",
            "name":"Quality"
          }
        ],
        "parametersProfile":[
          {
            "profileParameter":[
              {
                "rangeType":"Catalogue",
                "dataType":"String",
                "name":"StartPort",
                "rangeValues":[
                  {
                    "value":"Port1.4"
                  }
                ]
              },
              {
                "rangeType":"Catalogue",
```

```
                    "dataType":"String",
                    "name":"FinalPort",
                    "rangeValues":[
                      {
                        "value":"Port1.OUT"
                      }
                    ]
                  }
                ],
                "methods":[
                  {
                    "id":1
                  }
                ]
              }
            ],
            "methods":[
              {
                "processType":"Atomic",
                "id":1,
                "setupId":1
              }
            ]
          }
        ]
      },
      {
        "type":"SINK",
        "name":"SINK",
        "technology":"T1",
        "category":"Nothing",
        "description":"do something",
        "inputPorts":[
          {
            "value":"Port3.1"
          }
        ],
        "outputPorts":[
          {
            "value":"Port3.1"
          }
        ],
        "offeredServices":[]
      }
    ],
    "orders":[
      {
        "id":1,
        "numOfUnits":5,
        "priority":"NORMAL_PRIORITY",
        "maxParallelUnits":3,
        "product":1
      }
    ],
    "products":[
      {
        "id":1,
        "name":"pA",
        "services":[
          "S1",
          "S2",
```

```
294             "S3"
295           ]
296         }
297     ],
298     "layoutSpec":[]
299   }
```

Class diagrams

Figure 3.1 is the class diagram of the Java program we wrote. Figure 3.2 is the (simplified) class diagram of the SoHMS framework.

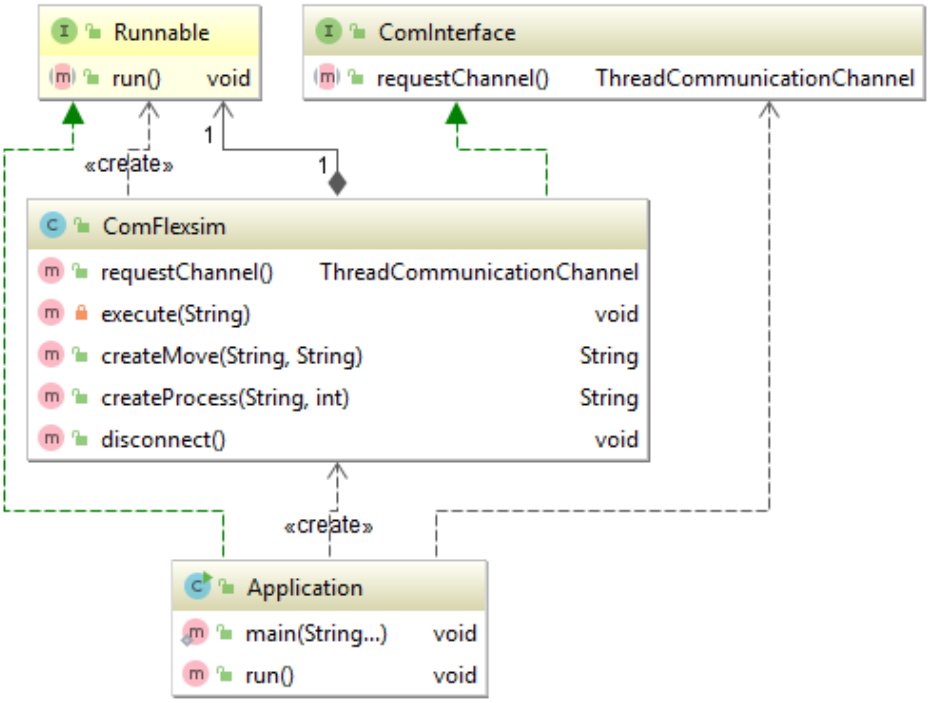

Figure 3.1: Class diagram of the Java program

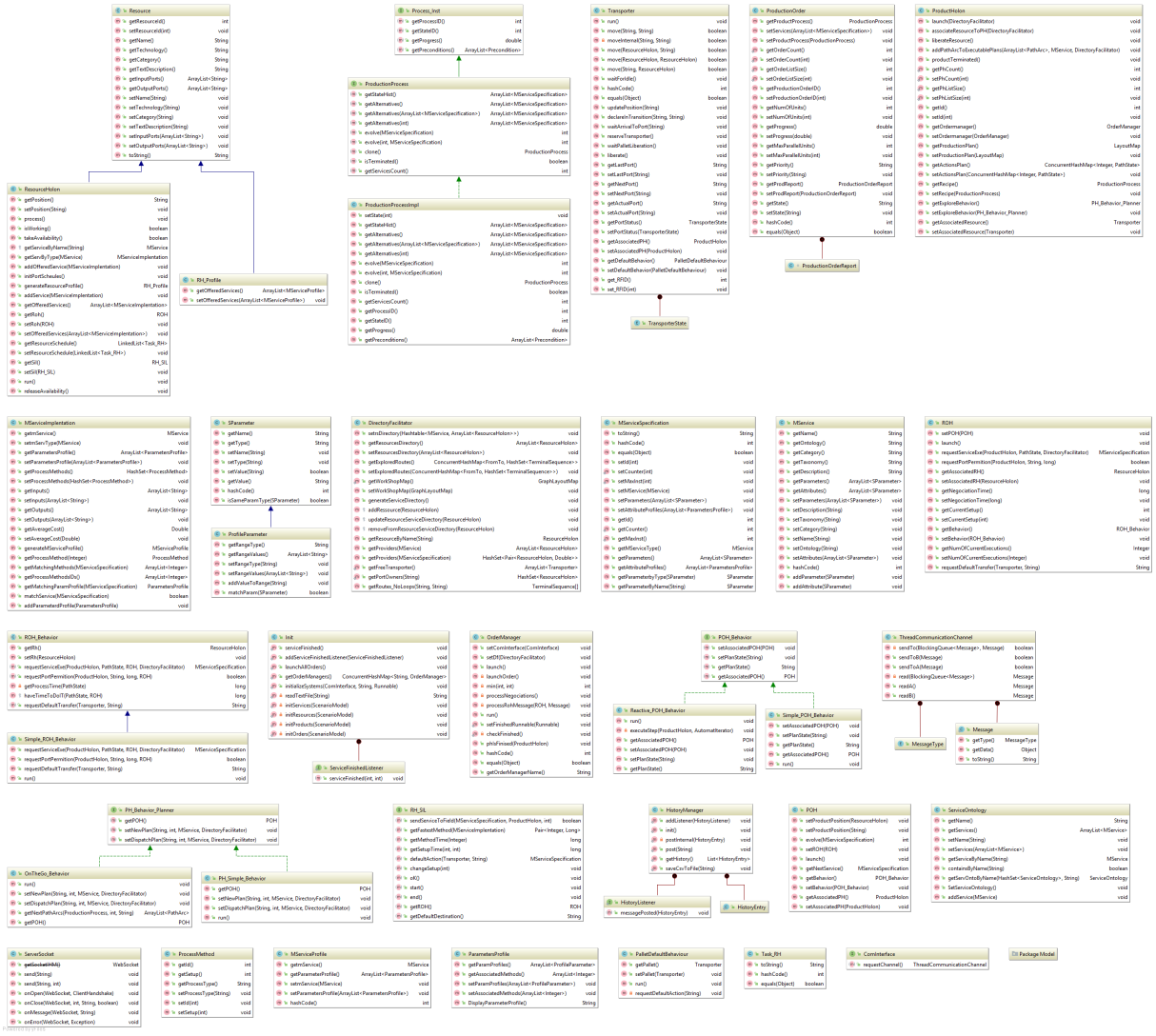

Figure 3.2: Simplified class diagram of the SoHMS framework (better seen at 300% zoom)

Sequence diagrams

Figure 3.3 is the sequence diagram of the method responsible for launching executions of all orders. More sequence diagrams can be generated (initialisation process, OH thread, ROH thread...) but are too large to be legible when included into this document.

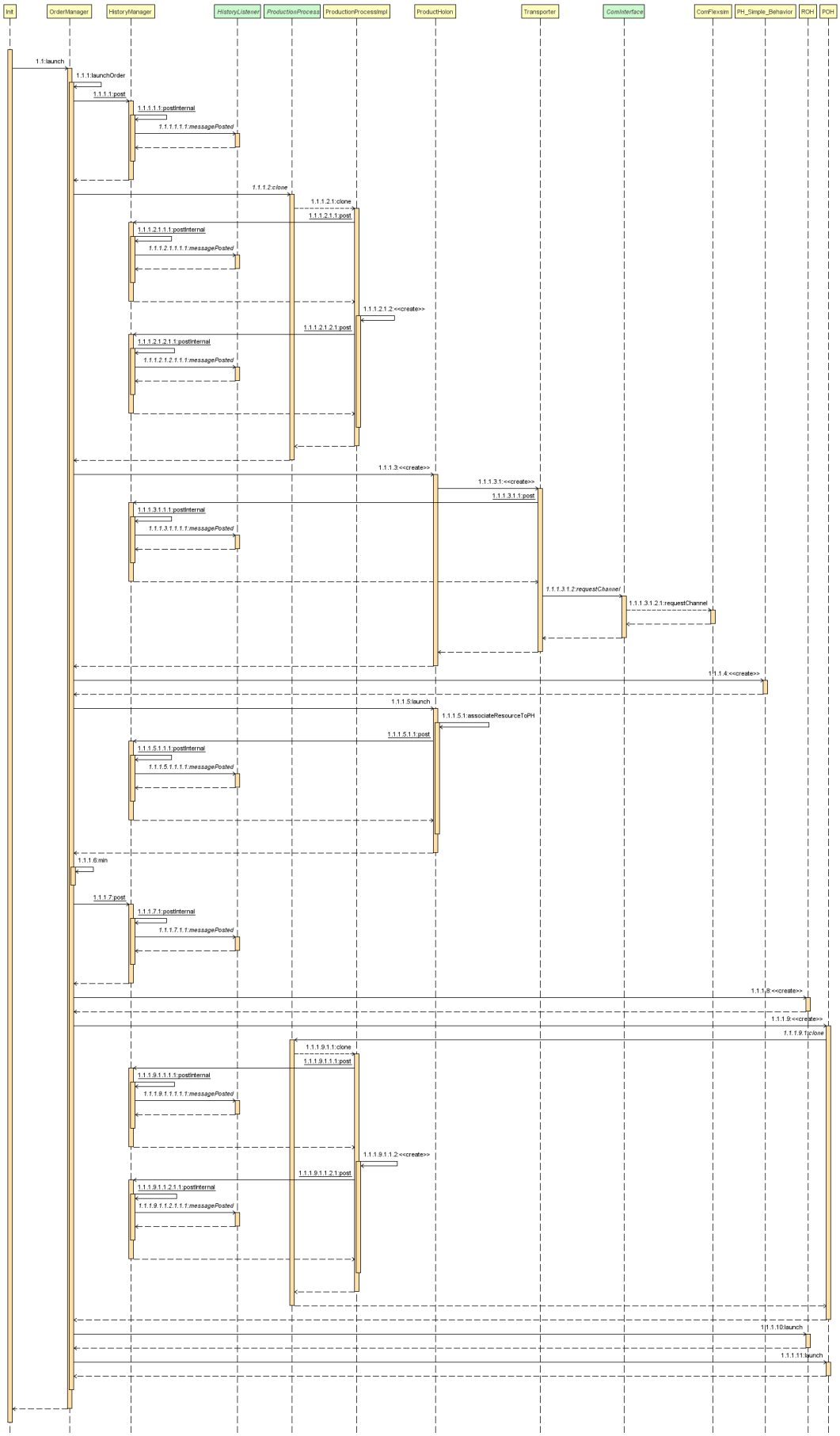

Figure 3.3: Sequence diagram of the method responsible for launching orders