02268 Exam Report: Modelling S-train processes

Markus Brammer Jensen, s183816	Cecilie Krog Drejer, s185032
Ke Yang, s212495	Marcus Møller Pedersen, s213432
João Luís Gonçalves Mena, s223186	–

Screencast: `https://youtu.be/l5RSWFFc8hI`

1 Introduction and Domain Description

In Denmark many people interact with public transport every day to go to work, school, or for leisure activities. Statistics Denmark state that the total amount of trips done using trains in 2021 was 211 million total trips[1]. This goes to show the importance and the reliance many people have on the train services as a mode of transportation. For this reason, we decided that the domain for the project should be public transportation, with a focus on S-train transportation. The reliance on a system like the train transportation system results in having to be as reliable and predicable as possible. This is important for the passengers, but also for the thousands of entities and services that are constantly communicating with each other in real-time or otherwise. Ensuring that all processes are executed in the correct manner in a system like this is detrimental to the functioning of the whole system. Trying to understand the business processes and the events behind a large system that most of us use frequently, if not daily was a large factor when picking the public transport domain. In this report, we try to create some of the business processes that might exist in a system like the Danish S-train transportation system.

2 Processes

The processes identified are sourced mostly from personal experience riding the S-trains on a daily basis. Simple, day-to-day processes like the typical journey of a S-train (Section 2.2) and the standard journey of a customer (Section 2.1) are extrapolations of how the daily commute could look behind the curtains. Other processes, like reserving tickets (Section 2.3), purchasing online (Section 2.7) and ticket inspection (Section 2.4) is a mixture of personal and software development experience. That is, we have experience with the "front-end" of these services and use our knowledge of software systems to model a back-end.

Finally, the processes train maintenance alert (Section 2.5) and train delay (Section 2.6) are educated guesses of how maintenance and more immediate break downs are handled.

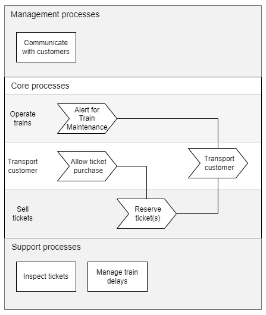
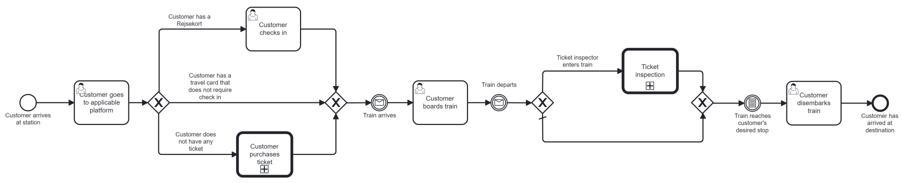


Figure 1: Process Landscape

[1]Statistik Danmark: https://www.statbank.dk/statbank5a/default.asp?w=1536

2.1 Standard Customer Journey

The Standard Customer Journey process was never intended to be executable. This process was initially created to grant us an overview of the key processes necessary for the completion of a customer's journey with an S-train. As can be seen from the sub-processes in the model, we identified both the purchase ticket (Section 2.7) and ticket inspection (Section 2.4) processes.

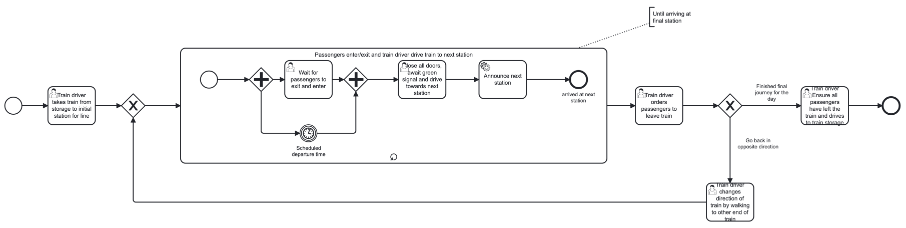


Customer arrives at station
Customer goes to applicable platform
Customer has a Rejsekort
Customer checks in
Customer has a travel card that does not require check in
Customer does not have any ticket
Customer purchases ticket
Train arrives
Customer boards train
Train departs
Ticket inspector enters train
Ticket inspection
Train reaches customer's desired stop
Customer disembarks train
Customer has arrived at destination

Figure 2: Standard Customer Journey

2.2 Typical S-train journey

It is nice when everything runs smoothly. This is exactly the case in the process Figure 3 which depicts a standard journey for a train on a S-train line. The train goes from stop to stop, picks up and deposits passengers, all while following the schedule and respecting signals.



Until arriving at final station
Passengers enter/exit and train driver drive train to next station
Train driver takes train from storage to initial station for line
Wait for passengers to exit and enter
Scheduled departure time
Close all doors, await green signal and drive towards next station
Announce next station
arrived at next station
Train driver orders passengers to leave train
Finished final journey for the day
Train driver Ensure all passengers have left the train and drives to train storage
Go back in opposite direction
Train driver changes direction of train by walking to other end of train

Figure 3: Typical S-train journey

2.3 Reserve Tickets

There are many reasons for having to increase capacity on a train line. Such a reason could be the sudden increase in reserved tickets made up to a train departure. This ensures that the supply of train tickets follows the demand, and helps predict the capacity that is need for the actual departure. In the Reserve Tickets business process, the rate at which the tickets are reserved and the total possible reserved seats are indicators for such an increase in capacity. If there is a high rate of reservation made, then a decision must be made. Either more capacity is added in the form of more wagons attached to the train, another train is added to the train schedule, or no more capacity is created. If there instead is a slow rate at which the tickets are reserved, then the capacity does not increase, and once maximum capacity is reached, no more tickets can be reserved.

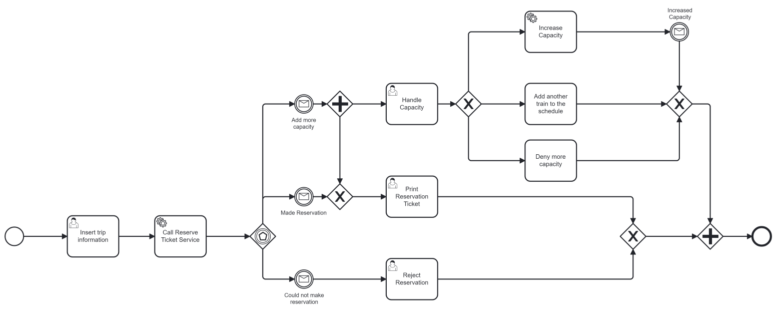


Figure 4: Ticket reserve process

2.4 Inspect tickets

It is important to ensure that the passengers aboard the train actually do have tickets. This is the job of the ticket inspector. For each passenger in the train, the ticket inspector checks whether or not the passenger has a valid ticket. If the passenger does not have a valid ticket, they are handed a fine.

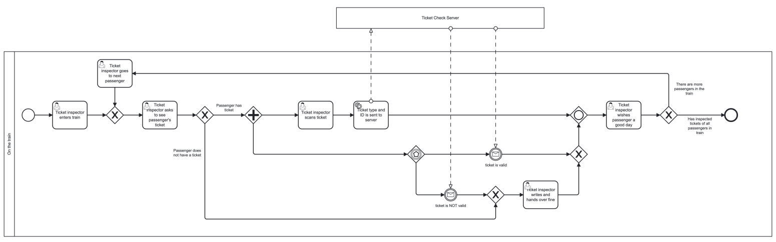

Ticket Check Server
On the train
Ticket inspector enters train
Ticket inspector goes to next passenger
Ticket inspector asks to see passenger's ticket
Passenger has ticket
Passenger does not have a ticket
Ticket inspector scans ticket
Ticket type and ID is sent to server
ticket is valid
ticket is NOT valid
Ticket inspector writes and hands over fine
Ticket inspector wishes passenger a good day
There are more passengers in the train
Has inspected tickets of all passengers in train

Figure 5: Ticket inspection process

2.5 Alert for Train Maintenance

This process is applicable to each individual train during a trip. Siddhi filter and processes the events sent by the Train Sensor Simulator event-source, and depending on the outcome of that processing, sends messages to Camunda to trigger actions that deal with events such as excessive speed, persistent deviations in temperature and/or pressure that indicate the need for maintenance, and event extreme cases where the immediate stoppage of the train is required.

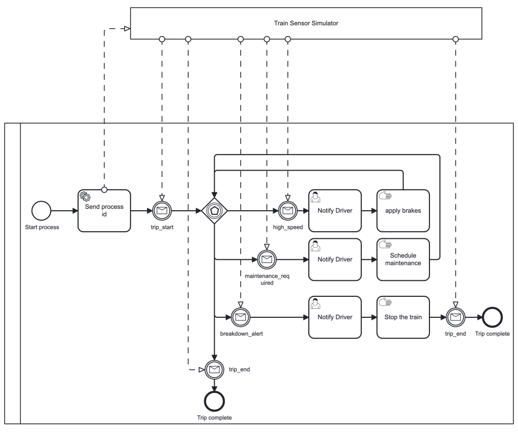


Figure 6: Alert for Train Maintenance

2.6 Manage Train Delays

Should a train be significantly delayed - or cancelled - the system should ideally arrange for replacement buses on the applicable journey automatically. When delays/cancellations are no longer present, the system should also cancel any replacement buses.

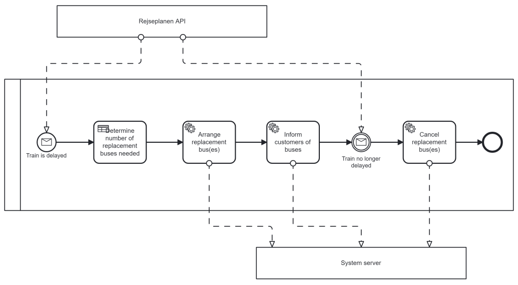

Rejseplanen API
Train is delayed
Determine number of replacement buses needed
Arrange replacement bus(es)
Inform customers of buses
Train no longer delayed
Cancel replacement bus(es)
System server

Figure 7: Manage train delays

2.7 Allow ticket purchase

This process is the business process used to purchase tickets online.When the customer enters the booking system or app, the process begins. If the customer already has a valid ticket, the process ends, if not, the user needs to decide on details such as departure, destination, time and quantity. Then stores the information in the corresponding table and sets the status to pending and generates the order number in the response to be used in the next task. Typically a request for payment will use the API of a bank or other payment system and an order number. We then have a receiving task waiting to hear back from the banking system and once received, change the status of the order in the table to completed. Once this is done, an order completion email is sent to the customer.Ideally, the customer-determination ticket detail task and the order creation task should be able to be extended to the processes in 2.3 to cope with situations such as full carriages. Unfortunately, due to the time and some problems with the Siddhi code, this part was not implemented.

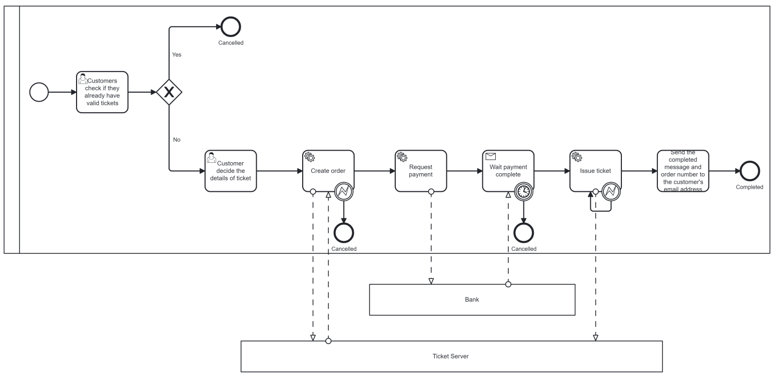

Customers check if they already have valid tickets
Yes
Cancelled
No
Customer decide the details of ticket
Create order
Cancelled
Request payment
Wait payment complete
Cancelled
Issue ticket
Send the completed message and order number to the customer's email address
Completed
Bank
Ticket Server

Figure 8: Allow ticket purchase

3 Events

3.1 Event Sources

3.1.1 Rejseplanen's API

Unfortunately, this event source has not been used in our implementation. It should have initiated the Train delay process (see section 2.6), but due to several issues with communication between Siddhi and Camunda, this process is not executable. In particular, the Message Start Event presented some problems, as variables (such as a train ID) given by the message payload were needed. Testing showed that this could be fixed by using a normal Start Event, immediately followed by a Message Catch Event. However, even with this change, the process resulted in time-out errors when using Siddhi and various vague HTTP errors when using Postman. We were not able to identify the problem within the time constraints of this project.

Ideally, live information from the Rejseplanen API regarding significantly delayed or cancelled trains should trigger the process, wherein replacement buses are arranged and cancelled as necessary and customers are informed of any changes. The implementation would entail live data from the API to Siddhi, wherein it would be determined if the delay is significant or not (in testing anything over 5 minutes was deemed significant).

Nevertheless, we have used the Rejseplanen API elsewhere in a fully-implemented process within the Reserve Ticket Business Process from section 2.3, although the API cannot be considered an event source in this usage.

3.1.2 Train Sensors Simulator

As another source of events for our project, we developed a tool to simulate the transmission of events with data regarding various sensors on a train to be filtered and processed in Siddhi, and later used by our Process Aware Information System, Camunda. This tool simulates a trip, starting with an event that indicates the start of a journey, another one that indicates the end of the journey, and a stream of events containing the train's metrics during the ride, such as the number of passengers on board, the temperature of the brakes, the pressure of the hydraulic system and the speed of the vehicle. The event source can represent, on a smaller scale, an application of Internet of Things systems to event-driven systems. A more detailed description about the development of this tool can be found in Section 4.

3.2 Event Processing

Due to the inability to process the desired events from the Rejseplanen API, the only true event processing that is implemented is regarding the Train Sensor Simulator events. These events are filtered and processed in Siddhi. If all the parameters values are set to 1, the CEP recognizes it as a start event. If they are all 0, it is processed as an end event. More complex filtering is done with the speed, hydraulic pressure and brake temperature parameters, which can trigger courses of action to deal with high speed, required maintenance and immediated breakdown alerts.

4 Implementation and Interaction Between Processes and Events

Camunda is used to implement the processes; Siddhi is used to implement event processing. Both of these tools (Camunda Platform 7 and Siddhi Tooling) are hosted as Docker containers.

Note that because of this hosting setup, in Camunda, the `url` is set to `http://siddhi:7370/checkValid` where `siddhi` is the name of the container. This behaviour is mirrored in the Siddhi code: When Siddhi wants to deposit a message to Camunda Platform 7, the url is `http://camunda:8080/engine-rest/message` where `camunda` is the container name for Camunda Platform 7 instance. For the containers to communicate via their names, the containers must be in a custom Docker network.

4.1 BPMN Implementation

While most of the BPMN implementation is fairly straight forward, this section discusses some of the more notable choices.

While the parallel split in the ticket inspection process (Figure 5) is in theory redundant (see Section 5.1), the inclusive join is an elegant solution. The inclusive join in necessary in this case because of the flow where the passenger does not have a ticket. If the join was a parallel join, then the process would get stuck as it never reaches the scanning of the ticket.

In Figure 3, the main chunk of the process is a looping sub-process. The looping encapsulates the repetition of going from station to station. The loop *could* be included in another loop when reaching the final station but that would potentially make the diagram less intuitive.

Checking all the customers in Ticket Inspection (Section 2.4) could easily be transformed into a loop of visiting all the passengers. The loop could remove the left-most and right-most exclusive gateway in the diagram.

The Standard Customer Journey (see Figure 2) makes use of two Call Activities to the Ticket Purchase process (Figure 8) and the Inspect Tickets process (Figure 5) respectively. Most notably, in order to model the contingent nature of the ticket inspection, the reference to this process is added in an Exclusive Gateway with an empty Default Flow.

The Alert for Train Maintenance process (Figure 6) is initiated with a normal Start Event followed by a Service Task responsible for sending the process ID to Siddhi. This is a workaround in order to make the process executable. In an ideal implementation, this process would be triggered by a Message Start Event matching the implementation of the *trip_start* Message Catch Event.

4.2 Interaction Between Processes (Camunda) & Event Processing (Siddhi)

In Section 2.4, the service task "Ticket type and ID is sent to server" is implemented as a connector. The connector is implemented much like in Tutorial 08.5: The `header` and `method` fields are identical to the tutorial, the `payload` includes the data contained in the form filled out in the previous user task ("Ticket inspector scans ticket"), and the `url` points to a Siddhi end-point.

The ticket form has two fields: Ticket type and ticket ID. The ticket type can be one of two possibilities: "rejsekort" or "ticket". The ID is a unique ID for each ticket. (Note that ticket is the broad term for both "rejsekort" and "ticket" while "ticket" is a special type of ticket).

In Siddhi, whenever a ticket is scanned, it is determined valid/invalid dependant on the ticket type. If the ticket is a "ticket", then the criteria is that that "ticket" has been purchased and thus exists in the database already. If the ticket is a "rejsekort", then the "rejsekort" must exist *and* be checked in. Siddhi sends a message to Camunda's message end-point: `valid_ticket` or `invalid_ticket`.

In Section 2.7, two service tasks have been implemented as connector as a way of interacting between process and Event Processing.One of these is the "Create Order" task. This task takes the start, end, time, number and other parameters provided by the user and sends them to Siddhi as a `payload` in an http request. The corresponding stream for Siddhi stores this data in a table and sets the status to pending. It also receives a response with the order number which is a unique ID.The other is the "issue ticket" task. This task follows a receiving task, which sends an http request to Siddhi with the order number once a successful payment has been received from the bank. The corresponding stream of Siddhi will look up the corresponding order number in the table and change the status to completed.

4.3 Reserve Tickets

The Reserve Tickets business process starts when a reservation request is made in Camunda. This request consists of an origin, destination, time and a date. Then an event is sent to Siddhi with the mentioned information. Siddhi then processes this data by starting with the two locations. These locations, usually names of stations, are sent to the Rejseplanen API where the ids of these locations are then returned to Siddhi. These ids, along with the time and the date, are then sent to the Rejseplanen API to fetch the next departure. Once Siddhi has the correct trip based on the information from Rejseplanen API, it then determines the rate of the reservations made for just this trip. A message is then sent back to Camunda depending on if the rate is high, low and with available tickets or low with no available tickets. If there is a high rate, then a task must be completed on Camunda, determining whether to add a wagon, schedule another train or to ignore the high rate of reservations. If, however, the rate is low and there are available reservations, then a reservation is made. If the rate is low and capacity is full, then no reservation is made.

4.4 Train Maintenance Alert

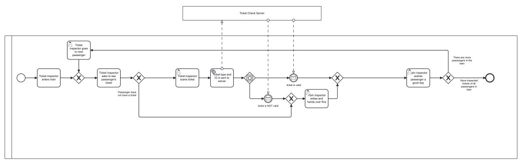
This process uses an event-source which simulates a sensor network in a train that was developed by us. It sends a stream of data regarding the "measurements" of the sensors. These values are set by a pseudo-random number generator tuned with different probabilities for different ranges. The values are also refreshed after a randomly generated interval of time, which gives the system very low predictability and some realism. At the beginning and ending of a trip, the data is sent with identifiable parameters that can be used to process the respective events. The messages sent by the CEP to the PAIS trigger a user task to notify the train driver, which should check a box to acknowledge the notification and is followed by manual tasks meant to deal with the situation identified by the event processor.

5 Practical Implications and Limitations

5.1 Fighting Race Conditions in Camunda

When deploying the ticket inspection process, Figure 5, using Camunda Platform 7, a race condition might appear. The race condition is not with the process itself but with Camunda Platform 7: The process can receive the response message *before* reaching the event-based gateway. Hence the process does not know what to do with the message.

To combat race conditions in Camunda, Figure 5 is a bit more complicated than it could be (see Figure 9): The parallel gateway is there so that the process reaches the event-based gateway *before* the response from the server is received. This is under the assumption that the user task “Ticket inspector scans ticket” is slow to execute. The pools were added to make the diagram more readable but the order of events is not set in stone.



Ticket Check Server
Ticket inspector goes to next passenger
Ticket inspector enters train
Ticket inspector asks to see passenger's ticket
Passenger does not have a ticket
Ticket inspector scans ticket
Ticket type and ID is sent to server
ticket is valid
ticket is NOT valid
Train inspector writes and hands over fine
Train inspector wishes passenger a good day
There are more passengers in the train
Have inspected tickets of all passengers in train

Figure 9: Ticket inspection with Camunda race condition

5.2 Rejseplanen API

Accessing Rejseplanen and using it as an event source was not possible for this project, as the Rejseplanen API can be mostly defined as a REST API, so having events to subscribe to was not possible, at least not with our approach. This meant that we had trouble using Rejseplanen as an external event source.

6 Organizational Aspects

6.1 Responsibilities Distribution & Project Management

Marcus Pedersen (s213432) has been in charge of the Reserve Ticket BPMN and the Siddhi functionality. He has also created the Siddhi functionality for ticket inspection.
João Mena (s223186) has been in charge of the Train Maintenance Alert BPMN and the respective Siddhi functionality. He has also developed the Train Sensor Simulator event-source.
Markus Brammer Jensen (s183816) has been in charge of Ticket Inspection and Typical S-train Journey. Focused a lot on communication between Camunda and Siddhi. Did a lot of the initial process identification.
Cecilie Krog Drejer (s185032) has been in charge of the Standard Customer Journey and Manage Train Delays BPMNs along with the partial Siddhi functionality of the latter. She has also been responsible for initial drafts of most BPMN models.
Ke Yang (s212495) has been in charge of the Ticket purchase BPMN and the respective Siddhi functionality.

6.1.1 Workload Distribution for the Report

	s183816	s185032	s212495	s213432	s223186
1 Introduction	0.0	0.0	0.0	1.0	0.0
2 Processes	1.0	1.0	1.0	1.0	1.0
3 Events	0.0	1.0	0.0	0.0	1.0
4 Implementation	1.0	0.5	1.0	1.0	1.0
5 Implications & Limitations	1.0	0.0	0.0	0.5	0.0
Average	*0.6*	*0.5*	*0.4*	*0.7*	*0.6*