System Integration - Assignment 1

1 Introduction

In today's world, IT has become a ubiquitous part of every person's life. With a smartphone in everyone's pocket, it seems like a perfect opportunity for businesses to utilize this hardware. One common place where this utilization can be seen is when one is doing grocery shopping where it is now possible for the customer to scan the groceries and pay through their phone. Netto, a Danish supermarket franchise, has achieved this through their *Netto Scan and Go* app. And this app is what has been chosen to model in this report.

The system was chosen due to its wide usage, and because we, as a group, can also be considered an actor since we also use the app. It is interesting and exciting to see how far technology has come in the past few years. Going into a supermarket, scanning the barcode with your phone's camera, and paying with the same phone was not something most people imagined just a few decades ago.

Despite the group being considered an actor as well, some deviations to the exact modeling of the system have of course been made. The main reason is that it is not publicly available how the app is implemented. But through observations, personal usage, and common knowledge within IT, a possible proposal of the design has been made.

2 Ethnography

Article [1], recommended in the assignment notes, offered us the possibility of choosing between conducting a participant observation or a structured observation. We discussed the feasibility of one method versus the other and the outcomes are presented below.

Participant observation
Pros:

1. More direct involvement in what it means to use the app, how it works, and how it changes the customer experience.
2. Direct observation of an example user of the app in the chosen environment.

Cons:

1. More time-consuming than structured observation.
2. It would take a lot more effort to gather a wider range of data.
3. More disruptive method (stopping users and asking questions) and therefore potential lower response rate.

Structured observation
Pros:

1. Faster and easily shareable between people, giving a wider range of results.
2. Time efficient.
3. Response data is already structure-oriented (i.e. aggregated in charts).

Cons:

1. Less user involvement.
2. Almost always people will not provide further details about their experience, they'll just fill out the form as fast as they can.

3 Results of the observations

Since the majority of our group is linked to DTU facilities, we realized that relying on the participant observation methods might limit our insights to mainly student viewpoints. Understanding the importance of capturing a wider spectrum of perspectives, we recognized the necessity to diversify our participant pool not only in terms of age but also in terms of occupation. Without incorporating individuals from various professions, our data could lack the richness and diversity needed for a comprehensive analysis. After careful consideration, we agreed to adopt a structured observation via questionnaire, allowing us to gather insights from individuals across different age groups and occupations. This inclusive methodology ensures that our research encompasses a broader range of experiences and perspectives, ultimately enhancing the depth and validity of our findings.

We chose Google Forms for our questionnaire because it conveniently organizes responses into charts, making it easier to interpret and analyze the data. Our questionnaire includes questions aimed at gathering insights on various aspects of app usage and satisfaction. The questions are:

- age;
- occupation;
- frequency of app usage;
- when do you use the app the most;
- how much has the app improved your shopping experience;
- how often do you have problems with the scanning;
- how often do you have problems with the check-out;
- are you overall satisfied about the app.

We intentionally kept the questions broad and non-technical to encourage participation from a diverse range of individuals. Our goal is to collect personal experiences and opinions about the app in a user-friendly and accessible manner. Currently, an observationpool of 30 respondents was involved in the data collection process. We've also shared the Google Form within the WhatsApp community of DTU to further expand our reach and gather a more comprehensive set of responses. This approach allowed us to engage with a more heterogeneous audience and obtain valuable insights into user experiences with the app.

Results of the observations More than half of the collected data belongs to young people under 30.

The answers to the second question reveal how our goal of diversifying the data was achieved: only 40% of the answers belong to students.

Taking a look at the frequency of use of the application and the conditions that lead the user to use it, we noticed that there's a solid 30% of users that never used the app. This could be linked to two factors:

1. Age-technology usage relation: Usually older people tend to not use new technology.
2. Impossibility to get the app from several mobile stores on Android devices or due to language barriers for non-danish people.

Moreover, from our observations it has been found that participants have a little more trouble with the scan function rather than with the checkout process/payment.

Aside from the participants who never used the app (due to whatever reason), we agree that there's a general sense of satisfaction towards the benefits the app brought.

In figure 1 the detailed results of the observations are presented.

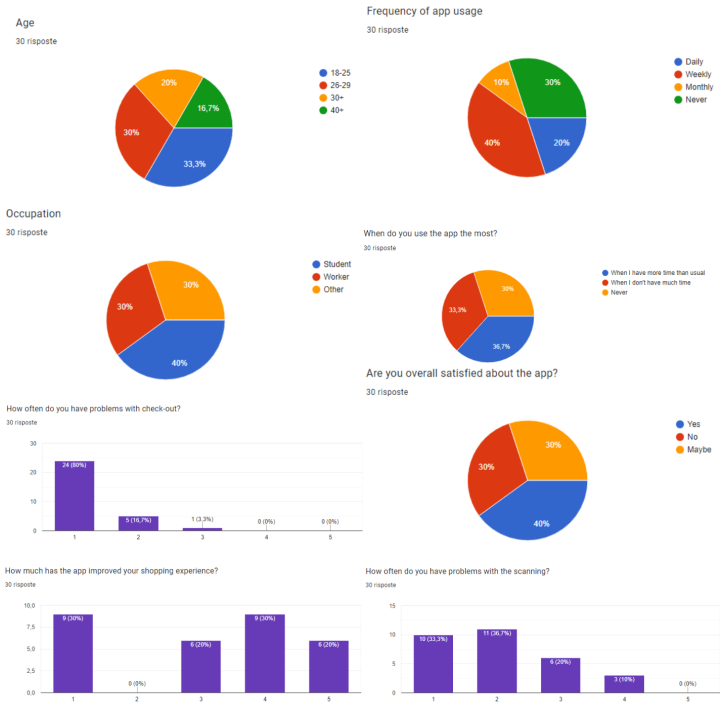

Fig. 1. Results of ethnographic observations

4 Requirement Models

To describe how the different actors, goals, and activities are decomposed in our chosen system, we developed an ArchiMate model, which can be seen in figure 2.

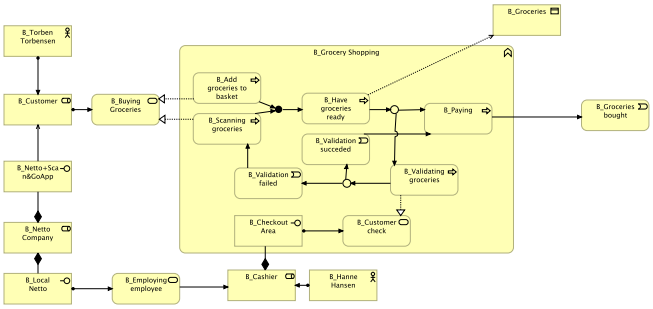

Fig. 2. Archimate Model describing how the different actors, goals, and activities are decomposed.

It can be seen that there are the two main business roles `B_Customer` and `B_Cashier` which are the two main parties involved in our chosen system. To give an example we have assigned two concrete actors to these roles `B_Torben Torbensen` and `B_Hanne Hansen` respectively. The main business interfaces are `B_Local Netto` which represents the supermarket where the customer can use the Netto Scan and Go App, the Netto Scan and Go App itself `B_Netto+Scan&GoApp`, and the checkout area where the cashier works `B_Checkout Area`.
The business role `B_Netto Company`, which represents the Netto company itself, is composed of two business interfaces, the actual app `B_Netto+Scan&GoApp` and the local supermarket `B_Local Netto`, since without the overall company both of these interfaces (app and local supermarket) would not exist. `B_Local Netto` is assigned to the business service `B_Employing employee`, triggers the business role `B_Cashier`, since the local Netto employs the cashier. The Netto app serves the customer business role, as it can be used for the actual grocery shopping.
The main business function is `B_Grocery Shopping`, since this is the overall process that we want to model — that is, grocery shopping through the use of the app. To realize this, `B_Customer` is assigned to the business service `B_Buying Groceries`, which itself is realized by the two business processes `B_Add groceries to basket` and `B_Scanning groceries`. Despite an `and` junction being used it is important to notice that one process can happen without the other as a customer can forget to scan an item but still place said item into their basket and vice versa. The next step is the business process `B_Have groceries ready`. To account for the groceries themselves we have added the business object `B_Groceries`, which the business process `B_Have groceries ready` accesses.
After having the groceries ready, the customer either gets checked (by a cashier)

or pays. This validation of the scanned items against the actual items in the basket of the customer is not conducted each time, but rather happens at random times, whenever the system thinks that a validation is needed. We model this by having the business process `B_Have groceries ready` trigger either the business process `B_Paying` or `B_Validating groceries` with the help of an `OR` junction.
When looking at the case of a required validation we can have two different outcomes, which are modelled as business events. Either the scanned items match the items in the basket of the customer, or they don't. We model this by having the business process `B_Validating groceries` trigger either the event `B_Validation failed` or the event `B_Validation succeeded` with the help of an `OR` junction. The business process `B_Validating groceries` is realized by the business service `B_Customer check`, to which the interface `B_Checkout area` is assigned to. The earlier mentioned business role `B_Cashier` is composed of this interface, since without an employee there is also no checkout area (the physical area would exist, but the function of the place would not be there, and thus we would not consider it an actual checkout area anymore).
If the validation has failed then the customer has to scan the missing items, which is modelled by having the business event `B_Validation failed` trigger the business function `B_Scanning groceries` again. From there the process then continues as described, as the business process `B_Add groceries to basket` is still completed from earlier. If the outcome is a successful validation, then the business event `B_Validation succeeded` triggers the business function `B_Paying`, which is also triggered if no validation was needed. This business function then triggers the final business event `B_Groceries bought`, which is also the goal of the customer journey.

5 Enterprise Architecture Models

The application layer represents the functional aspects of the business layer and any form of software that is used in the process. In this scenario there are 3 main modules, 2 directly linked to the Netto App and one more.

1. `A_PayingModule`: Handles the payment a customer would do for the products, and it is required by `A_AppUI` since the payments are completed there. The module has the `A_PaymentHandler` which realizes the `A_PayingService` to enable the customer to accomplish `B_Paying`.
2. `A_ScanningModule`: It has two important functions, being able to scan which is the `A_BarcodeProcessor` and knowing what is scanned and retrieving the product that was scanned which is the `A_DigitalBasketHandler`. These functions are both required for the `A_AppUI` as well for the customer to scan the items, represented by `B_ScanningGroceries`, to be added to the digital basket to later be paid.
3. `A_ValidatingModule`: It also has two important functions, the `A_DigitalBasketRetriever` and the `A_ValidationDocumenter`. They make sure that the purchase can

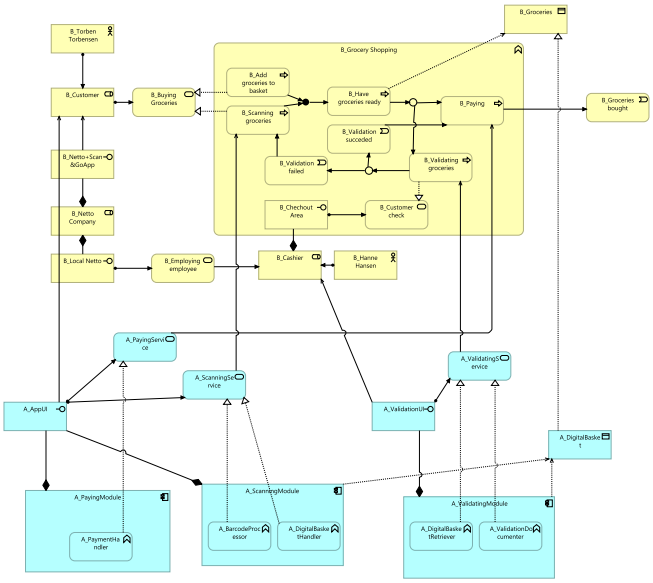

Fig. 3. Archimate Model showing Business and Application layer with the connections between them.

be validated and that the purchase will be stored respectively. This module does not have any connection with the App itself but rather with the `A_ValidationUI` that the `B_Cashier` would need to use to validate the groceries at checkout `B_ValidatingGroceries`.

The technology layer's purpose is to support the application layer to be able to execute its functions and to provide the components necessary. For the Netto App it represents the back-end and hardware, providing all the processing and data that the Application Layer might require and also where it will happen.

This technology layer can be split into three nodes and what they enable:

- `T_Server-Node`: Holds the app backend and computing through a CentOS server.
- `T_Database-Node`: Contains all the data for the products, like price barcode, purchase history and also history of the products as they have changed over time, ran on an Ubuntu server with a MySQL database.
- `T_End-Device-Node`: Represents a user's device where the Netto App would be run and used for scanning grocery products.

The technology layer as it handles the back-end of the Netto App, needs connection between its Nodes directly or indirectly. The main back-end elements are the database and the server which are connected by an internal LAN `T_Netto-Internal-Lan` for the `T_Server_Node` and the `T_Database-Node`.

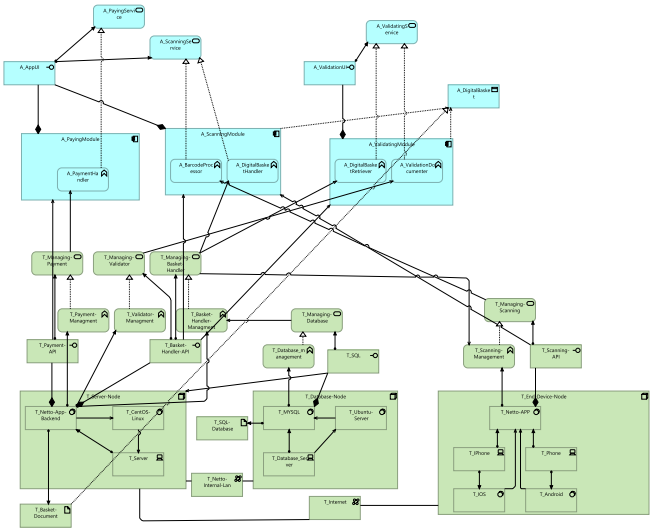

Fig. 4. Archimate Model showing the Application Layer and Technology Layer to showcase them and the connections within them.

`T_Database-Node` provides the server product data to process and T_Server-Node provides data to store to the Database. The end device communicates with `T_Server_Node` over the internet `T_Internet`.

- `T_Managing-Payment`: With the `T_Payment-API` interface assigned to it from the server node it serves the `T_PaymentHandler` function for the `A_PayingModule` in the Application Layer.
- `T_Managing-Validator`: Provides the data necessary for the cashier to verify the purchase by the customer using the Basket-Handler-API interface which has all product data and then serves the `A_ValidationDocumenter` function.
- T_Managing-Basket-Handler: Also uses the T_Basket-Handler-API interface, but this uses the T_Scanning-Management function and the T_Basket-Handler management function which enables getting the product based on the scanned barcode. This then provides the information to the `A_DigitalBasketHandler` from `A_ScanningModule` and the `A_DigitalBasketRetriever` from `A_ValidatingModule`.
- `T_Managing-Scanning`: Uses the `T_Scanning-API` interface from the Netto App to be able to enable the `A_BarcodeProcessor` function
- `T_Managing-Database`: With and SQL interface as its code language for handling the data, it provides the data to the `T_Basket-Handler-Management` function.

6 Workflow Models

6.1 Overview

With Archimate or iStar not providing a formal verification technique indicating when a process is going to terminate and whether all tasks can be executed, a Workflow model is needed to analyze this. The Archimate model designed in this assignment was translated into a Workflow Net, which was used for its verification and execution. A Workflow Net is further defined as a Petri Net that models a workflow process definition, i.e. the life cycle of one case in isolation. The Petri Net involves Transitions (T) and Places (P) which define the structure of the net. All structures have a token or marker present, which defines the behaviour of the net. It also has a representation of Flow relation (F) which shows the linkage between places and transitions in a structure. A Petri Net (N) is then said to be made up of a triple (P, T, F), where:

- P is a finite set of places
- T is a finite set of transitions
- $F \subset (P \times T \cup T \cup P)$ is a flow relation

6.2 Petri Net Model

The Petri Net model created was based on two actors from the use of the Netto+Scan&go App. These two actors are in the form of a customer and a cashier. The former makes purchases and the latter validates the purchases made by the customer which serve as a basis for the Archimate model from which these workflow models were created. The two workflow models created are:

Customer PetriNet
Figure 5 describes the workflow from the customer's point of view.

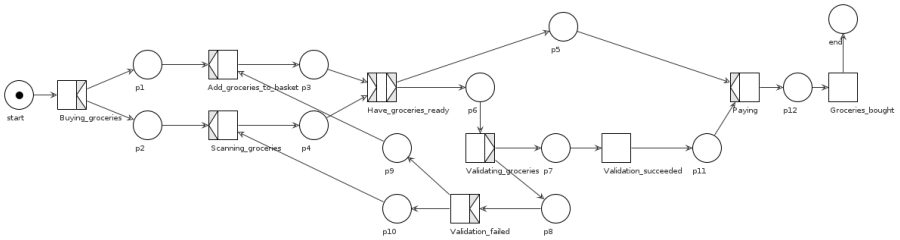

Fig. 5. Workflow of Customer

The model starts with a token in the starting place `start` which enables the first transition `Buying_groceries` which serves as the initial task the customer performs when grocery shopping. It then fires through to the next two

places `p1` and `p2`, which then enables the transitions `Add_groceries_to_ basket` and `Scanning_groceries` which represent the customer having to perform both actions in order to fire to the next transition `Have_groceries_ ready`. This transition occurs after the `AND` junction, leading to the merging of the transition and splitting based on the next transition. From which the customer would have to either perform payment directly or go through a validating process in the form of a `validating_groceries` transition which when enabled can be split up using an `XOR` split enabling either `validation_succeeded` making it possible to also enable the `paying` transition. When the `paying` transition is enabled it leads to the end of the workflow which is the made possible when `Groceries_bought` is enabled to the output place `end`.
Suppose the validation process fails for the customer. It is then seen to have the `validation_failed` transition enabled, which then fires to enable the start of the buying of groceries procedure by enabling both `Add_groceries_to_basket` and `Scanning_groceries` transitions again.

Cashier PetriNet
Figure 6 describes the workflow from the cashier's point of view.

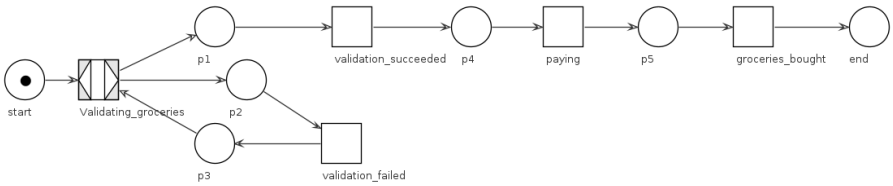

Fig. 6. Workflow of Cashier

This model also starts with a token in the starting place `start` which enables the first transition `Validating_groceries` which serves as the initial task the cashier performs towards the customer. The transition has an `XOR` split characteristic which then splits into whether the validation process is a success or failure. When the groceries are said to be validated successfully, the `Validating_succeeded` transition is enabled, from there the process is fired through the `paying` transition and `groceries_bought` transition. They represent the payment and finalizing of groceries leading to the place-end which is the output and end of the process.

In case the validating of groceries fails, the cashier would have to start the validating process again. From this, the `validation_failed` transition is enabled and the transition process of `validating_groceries` is enabled once again.

6.3 Soundness

When executed, both workflow models passed the soundness requirements which are:

1. Each node in the process model being on a path from the initial node to the final node. This clearly showed that for every reachable point from the beginning there exist a firing sequence until the place-end which is the output.
2. The process showed that when it terminates there was a token in the place-end and all other places were empty. The output was the only state reachable from the place-start that had at least one token in place.

7 Interaction Models

To describe how a user may use the system, and how the different actions may interact, a CCS model has been constructed. The model details, both actions and states that a given user can be in or take. These states for the general tasks of buying groceries are:

- Initiate buying groceries
- Scanning groceries
- Adding groceries to basket
- Validating groceries in the basket
- Being ready to pay for groceries
- Finishing purchase

For each of these one or more related actions can be taken, as table 1 shows.

State	Related Action
buyGroceries	initiates sequence
addGroceries	the act of adding groceries
scanGroceries	the act of scanning a selected grocery
haveGroceriesReady	when the customer is finished shopping
pay	pay for the scanned groceries
validateGroceries	validate if basket reflects scanned items
fail/succeed validation	fail or succeed an validation
completeGroceries	finish shopping

Table 1. Table showing states and related actions for the CSS model.

The model 7, depicts the complete overview of all the available actions.

Model 7 depicts the process of buying groceries, using the Netto App. In the app, users can select to take action of buyGroceries, which in the initial state is the only available. From there the user can add groceries to their basket and/or scan it. As a user, you might not want to scan immediately and might prefer to scan at checkout, which is why the model allows for adding groceries

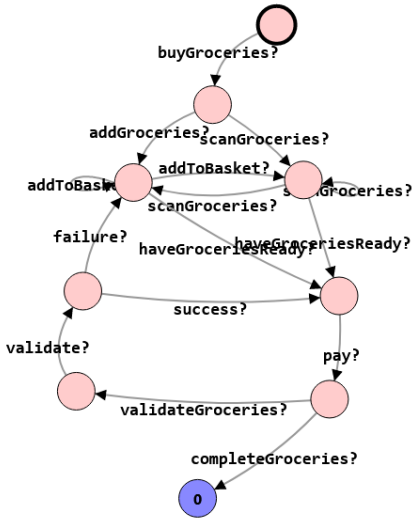

Fig. 7. Complete CCS model

to the basket without being required to scan. In another way, a user might take a grocery and scan it before putting it into their basket, so this gives the best depiction of what a user might do.
After scanning or adding the groceries, the user can select that they have their groceries ready, which translates to the user having found everything they need, and is ready to proceed with payment. But on a random occurrence, the user can be directed to the validation page instead of the actual payment platform, which is where the validateGroceries comes into play. Here an employee can validate the groceries, and approve or refuse the purchase, requiring the user to scan the groceries. After scanning, the user is allowed to complete payment, and potentially face another validation.
In conjunction with this model, two customer scenarios were created, one for validation success and one for validation failure ,both of which can be found in appendix A lines 27 and 29. The success scenario sees a customer scanning and adding groceries to their basket, having the groceries ready, and paying without hindrance. The second scenario depicts the customer failing a validation, having added a good without scanning, and after having their groceries ready, failing validation and having to rescan before completing their purchase.

8 Threats to Validity

In general, we consider our models close to reality. There are however three limitations that need to be considered:

1. **Limited amount of questionnaire responses**
   Since the amount of time for the assignment was limited, we were only able to gather a limited amount of insights from our questionnaire in the ethnographic research. This means that the insights we gathered from this research are not statistically significant and should thus be taken with a grain of salt.
2. **Simplification of payment process**
   In reality the payment process in the checkout process is rather complicated and involves at least one or more third-parties (e.g. payment provider, customer's bank etc.). Since Netto itself has no direct influence on these parties, and modelling the whole payment flow with all involved parties would be rather complex, we have decided to simplify this in our models. Instead of modelling the full complexity of the payment step, we have decided to only include the payment step itself and assume that everything else is hidden behind this step, and works flawlessly without any errors. In reality this would of course not always be the case (e.g. Customer's card gets declined, invalid payment details etc.). We however argue that these cases do not occur often and additionally do not change much about the processes that we have modelled, and thus it is a fair trade-off to leave the exact details out.
3. **Assumption of technical details**
   The most important limitation is the fact that the Netto Scan and Go system is not transparent, and only business processes in the store are observable. The technical processes are however not observable and Netto does not provide any details on this, most likely due to them being a business secret. This however means that we had to make assumptions regarding the exact technical details and thus also for example for the ArchiMate Application and Technology View. We believe that the assumptions we made align well with what can be considered standard in such an IT system, but do not have any proof of this being actually the case for the Netto Scan and Go system. Therefore, it should be considered that parts of the models, especially of the Archimate Application and Technology Layer might look different in reality.

References

1. How to use ethnographic methods & participant observation, https://www.emeraldgrouppublishing.com/how-to/observation/use-ethnographic-methods-participant-observation.

Appendix

A Interaction model

```
1  // Netto scan and go example
2
3  NettoApp := buyGroceries? . (addGroceries? . ScanGroceries +
       scanGroceries? . AddToBasket)
4
5  AddToBasket := scanGroceries? . ScanGroceries + scanGroceries?
       . AddToBasket + haveGroceriesReady? . ReadyToPay
6
7  ScanGroceries := addToBasket? . AddToBasket + addToBasket? .
       ScanGroceries + haveGroceriesReady? . ReadyToPay
8
9  ReadyToPay := pay? . (completeGroceries? . 0 +
       validateGroceries? . ValidatingGroceries)
10
11 ValidatingGroceries := validate? . (success? . ReadyToPay +
       failure? . ScanGroceries)
12
13 // Customer story success
14 CustomerSuccess := buyGroceries! . addGroceries! .
       scanGroceries! . addGroceries! . scanGroceries! .
       haveGroceriesReady! . pay! . completeGroceries!
15
16 //Customer story failure
17 CustomerFailure := buyGroceries! . scanGroceries! .
       addGroceries! . addGroceries! . haveGroceriesReady! .
       validateGroceries! . validate! . failure! . scanGroceries!
        . pay! . completeGroceries!
18
19
20 ////////////////////////////////////////////////////////////////

21 // Examples of initial processes: uncomment only one
22
23 // Initial process for the netto scan and go app
24 NettoApp
25
26 // Customer senarios
27 //(NettoApp  | CustomerSuccess) \ {buyGroceries, addGroceries,
       scanGroceries, haveGroceriesReady, pay, completeGroceries
       }
28
29 //(NettoApp | CustomerFailure) \ {buyGroceries, scanGroceries,
       addGroceries, addGroceries, haveGroceriesReady,
       validateGroceries, validate, failure, scanGroceries, pay,
       completeGroceries}
```

B Overview of provided source code

Alongside the report we submit also the relevant source code for the models. Table X gives an overview of the model name, and the respective source code file and tool that was used.

Model	**Source Code File**	**Tool**
ArchiMate Model	`NettoApp.archimate`	https://www.archimatetool.com/
Petrinet Models	`petrinet_cashier.pnml, petrinet_customer.pnml`	https://woped.dhbw-karlsruhe.de/
CCS Model	`CCS_Netto.txt`	https://pseuco.com/

Table 2. Overview of source code and respective used tool for each model.

System Integration - Assignment Part 2

1 Process Description

The model description provided in the first assignment has been clearly written using an imperative narrative: every goal or activity described in the system has a precise set of actions that need to be done in a certain order to achieve the expected result. Using a declarative narrative, the same model would be described as it follows.

Every time a person enters Netto, they can use the Scan and Go app to put in their basket whatever they need to buy. If an item is scanned, then it needs to be put in the basket. After the scanning process, every user must mark their shopping basket ready for validation and every time a basket is marked that way, it is not possible anymore to abort the process until the validation is done. Not every basket may need to be validated, but every basket has to be marked as ready before proceeding to the validation process. If no validation is needed, the user can proceed to pay for their items and mark the purchase as done. If validation is needed, the basket will be checked by the cashier and the outcome may be successful or failed. After a successful validation, the user can proceed directly to finalize the purchase as before. In case of a failed validation, the user needs to fix their basket by re-scanning and re-validating the missing items. After fixing their basket, they can proceed to finalize the purchase as before. The payment operation can only happen once.

2 Timed Behaviour considerations

The following is an excerpt of the model description from the previous assignment: *In today's world, IT has become a ubiquitous part of every person's life. With a smartphone in everyone's pocket, it seems like a perfect opportunity for businesses to utilize this hardware. One common place where this utilization can be seen is when one is doing grocery shopping where it is now possible for the customer to scan the groceries and pay through their phone. Netto, a Danish supermarket franchise, has achieved this through their Netto Scan and Go app. And this app is what has been chosen to model in this report.*

Based on the model description it is clear that in order for customers to not be furious during their shopping, the *Netto Scan and Go* app must respond within a reasonable time. That is (1) when scanning groceries, but also (2) when paying. Additionally, as identified during the first assignment, (3) a cashier in Netto might also sometimes need to validate said customer to ensure that they have not forgotten to scan anything or perhaps scanned something twice. But a

customer is not willing to wait forever, so that process must also happen within a reasonable time frame. That flow, with those real-time behaviors, will be modeled using the modeling tool UPPAAL.

As a result of the three identified real-time behaviors above, an Uppaal implementation has been developed. The results of the development are divided into three subsets representing the business process itself in figure 1, the customer in figure 2, and the cashier in figure 3. The entire set of global declarations used can be found in Appendix A.

Starting from Figure 1 one can see the flow of the "buying grocery" process using the Netto Scan & Go app. At the top of the figure, a node named `Buying_Groceries` with the marking of `initial` represents the start of the entire process seen from the app's perspective - an abstraction on how the users enter the app has been made. The node has an invariant on 300 meaning that the `clock t` cannot exceed 300 time units without the process making a transition. And due to the two edges going to and from the same node not resetting the clock, it is impossible to stay in the initial node forever. That has been modeled due to the real-world use-case behavior of a customer who will not stay in the store forever. The two edges going to and from the initial node also have a `channel` each which synchronizes with the customer, represented in Figure 2, on said channel. One for each behavior; a customer 'forgetting' to scan an item and only placing it into the basket, and a customer who both scans and adds the item to the basket. For each channel, global variables are updated accordingly to reflect the behavior, with the `add_to_basket_only` channel only increasing the `groceries_amount` and the `scan_add_to_basket` channel both incrementing the scanned items and the groceries. To further ensure that the model does not reach a state space explosion by reaching the maximum possible value for an integer a guard has been set on both edges with a maximum of 200 groceries which seems more than reasonable for a store like Netto. Additionally, in the use case of misusing the application and not scanning the groceries, an additional guard has been set to ensure that the customer scans at least one item.

When the flow moves to the node `Having_Groceries_Rdy` the `clock t` is reset and a bool variable `rdy_to_pay` is set to true. From there, the customer, modeled in Figure 2, can either be picked for validation or not. That has been modeled as a template parameter for the `BuyingGroceries` template, and the value is defined in the `Systems Declarations` of UPPAAL. A detailed figure can be found in Appendix B for the template parameter and Appendix C for how to instantiate a template with a parameter. If the `validate_customer` is false the `customer` must synchronize with the `BuyingGroceries` on the `pay` channel and end the flow when reaching the node `Groceries_Bought`.

If the `validate_customer` is false on the other hand the flow moves to the intermediate state between `Having_Groceries_Rdy` and `Validating_groceries`.

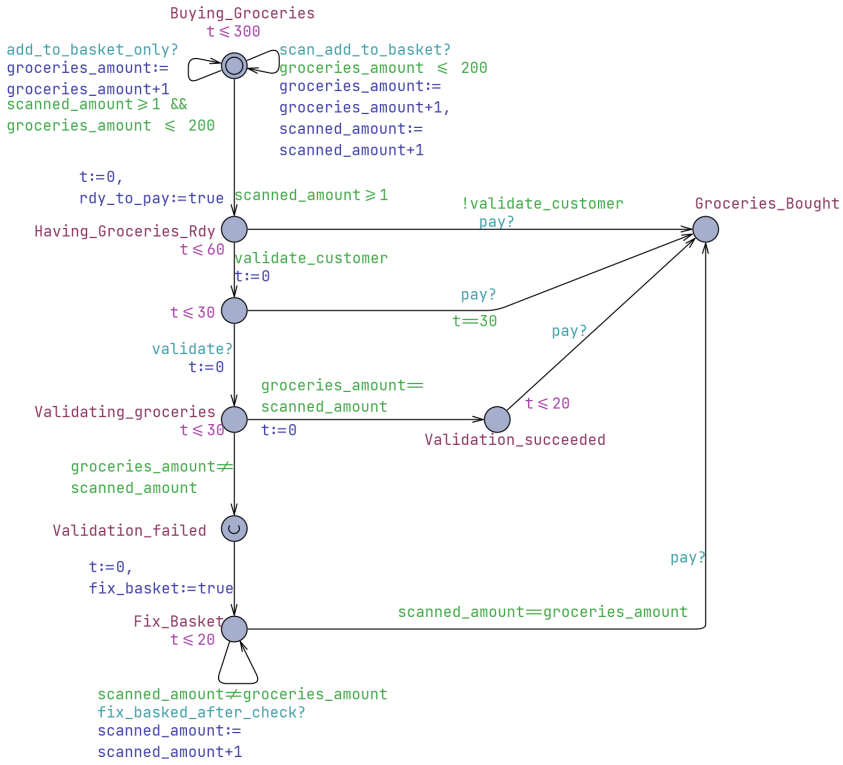

Fig. 1. The Uppaal implementation of the main business flow; buying groceries

Here, action must happen within, including, 30 time units specified by the invariant on the node. If the cashier, modeled in Figure 3, does not take action and synchronizes on the `validate` channel, the customer can, when exactly 30 time units have passed, go and synchronize on the `pay` channel and finish their purchase - just like in the real world where if a cashier does not arrive within a given time, the customer's validation is canceled.

If the cashier gets to validate the flow moves to `Validating_groceries` where if the customer's basket reflects what has been scanned, the customer is allowed to proceed with their purchase and finish the flow. But if the basket does not reflect what has been scanned, the validation failed, moving to the `urgent` node `Validation_failed`. The node is urgent to ensure that the next edge is taken immediately when available without time passing. Here, the flow moves to the node `Fix_Basket` where the customer is tasked with fixing their basket. When said task is complete, the edge leading to and from the same node becomes unavailable and the edge leading away from the node becomes available, and the customer can now finish their purchase.

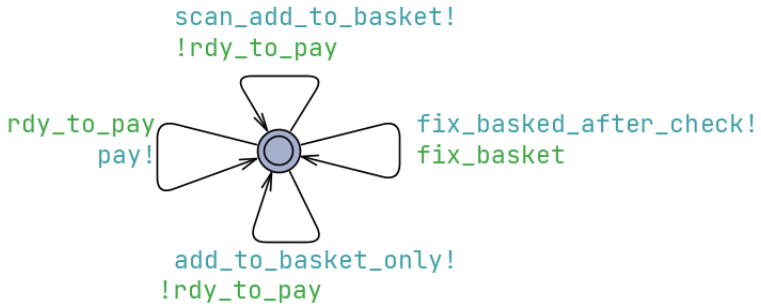

Fig. 2. The Uppaal implementation of the customer

Fig. 3. The Uppaal implementation of the cashier

It has been decided that the UPPAAL model will not reflect the flow of a customer not willing to fix their honest mistake, i.e. the model does not reflect any customers willing to steal.

3 Temporal Properties

The following describes the identified timed properties for the timed automata described and developed in Section 2. All properties are verified in Section 4.

1. No deadlock can occur except when the flow is finished. This ensures that nothing bad ever happens that cannot be recovered from.
2. A customer must be able to fix their basket if they reach said location. This ensures that something good will eventually happen.
3. Any customer cannot have the flow moved to `Having_Groceries_rdy` if the variable `rdy_to_pay` is not true.
4. The time cannot exceed 300 time units when being in location `Buying_Groceries`. This ensures that the invariant is never broken.
5. Being in location `Validation_failed` the variables `scanned_amount` and `groceries_amount` must not be equal.
6. Being in location `Validation_succeeded` the variables `scanned_amount` and `groceries_amount` must be equal.

4 Verification stage results

To verify the model's safety/liveliness properties, six validations have been constructed, as seen in figure 4. The following explains what each validation validates:

1. ensures that 'for all paths, always', - `A[]` - deadlock can only occur in the location `Groceries_Bought` for process `buying_groceries` - which is intended.
2. being in location `Fix_Basket` for process `buying_groceries` 'leads to' the variables `scanned_amount` and `groceries_amount` being equal which means that 'for all paths, always, eventually' (represented with the leads to property '–>' in UPPAAL) if the customer ends up in said location they will fix their basked.
3. ensures that 'for all paths, always', - `A[]` - being in location `Having_Groceries_rdy` implies that the variable `rdy_to_pay` is true.
4. ensures that 'for all paths, always', - `A[]` - the clock cannot exceed 300 time units when being in location `Buying_Groceries`.
5. ensures that 'for all paths, always', - `A[]` - being in location `Validation_failed` implies that the variables `scanned_amount` and `groceries_amount` are not equal.
6. ensures that 'for all paths, always', - `A[]` - being in location `Validation_succeeded` implies that the variables `scanned_amount` and `groceries_amount` are equal.

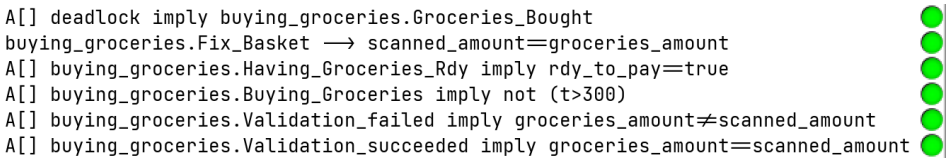

Fig. 4. The results of the Uppaal validation queries

5 Timed DCR graph

In a daily case scenario, a person may enter the supermarket to buy groceries. They use their smartphone to scan the items they want to buy and place them in their shopping basket. They have 2 hours after every scan to mark their shopping as ready for validation. After the shopping basket is marked as ready, the user waits for the response. If Netto's cashier believes that no validation is needed, the customer has 5 minutes to pay and 10 minutes to leave after success. If a validation is needed, there are two possible outcomes:

1. The validation is marked as successful by the cashier, which means that the number of items in the basket corresponds to the number of items scanned. The user proceeds to finalize their order (they have 5 minutes to pay and 10 minutes to leave after success).
2. The validation is marked as unsuccessful by the cashier, which means that the user has to re-scan and re-validate the missing items in order to fix their basket. To do so, they've got 10 minutes. After fixing it, the user waits for the cashier to re-check his basket and mark the fix as done. After that, the user proceeds to finalize their order (they have 5 minutes to pay and 10 minutes to leave after success).

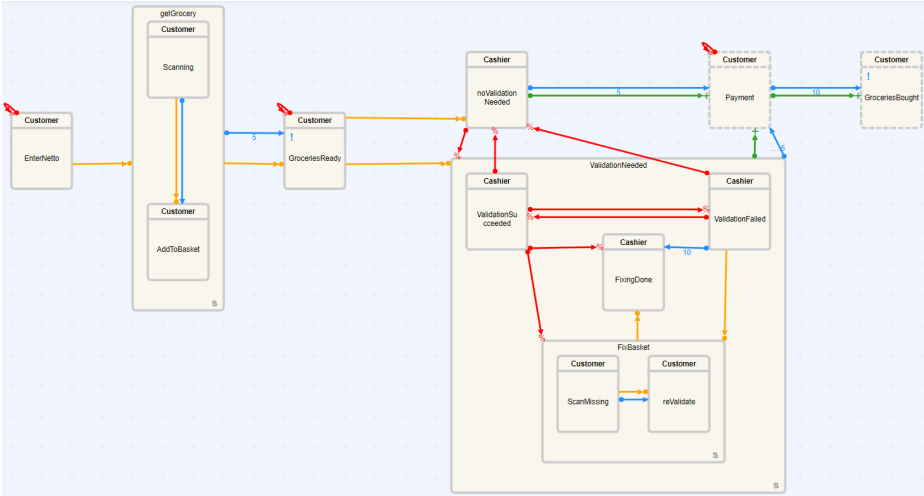

Fig. 5. Timed DCR graph of our model[1]

6 Tests

In the timed DCR graph developed in the earlier section 5, it was required to assume three traces for all types of flows. All positive, neutral and negative traces will be included as witness of validity of the model to show that the positive and neutral traces are re-playable, while the negative is violated. These three types of flows are further described as :

Positive traces

[1] the dots before deadlines' numbers are there just to move said number more to the right, in order to make it more readable

These are traces that are assumed as a sequence of activities that are complete and can satisfy a goal. In other words, this refers to a path that must be possible to complete.

1. **Test 1**
   Here, the trace was formed based on the path where the customer performs the grocery buying process without having to be validated by the cashier. The customer performs the activity `EnterNetto` to start the process, followed by `Scanning` and `AddToBasket`. After this the `GroceriesReady` activity makes it possible for the cashier to perform the `noValidationNeeded` process which enables the `Payment` and `GroceriesBought` activities to end the process.

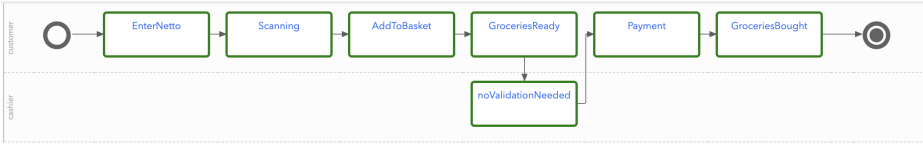

Fig. 6. Positive Trace Test-1

2. **Test 2**
   In this test, the trace was based on the path where the customer goes through the grocery buying process and being successful validated by the cashier. The customer performs the activity `EnterNetto` to start the process, followed by `Scanning` and `AddToBasket`. After this the `GroceriesReady` activity makes it possible for the cashier to perform the `ValidationSucceeded` process which being successful enables the `Payment` and `GroceriesBought` activities to end the process.

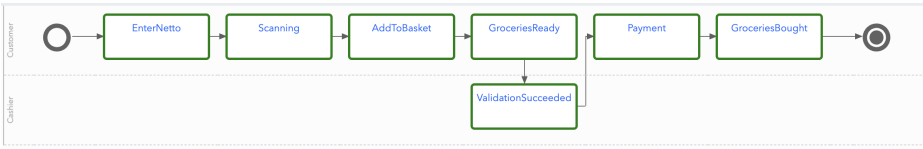

Fig. 7. Positive Trace Test-2

3. **Test 3**
   For the final positive trace test, it was based on the path where the customer goes through the grocery buying process and fails the validation process performed by the cashier. The customer performs the activity `EnterNetto` to start the process, followed by `Scanning` and

`AddToBasket`. After this the `GroceriesReady` activity makes it possible for the cashier to perform the `ValidationFailed` process which requires the customer to perform `ScanMissing`. After which the cashier performs `ReValidate` leading to the customer performing the following `FixingDone`, `Payment` and `GroceriesBought` activities to end the process.

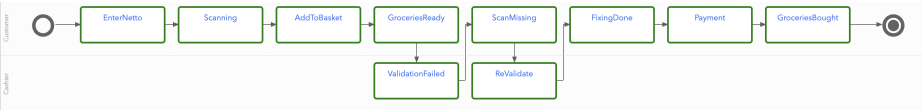

Fig. 8. Positive Trace Test-3

Neutral traces

These are traces that are not complete but can achieve a goal. It is a path that might be possible to do or a possible-yet-inefficient trace, making it an optional trace.

1. **Test 1**
   Here, the customer performs the activity `EnterNetto` to start the process, followed by `Scanning` and `AddToBasket`. After this `GroceriesReady` is reached, but it is not complete as the process has not reached its end but can achieve the goal of having groceries ready.

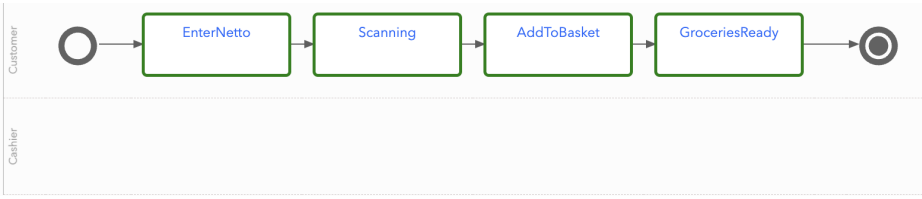

Fig. 9. Optional Path-1

2. **Test 2**
   Here, the customer performs the activity `EnterNetto` to start the process, followed by `Scanning` and `AddToBasket`. After this `GroceriesReady` is reached and the cashier performs `noValidationNeeded`, but it is not complete as the process has not reached its end but can achieve the goal of having a successful validation state.

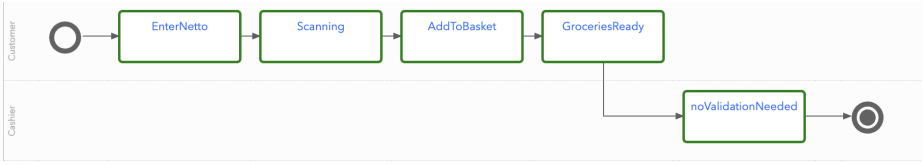

Fig. 10. Optional Path-2

3. **Test 3**
   In the last optional trace, the customer performs the activity `EnterNetto` to start the process, followed by `Scanning`, `AddToBasket`,`GroceriesReady` and the cashier performs `ValidationFailed`. This ensures the customer to perform `ScanMissing`, after which the cashier performs `ReValidate` which makes the process incomplete, as the process has not reached its end but can achieve the goal of having a successful re-validation state.

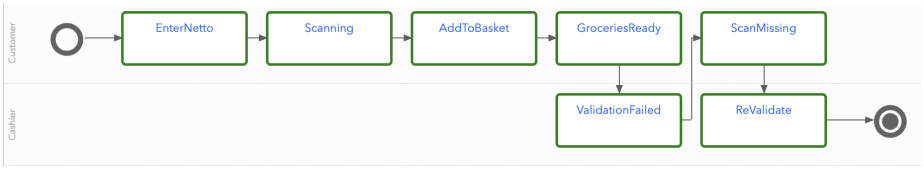

Fig. 11. Optional Path-3

Negative traces

This is also referred to as the forbidden trace which clearly violates the specifications of the timed DCR graph. It is a path that cannot be completed.

1. **Test 1**
   Here the customer performs `EnterNetto` then proceeds to `GroceriesReady` without fulfilling `Scanning` and `AddToBasket` that make the current activity available. Making the path incomplete and not executable.

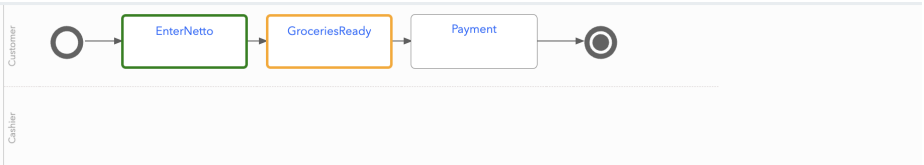

Fig. 12. Forbidden Path 1

2. **Test 2**
   In this test, `Scanning` and `AddToBasket` are performed, but there is a missing activity `AddToBasket` which does not fulfill `GroceriesReady`. This violates the process as it is not complete and not executable.

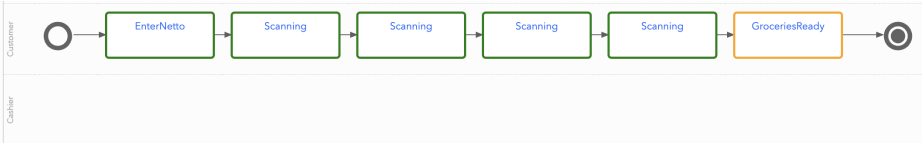

Fig. 13. Forbidden Path 2

3. **Test 3**
   In the final test, the customer has a successful grocery process but after `noValidationNeeded` activity, it is followed by the wrong activity `ScanMissing` which should be executed when `ValidationFailed` is fulfilled. From this the trace is not executable, violating the specification of the process

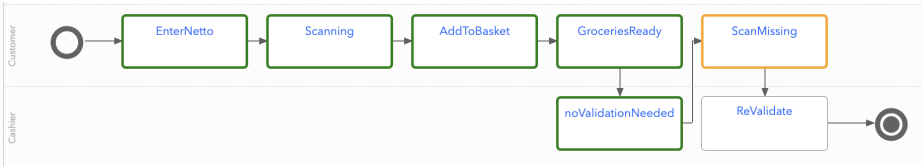

Fig. 14. Forbidden Path 3

7 Properties

This is how we modeled the properties shown in Fig.4. in our timed DCR graph:

1. To ensure that no deadlock occurs, we've made the payment option available only after the groceries have been marked as ready and have been eventually validated.
2. To ensure that the customer actually fixes their basket, we've made the activity FixingDone mandatory after a failed validation.
3. To ensure that the customer always pays we've made the GroceriesReady activity as self-exclusive, so that the user can't go back to scan more items. Also, we've made the Payment activity mandatory after whichever validation activity the user went through.

4. To ensure that the time cannot exceed 5 minutes for the whole scanning and adding to basket operation, we've put a deadline of said time to mark the shopping basket as ready.
5. To ensure that we can only be in ValidationSucceeded whenever the groceries amount and scanned amount are equal, we've put an exclusion between it and ValidationFailed, FixingBasket, and FixingDone.
6. To ensure that we can only be in ValidationFailed whenever the groceries amount and scanned amount are not equal, we've put an exclusion between it and ValidationSucceeded.

8 Compliance checker

In the following, we developed a compliance checker in pseudocode which is based upon the given open source implementation, which can be found at https://github.com/tslaats/REBS2021/tree/main. Our solution is easy to implement on top of the given open-source implementation as a simple extra function that runs in addition to the existing code. For this, the `while` loop would then have to be removed, as this loop has only been introduced to make the pseudocode more understandable. Because of lack of time, the pseudocode was not actually implemented in the given open source code, as this would require a lot of time, considering that there is no existent documentation at all for the given open source code.
Due to space requirements for the report, the pseudocode can be seen in the appendix in section D.

In the following, we describe the pseudocode briefly for a better understanding. The provided pseudocode represents a compliance checker for a timed Dynamic Condition Response (DCR) graph model. The checker begins by initializing several variables: `executed_traces` to store the traces that have been executed, `rules` to contain the rules to be checked for compliance, `params` to store the parameters of the model, and `pending_traces` to store the traces that are yet to be executed. A clock object is also initialized and set to 0.

The model then enters a loop that continues as long as the model is running. In each iteration, a trace is executed and added to `executed_traces`. If the executed trace is in `pending_traces`, it is also removed from there. The clock is then updated based on the time conditions of the executed trace, and the next trace to be executed is added to `pending_traces`. The parameters of the model are updated based on the instructions of the executed trace.

The model then checks the compliance with the rules. The rules can be of four types: `check_deadlock`, `check_execution_requirements`, `check_clock`, and `check_parameter_requirements`. Each rule type checks for a specific condition in the model as can be seen in table 1.

If any rule is violated, the model outputs a message indicating the violation is returned. If the model finishes running without violating any rules, a message indicating successful completion is returned. If the model is still running, it continues to the next iteration of the loop. This process continues until the

Check Name	Purpose
`check_deadlock`	Check if the model has run into a deadlock
`check_execution_requirements`	Check if a prior-execution requirement is satisfied
`check_clock`	Check if a timing constraint is satisfied.
`check_parameter_requirements`	Check if a parameter requirement is satisfied.

Table 1. Overview of the implemented checks and their purpose.

model has finished running, or a deadlock occurs, in which case the model also outputs an error message regarding the deadlock.
Describing each check in detail would require too much space, so we refer to the pseudocode for this, given that the checks are quite trivial, and we have added examples as comments.

9 Compliance checking results

In the following it is argued why the compliance check with the compliance checker from Section 8 completes successfully for the timed DCR graph from Section 3 and the temporal properties/rules from Section 5. First, it should be noted that checking if the timed DCR graph fulfills the respective temporal properties results in a check similar to the ones for the Uppaal model, which is described in Section 4.
As can be seen in the compliance checker section, there is support for the four necessary goal-checking categories that can also be seen in figure 4:

1. Check that no deadlock occurs.
2. Check whether the amount of scanned items equals the amount of items in the basket.
3. Check that when being in step `Having_Groceries_Rdy` implies being ready to pay.
4. Check that being in step `Buying_Groceries` implies clock ≤ 300.
5. Check that being in step `Validation_failed` implies that the amount of scanned groceries does not match the amount of items in the basket and vice versa for `Validation_succeeded`.

If one now applies the Timed DCR graph to the Compliance Checker (which as mentioned builds upon the open-source implementation), then the outcomes in Table 2 can be trivially observed.

It would take too much space to go through each of the checks manually, but it can be seen easily from following the timed DCR Graph with the Compliance Checker pseudocode that each safety/liveness rule is satisfied. To give an example of how this would be done, a compliance check for rule $5a$ is briefly described in the appendix in section E.

The other checks can be validated accordingly by applying the respective logic in the same way.

With this we have shown that all liveness/safety properties get validated successfully for the timed DCR graph, by using the created Compliance Checker.

Validation No	Compliance Check Outcome
1	`"No deadlocks!"`
2	`"No violation for rule scan_basket_comparison"`
3	`"Rule Check_groceries_rdy is okay."`
4	`"No time violation for rule buying_clock_check"`
5a	`"Rule check_validation_fail is okay."`
5b	`"Rule check_validation_success is okay."`

Table 2. Overview of the compliance check outcomes. Note that the respective rule names have been chosen arbitrarily and are just exemplary.

10 Consent

The following table, Table 3, gives an overview of the consent given to the various parts of the code, developed models, and process descriptions.

Artifact	We provide consent	We do not provide consent
Process description	x	
Model	x	
Code	x	

Table 3. Overview of consent given.

Appendix

A Uppaal Global Declarations

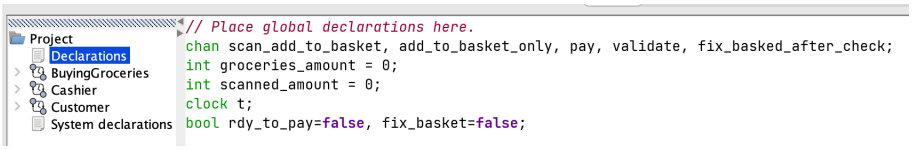

Fig. 15. Figure illustrating the global declarations used for the Uppaal timed automata.

B Uppaal Buying Groceries Model With Parameter

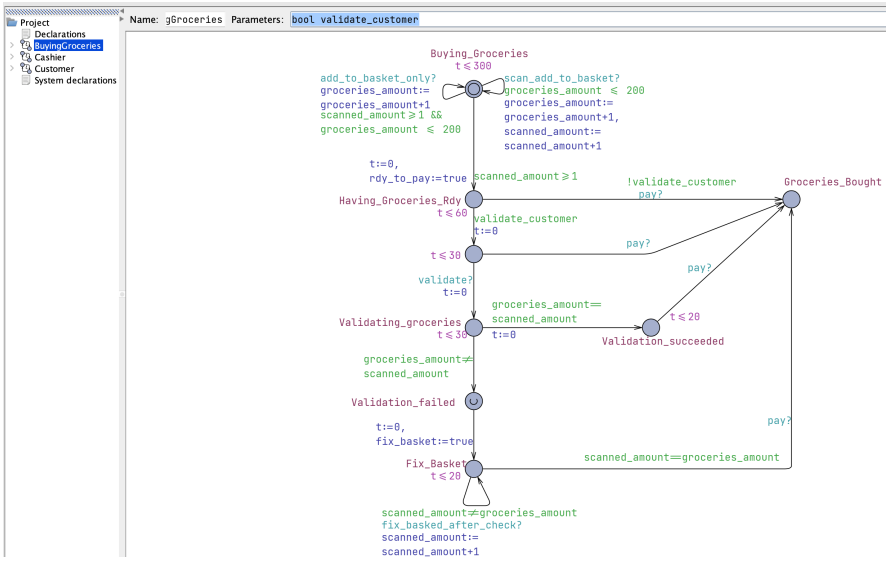

Fig. 16. Figure illustrating the `BuyingGroceries` template with parameters visible.

C Uppaal System Declarations

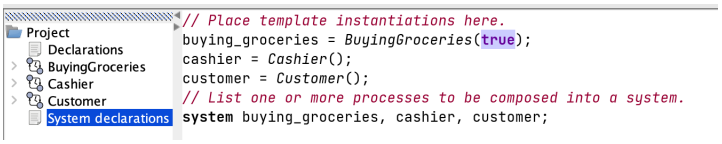

Fig. 17. Figure illustrating the System Declarations and how the parameter from `BuyingGroceries` is initiated.

D Compliance Checker Pseudocode

```
array executed_traces = []
array rules = []
dictionary params = {}

array pending_traces = []

clock.set_time(0)

model_running = True
while model_running:
    executed_tr = execute_trace()
    executed_traces.append(executed_tr)
    if executed_tr in pending_traces:
        pending_traces.pop(executed_tr)
    clock.update(executed_tr.get_time_conditions())

    pending_traces.append(execute_traces.get_following_trace
        ())

    params.update(executed_tr.get_param_instructions())

    # check rules
    for i in rules:
        parsed_rule = i.parse_rule()

        if parsed_rule.get_category() == "check_deadlock":
            # check for deadlock - only makes sense if this
                is the last step
            if (len(pending_traces) > 0
                && executed_tr.get_next().is_available() ==
                    False):
                model_running = False
                return "Deadlock is reached!"
            elif (len(pending_traces) == 0
                && executed_tr.get_next().is_available() ==
                    False):
                # reached the end!
                model_running = False
                return "No deadlocks!"
            else:
                model_running = True
                return "Model not finished running yet, so no
                     assessment possible"

        elif parsed_rule.get_category() == "
            check_execution_requirements":
            # example for such a rule:
```

```
42              # "B can only be executed if at any state A has
                    been executed"
43              array needed_executed_traces = parsed_rule.
                    get_req()
44              for i in execute_trace:
45                  if i in needed_executed_traces:
46                      needed_executed_traces.pop(i)
47              if (len(needed_executed_traces) > 0):
48                  return "Rule " + parsed_rule.get_name() + "
                        is violated!"
49              else:
50                  return "Rule " + parsed_rule.get_name() + "
                        is okay."
51
52          elif parsed_rule.get_category() == "check_clock":
53              # example for such a rule:
54              # "Execution of C implies clock < 200"
55              if (parsed_rule.get_time_constraint() >=  clock.
                    get_time()):
56                  return "No time violation for rule " +
                        parsed_rule.get_name()
57              else:
58                  return "Time violation for rule " +
                        parsed_rule.get_name()
59
60          elif parsed_rule.get_category() == "
                check_parameter_requirements":
61              # example for such a rule:
62              # "Execution of D implies param1 == param2"
63              param_to_check = parsed_rule.get_req_param()
64              if (parsed_rule.get_target_value()
65                  == params.get(param_to_check).get_value()):
66                  return "No violation for rule " + parsed_rule
                        .get_name()
67              else:
68                  return "Violation for " + parsed_rule.
                        get_name()
```

Listing 1.1. Pseudocode for model validation

E Compliance checking results - Algorithm flow

Note that reoccurring steps like adding further traces to the array and parsing them etc. are skipped here with "..." to save some space.

```
1     Initialize all arrays and variables  & set clock to 0
2     Add EnterNetto to pending_traces
3     Enter while loop
4     Execute EnterNetto
5     Add EnterNetto to executed_traces
6     Remove EnterNetto from pending_traces
7     Update clock with one tick
8     Add Scanning to pending_traces
9     (No parameter changes, so no parameter update)
10    ...
11    Execute Scanning
12    ...
13    Update parameter to scanned_amount+=1
14    ...
15    Execute AddToBasket
16    ...
17    Update parameter to groceries_amount+=1
18    ...
19    Execute ValidationSucceeded
20    ...
21    Enter elif for "check_parameter_requirements"
22    Set param_to_check = scanned_amount
23    Check if scanned_amount.target_value == groceries_amount.
          target_value
24    Comparison evaluates to True
25    // Note: This can not be explicitly verified, since this
          depends
26    // on the actual real-life behavior. It can however be
27    // implicitly verified, by the assumption, that the
28    // cashier is able to count correctly (simple summation)
          and
29    // thus the model only ends up on ValidationSucceeded if
30    // and only if the two parameters match.
31    // Since the model also allows for a Fail scenario,
32    // i.e. the parameters do not match, we can verify
          implicitly
33    // (under the constraint that the cashier can do basic
          math)
34    // that we only end up in ValidationSucceeded if the
35    // constraint is correct, because else the model allows
          for
36    // another scenario to be executed.
37    // Thus we have verified that the check completes
          successfully.
```

Listing 1.2. Compliance check for rule 5a