Bachelor Thesis Project
Proof of concept:
Test Driven Modelling with open Test - supervised by Tijs Slaats

Leon Krüger - rtj284

June 11, 2023

Abstract

This work is the first step of a proof of concept - of theory provided by my supervisor. The thesis captures the challenge and process of going from theory to implementation. This is a theoretical and practical project, as there were a lot high level formal concepts to understand, and implementation work to do. Both the domain of business process modeling and all the theory coupled to it, as well as front end development in react were completely new to me in the beginning of the project. There will be shown how the focus changed throughout the course of the project, as new connections opened up, and how this project was a starting point for some original distribution to this topic.

Note that all the concepts which refereed to by citation will also be described in this thesis to the extend it is relevant for the project.

The whole project code is found under
`https://github.com/Vortex0506/BA-Thesis-TDM.git`

Contents

1 Introduction

1.1 Test Driven Modelling

Test Driven Modelling with open tests, such as it is defined in [1] is the main concept of this thesis. This approach describes how the correctness of a modelled workflow or process can be ensured, when using a thread based process notation.

A thread in this context is a sequence of activities in this workflow/process.

The idea is, that before beginning to model a extensive and complicated workflow the user establishes some constraints, here defined as tests or more specific open tests (which will be described in section 3). By that the user can observe whether the model lives up to the defined constraints.

However a model might change to a degree where some tests are heavily affected by the changes, such that new activities interfere with the defined constraints. In order preserve the relevance of tests without having to modify them each time a model is updated, open tests introduce the concept of a context. Each test will have a context, which specifies the relevant activities for that specific test. This ensures that what ever happens on a workflow/process between the relevant activities does not matter, as long as this test-trace projected onto the relevant activities satisfies the wished functionality. That means when evaluating the test we can omit to look at non-relevant activities, only focusing on whether all important activities are present in the right order.

This approach makes it possible for the user to define some tests and by incrementally updating the model, see if those constraints still are satisfied. By that one can observe if extensions or changes in a workflow will have influence on standards or rules which have been set before.

1.2 The course of the project

This work should be seen as a first step in the process of proving the concept of Test-Driven-Modelling (TDM) with Open Tests which was introduced by Tijs Slaats et al. in [1]. The goal in the beginning of this project was to implement the whole formal defined test system from the paper into a basic modeling application. This thesis will focus on the process on going from

theory to implementation. In order to define a red thread for the report, the following will describe the route the project took.

In the course of this bachelor thesis project I had to gain the theoretical knowledge which builds the foundation for the concept of TDM. This included understanding of the Dynamic Condition Response (DCR) Graph process notation, which is a declarative process model approach.
Note that when referring to core concepts such as DCR-graphs or TDM I do that to the extend they are used and defined in [1].

After having acquired a basic understanding of the high level formal concepts introduced and used in the TDM approach [1], I had to transfer this understanding into a scientific prototype. I was supplied with a basic DCR-Graph modelling application. I developed a understanding of the library it was build upon (React), which was new for me at that time, and developed understanding of the structure of the application itself.

The paper of Slaats et al. [1] focuses a lot on how to define syntactical checks, such that tests for a model can be preserved throughout changes of the model, and it only scratches on how to evaluate the tests if these syntactical checks do not apply. This process of evaluating the tests on a model is called Model Checking. The starting idea was to do the Model Checking and follow the defined road-map for the syntactical checks after. However we realized that the problem of Model Checking the graph was a more complicated task then expected. This led to a change of focus, the task of Model checking became much more prevalent.
After consulting Axel Kjeld Fjelrad Christfort (Ph.D. student of Tijs) we discovered that Axel's latest work [2] could be used for solving the challenge of model checking a DCR-graph in a efficient manner. This mapping from the Alignment algorithm [2] to the test driven modeling is original work which had its provenance in the course of this project. It is described precisely and formal in [2].

It is also to be mentioned that Axel was part of developing the application I was supplied with. He was functioning as an additional informative consultant throughout the project. After understanding the new high level theory and implementation Axel's work introduced, I took my first steps in front end development in order to map the given application to the supplied Alignment algorithm [2]. This led to the scientific prototype being extended with new functionality and new features to the user-interface, such that is

it possible to define open tests and checking whether a model passes those tests, hence following the TDM approach for DCR-Graphs [1].

2 DCR-Graph

The Dynamic Condition Response Graph is the model concept used throughout this project. It is to note that in general business process modeling the term *event* and *activity* have distinct meaning. For DCR-Graphs is that not the case. When referring to *events* or *activities* in this thesis, it is the same thing, namely a node in the graph which represents something that can be executed in order to move along a process.

Definition 1 (DCR Graph [3]). A DCR graph is a tuple (**E,R,M**) where

- **E** is a finite set of activities, the nodes of the graph.
- **R** is the edges of the graph. Edges are partitioned into five kinds, named and drawn as follows: The conditions ($\rightarrow\bullet$), responses ($\bullet\rightarrow$), inclusions ($\rightarrow+$), exclusions ($\rightarrow\%$) and milestones $\rightarrow\Diamond$.
- **M** is the marking of the graph. This is a triple (Ex, Re, In) of sets of activities, respectively the previously executed (Ex), the currently pending (Re), and the currently included (In) activities.

The above definition gives and overview on how DCR-graphs work, the edges denoted by **R** are also refereed to as relations between activities.

However there are some things to note. In order to explain those details which have been important throughout this project consider the example below. Also note that my understanding of DCR-graphs is derived via the information from [1].

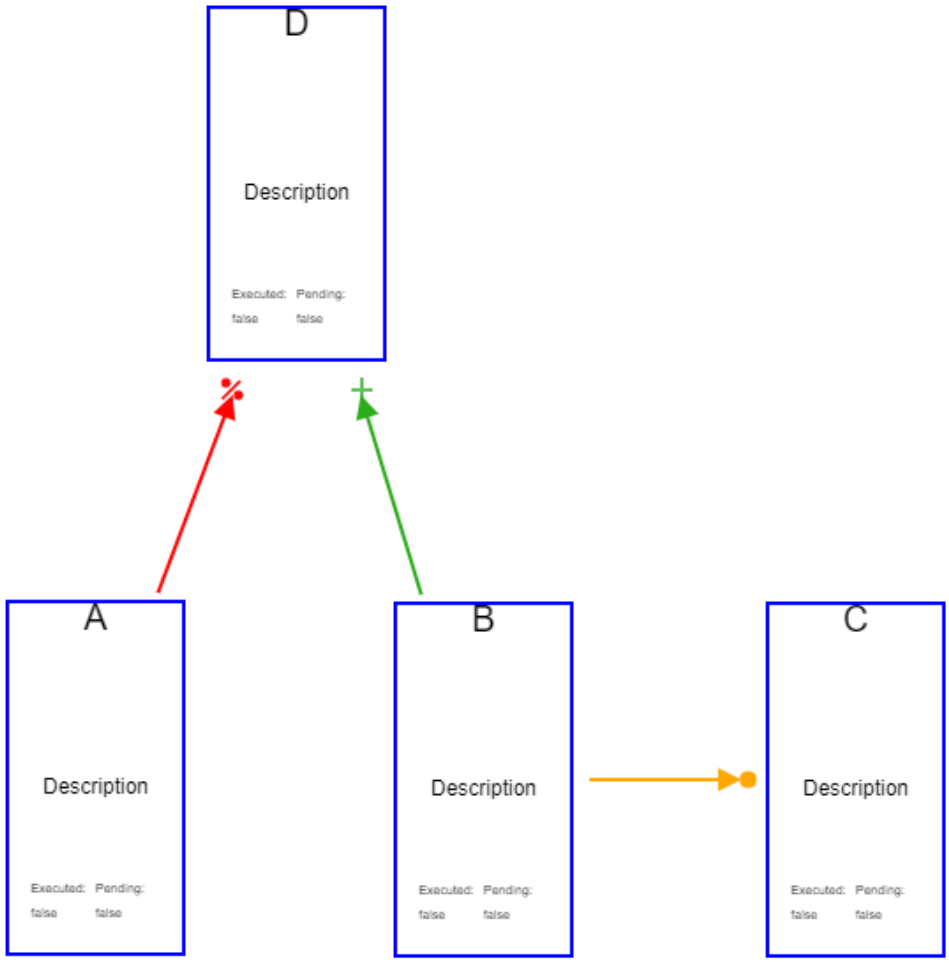

Figure 1: Example 1 DCR-graph

Recall from Definition 1 that there are three sorts of markings: Executed, Pending and Included. It is to be outlined that all activities are included by default if they are not excluded by a relation.

If A is executed in the above example the marking will change as:

$$< \emptyset, \emptyset, \{A, B, C, D\} > \quad \rightarrow \quad < \{A\}, \emptyset, \{A, B, C\} >$$

An excluded activity can only be included when a event is executed which has a include relation to the given event. In the above example D is included

again as soon as B is executed. The marking changes as:

$$< \{A\}, \emptyset, \{A, B, C\} > \quad \rightarrow \quad < \{A, B\}, \emptyset, \{A, B, C, D\} >$$

Also note, that the executed activities stay in the included set of the marking. Activities can be executed multiple times at any point in the model, as long as these are *enabled*. This property will gain in importance, as the marking can mutate often, which increases difficulty in defining all possible ways a model can be executed.

In order to explain the term *enabled* we have to talk about the relations: conditions, responses and milestones, presented in definition 1.
An activity is enabled if:

- The activity is included.
- All its conditions are executed.
- All its milestones are not *pending*, when included.

In order to understand what the term *pending* means we have to outline how the response and milestone relations behave. Lets refer to the example graph below.

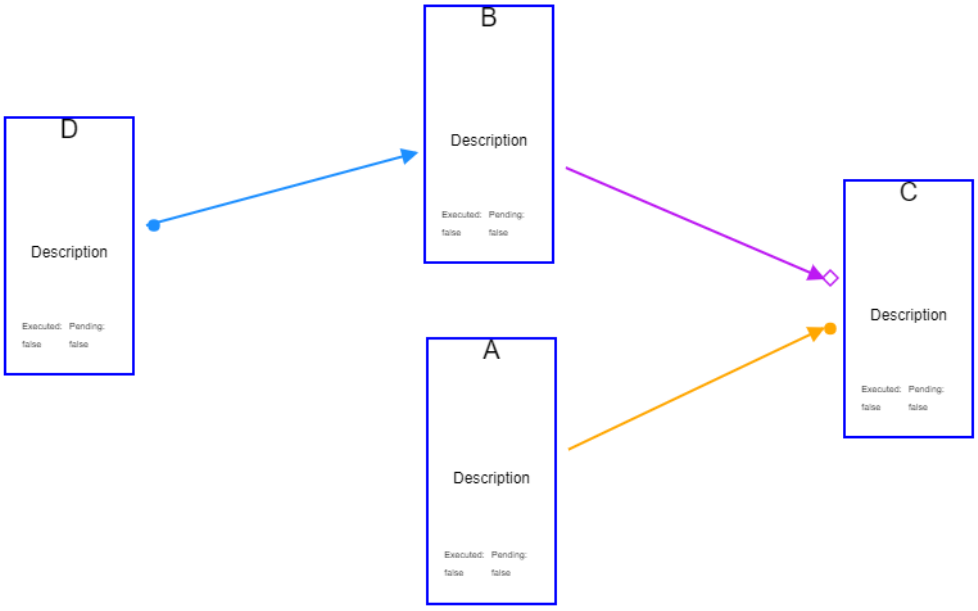

Figure 2: Example 2 DCR-graph

First lets define the marking: $< \emptyset, \emptyset, \{A, B, C, D\} >$
This are all enabled events, when no event is executed: Enabled: $\{A, B, D\}$

These activities are all enabled, as they all are included and they do not have any incoming condition or milestone relations.

We can observe, that when A is executed, which has a outgoing condition, C will be enabled. That is because the milestone relation from B to C is not specifying any prerequisite for activity C, as long as B is not pending. When A is executed the marking and enabled activities are changing as:

$$\begin{aligned} Marking: &< \emptyset, \emptyset, \{A, B, C, D\} > &\rightarrow &< \{A\}, \emptyset, \{A, B, C, D\} > \\ Enabled: &\{A, B, D\} &\rightarrow &\{A, B, D, C\} \end{aligned}$$

In order to demonstrate the response and milestone relation and there impact on the marking and enabled activities we assume event D is executed as well. By that B is set to be pending, as the event had a in-going response relationship from D. The next mechanism gets triggered and C will no longer be enabled as its milestone B is now pending. We can observe the following changes:

$$\begin{aligned} Marking: &< \{A\}, \emptyset, \{A, B, C, D\} > &\rightarrow &< \{A, D\}, \{B\}, \{A, B, C, D\} > \\ Enabled: &\{A, B, D, C\} &\rightarrow &\{A, B, D\} \end{aligned}$$

Now when B is executed there is no pending activity and all events are enabled again. When B is executed:

$$\begin{aligned} Marking: &< \{A, D\}, \{B\}, \{A, B, C, D\} > &\rightarrow &< \{A, D, B\}, \emptyset, \{A, B, C, D\} > \\ Enabled: &\{A, B, C\} &\rightarrow &\{A, B, D, C\} \end{aligned}$$

The important take away for the rest of the thesis is the observation that markings can change often, due to re-executed events. In the above example we could re-execute D and by that B is set pending again and the above example starts over again.

3 Open Tests - Theory of Paper

The concept of open Tests was introduced by Tijs Slaats et al. [1]. This concept can be generalized to a arbitrary process notation with trace semantics.

However will the following focus on the DCR-graph notation introduced in the section above.

Open tests contain three elements: a test-trace, a context and its polarity. The test-trace is the sequence of activities in the model we want to check. This trace can have duplicates, the context can not. It makes sense to think of those concepts as an array and a set.

The context defines which activities are of importance for the corresponding test. As introduced earlier, not all events in a model are relevant for a test, if the context does not explicit define so.

In order to understand how the context influences the outcome of a test consider the example below. Note we omit the concept of polarity for now.

We imagine a graph with activities $\{A, B, C, D\}$.
The test is defined as: $\big([B, D]\{B, D\}\big)$

Now assume that the model only can be executed in a linear fashion, hence the only trace we can find which contains B and D is $[A, B, C, D]$. However the context specifies what the relevant activities for the test are. Hence we do now consider the found trace in relation to the context. This is formally defined as *projection* in [1]. When looking at the projection from $[A, B, C, D]$ on $\{B, D\}$, we only consider the activities we can find in the context. Hence we are to omit all other activities. This leads us to the trace $[B, D]$ as we do not keep A and C in this projection.

Note that when cutting some activities, the order of the relevant activities stays the same.

In the example above we can observe that the projection onto the context has given us the trace we where looking for: $[B, D]$.

In order to bring polarity into play, consider the following. If the test above would have been a positive test, it would have passed, as we could find the specified trace. This is the intuitive behaviour of a test, if we achieve the outcome we were looking for the test passes.

However if the same test would be defined to have negative polarity, the outcome discussed above would lead to the test failing. That is because, whether a positive test wants to find a trace which satisfies the test, a negative test wants to ensure that there does not exist any trace which satisfies the test. Meaning if we in the above test would not be able to find a trace which projected to the context is $[B, D]$, then the negative test would pass.

In order to show a example we consider the same graph with activities $[A, B, C, D]$ and recall, that those events only can be executed in a linear fashion, due to linear dependency from A to D such as conditions.

The negative test is defined as the following: $\big([B, D]\{B, C, D\}\big)$

Now the trace $[A, B, C, D]$ projected onto the context $\{B, C, D\}$ gives us the trace $[B, C, D]$. As we were looking for the trace $[B, D]$ and there is no way of achieving this trace without executing C in between (due to linear dependency from A to D) the negative test will pass.

3.1 Syntactical Checks

So the main idea behind syntactical checks is, as Slaats et al. [1] specified formally, that if changes to a model do not change or introduce new activities which are in the context of a test, the test will be preserved throughout a evolving model.

It is to note that I will not dive into the details of these syntactical definitions, as the focus of my thesis shifted heavily and the syntactical checks became less relevant. This will be elaborated further in section 4.2.2.

However I will still introduce the main concept and underlying idea, as this is and was of relevance for the extension of the supplied application and also for potential users. Furthermore it is to note that the process of understanding the detailed formal concepts did cost a lot of resources, even if they became less prevalent after.

As introduced above the core idea of the syntactical checks are to observe whether a model has changed to a degree which has influence on the context. In order to check these changes the tests should be able to refer to the *old* state of the graph/model and the *new* state of the graph.

This would work as the following: The user specifies the tests, creates his model and executes the test afterwards. This would mark one iteration of the test driven modeling. After executing the tests the state of model is saved. Then when the model will be changed and the tests are to be executed again. Yhe idea is to compare the current state of the graph with the one saved before. By the static properties and definitions defined by Slaats et al. [1], we could now identify which tests are preserved.

However there is another thing to note. In order to be able to prove the mentioned properties there have been established a syntactical condition for modification of a DCR-graph. This syntactic condition ensures that positive tests can be preserved. I am noting that here, as this has influence on how the user should behave when modeling.

Intuitively speaking this condition defines, that when extending the DCR-graph: *new* relations are only allowed to be between *new* activities or they may be conditions or milestones from *new* to *old* activities [1]. *New* in this context means everything which came after the prior iteration of modeling and executing the test. In that sense *old* activities are the activities which have been in the version of the graph which already has been saved.

This property is ensuring that the *old* part of the graph is still working, even when the model evolved. As the *old* graph only can be connected by conditions and milestones, it is certain that the functionality of the earlier graph is preserved. The extension can only be perquisites to this functionality.

It is to be noted that this condition may limit the amount of freedom the user has when modeling a workflow/business process. However it should be considered that this is only a starting point for modeling. With focus on preserving the positive tests, hence the user can still change the *old* graph, while being conscious in the fact that the formal system can not ensure the correctness of old tests anymore. By that the user will have to know that it might be necessary to re-think or change some of the established tests.
As tests only specify a certain behaviour and conditions in a workflow, it is conceivable that the user at some point in time has to fundamentally change how the workflow/process can be executed and by that the constraints regarding this process will and should probably change as well.

3.2 Model Checking

Model checking is the task of running through the model/graph in order to find a specific trace.

If the syntactical checks for a test do not hold we have to execute the test manually. This is done by searching through the model in order to determine whether the given trace exists. As already hinted at, this task may be highly exponential, as the marking of a graph can change often due to re-executable events at all times.

Recall the marking changing example from Figure-4. We could observe that the set of enabled activities is changing when certain activities are re-executed. It seems intuitively obvious that this behaviour may open new sequences/traces of executing a certain workflow, assuming we are working with a highly complicated and big enough graph.

In the beginning of the project we were not aware of the dimension the problem of model checking would take. The first naive approach was to use a simple breadth-first search. However, that is not applicable.

The main reason for this is the high complexity of the *state-space* due to changing marking. The *state-space* in this sense is representing all the executions a model can take. When thinking of a breadth-first search in this context, we have to imagine its nature in connection to the open tests. Especially the negative tests make this approach not applicable, as a these tests must ensure that a specific trace does not exist in the model.

When working with a breadth-first search and picturing a tree like construct, we recall that for every possibility on a level we will exhaust all possibilities on the next level. As we do that for every single branch, in this case executed activities, we have to keep those traces in memory in order to check them all. For a big and complicated enough model the memory will run out of bound. The exponential space usage is to high.

After realizing that this task was much more complicated than expected, we pinned our focus on solving the model-checking task. Another reason for this decision is the essential importance of model checking for the TDM approach. We must be able to check any test. Whereas syntactical checks can ease the work is certain cases, the model checking task is fundamental in order to make this approach work. In order words, if the model checking task does not work test-driven modeling will not be practical applicable, as there are high chances that some of the tests do not aligning with the syntactical constraints.

When focusing on the model checking we realized that we could use the state of the art work from Axel Kjeld Fjelrad Christfort (Ph.D. student of Tijs). He was working on a method which exhaust the state-space of DCR-graphs, meaning defining a effective way of checking *all* possibilities of executing a trace in a given model. This concept is called *alignment* and is further elaborated in section 4.1.2. Note that *all* in this context does contain some limitations which will be discussed later.

It is to mention that this approach uses a bounded depth first search. It is exploiting a good trade-off between bounding the trace length, caching visited state-spaces and by that reducing the overall space it has to cover.

4 Design

4.1 Modules and components

4.1.1 Application

I was supplied with an DCR-graph model application. This application has been developed during a Computer-Science master course at KU and it contained under the MIT License.

This application is playing a central role in the practical part of the thesis. I had to extend the application by the formal test system defined in [1]. We had to cut down our expectations and focus more thoroughly on the model-checking task, resulting in the application being extended with the model-checking functionality and a user-interface for creating and using tests, such as it would make sense in a test-driven modeling approach. Further information is provided in section 5.

However in order to be able to do that I had to gain a understanding of this application. This was a substantial part of the project, as I came from a point of no experience in the programming language used and more importantly the library it was build in, namely React. Furthermore did my education to that point in time not supply me with any learning in hands on front-end development, so I had to invest in learning those aspects.

In order to give a brief overview of the application: it is build in React and uses electron to communicate with the operating system. When supplied, there was implemented the following functionality:

- Modeling a DCR-graph with events and relations.
- Saving and loading a file containing a DCR-graph.
- The swarm-context is enabling joint modeling sessions, where users can share and join a project.

In the process of grasping the applications structure and figuring out how the test system would fit in, the following non-formal graph was derived.

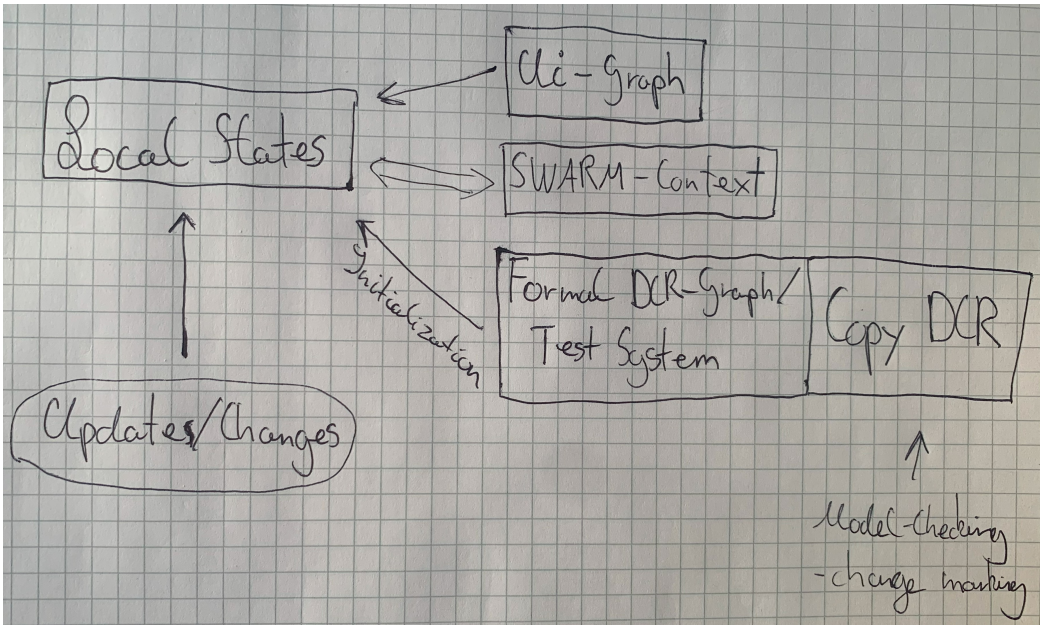

Figure 3: Structure of application

The Local States in the image above are the internal states from Reacts useState hook. This hook allows a component to have its own local states. In the application these represents all the data which defines the graph. Whenever the user changes the model these local states change accordingly.

The Ui representation of the application is directly connected to these states and does use the information to create the view on the screen. The swarm-context is referring to the local states and when a user in the modeling session sends a change proposal which is accepted, the local states of the accepting user are changed.

The question now was how the test system should be integrated. The structure of the application seemed to be a layup to introduce a whole new layer of functionality, which refers to the local states. Similar as the Ui-graph and the swarm-context.

The next question was how the formal DCR-graph, namely a graph which follows the formal definition (2) is to be derived. When supplied the application did not have a data-type which would allow one to work with the

model in a formal way. This data-type was to be instantiated from the local states. As this formal graph can be created in linear time it was a conscious decision to avoid connecting the graph directly to the changes of the user. Assuming that this would lead to a bigger overhead then having a explicit way of creating this formal DCR-graph in linear time when needed.

The fact, that in the beginning of the project there was no knowledge about that the alignment algorithm will be used to handle the model checking. And that his algorithm does handle the mutation of the marking internally. Can be derived from the figure above, as there have been some thoughts about how to copy the DCR-graph so there is a instance where the marking can be manipulated.

Another design detail to point out is that even if the formal DCR-graph is derived from the local states, it is not referring to the specific objects which are saved in these states. This is important as in React, if a object changes and this object is referred to at another place in the system, this part will also update/refresh. It is reacting on the changes. This behaviour is not desired in this context, as the change of a events name or its position on the canvas should not trigger a computational effort in the formal DCR-graph.

The formal DCR is build/instantiated from the information in the local states and parses this information (as strings) into the formal data-structure. By that there is no internal under-the-hood connection to the local states.

4.1.2 Alignment

It is to be noted that *Alignment* is a whole topic for itself. The understanding represented here is based upon Axel Kjeld Fjelrad Christforts paper on this topic [2], earlier draft versions of this paper and the personal discussion with Axel. In this section there will only be elaborated on the concept as detailed as needed in order to discuss the mapping from alignment to the test driven modeling approach.

A last note regarding Axel's paper ([2]) is the fact, that the paper is based upon a purely theoretical perspective of the DCR-graphs. Thereby the most fundamental version of the notation is considered in the provided engine. This engine does not cover the milestone relation, as the milestone is extension which is build upon of the pending system and the conditions. In that sense the milestone relation does not add anything new to the formal definitions. In the implementation section (5.3) there will be elaborated on

how this fact was taking into account. In the subsequent part of the section it is implied that the alignment algorithm also handles the milestone relation correctly.

Alignment in itself is a concept which captures how well a given trace can be aligned to a model. This means, the goal is to find a trace (a way of executing the model) which is as similar as possible as the given trace. It is conceivable that there might be situations where such a "similar" alignment is not found. In order to handle those situations and in general define what it means to find a valid alignment and setting up the boundaries of those, the definition of a *accepting* graph and trace comes into the picture.

A graph is said to be accepting, when there are no pending activities, which are included, left. A trace is accepting if the sequence of executed activities does leave the graph in a accepting state. A complete alignment is from this defined by having a accepting trace in the model. In order to make this definition more clear we now look at a example of possible alignments.

In order to do so we refer to the graph below.

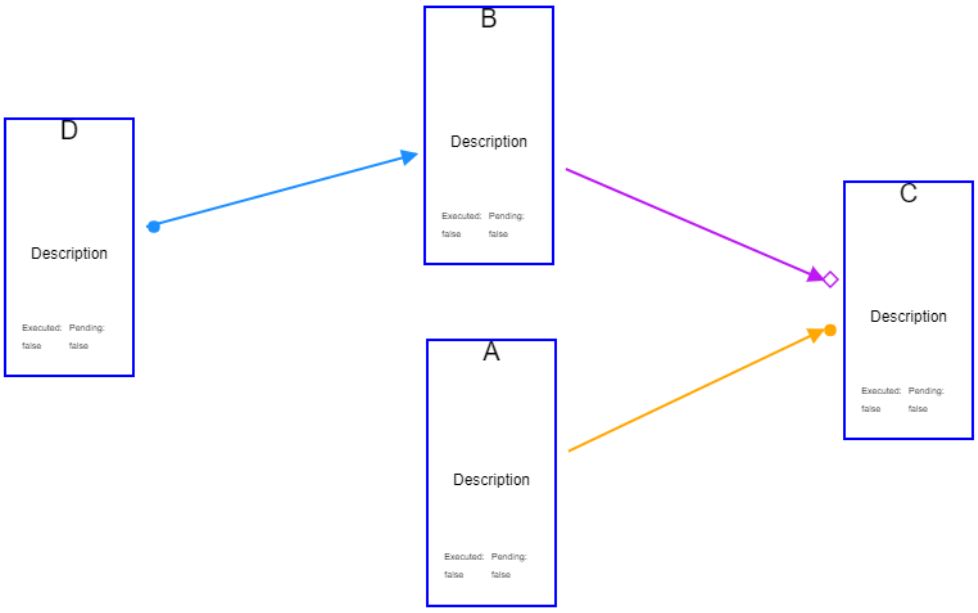

Figure 4: Example 2 DCR-graph

In order to grasp how to think about a alignment in the context of this thesis we take a look at the following example, where we want to align the trace $[A, D, C]$ with the model:

T	A	D	C
M	A	>>	C

Note that the M denotes the model and which activities has been executed there. Also the sequence denoted by M is the alignment-trace which have been found. That is the trace which actually is executed in the model. T denotes the trace which we are trying to align with model.

The symbol >> denotes a non-move, hence a misalignment between the trace and the model. Sometimes in order to find a alignment either the model or the trace has to leave a event out. This is described as skips or moves. In the above example the model does leave the execution of event D out this is called a *trace-skip* or more intuitively a *trace/log-move*, as we are moving in the trace without executing a event in the model. Hence note: >> symbol at the bottom of the table equals *trace/log-move*.

The above alignment is called a complete alignment, as the there are no pending activities and the model trace M is accepting.

Now consider the following alignment:

T	A	D	>>	C
M	A	D	B	C

This alignment contains a *model-move/skip* as there is executed a activity in the model while there is a no-move in the trace. Hence, note that a >> symbol at the top of the table denotes a *model-move*. This is also a complete alignment as the model trace executes B after it has become a pending activity (after execution of D), and thereby leaves the model in a accepting state.

The last example will show a non-complete alignment, as the graph is left non-accepting, hence there is a pending activity left.

T	A	D	C
M	A	D	>>

As we could observe are there multiple possible complete alignments for a single trace. In order to find the best alignment, hence the alignment with fewest mismatches the alignment algorithm works with a cost function. This cost-function does denote how much cost there is assigned to a *trace* or *model* move. With the accumulated cost of a found trace the alignment algorithm determines the optimal alignment.

Now I want to refer directly to the pseudo code of the alignment algorithm from [2]. It is to be noted that the focus will be on discussing the overall structure of the algorithm and only the details which are relevant to understand the mapping from alignment to the test driven modeling.

Algorithm 1 `Align(`G`, `σ`, cost default 0)`

global `min` $= \infty$.

```
 1: if cost ≥ min then
 2:    return ∞
 3: end if
 4: if accepting(G) ∧ |σ| = 0 then
 5:    return cost
 6: end if
 7: l:σ' = σ
    // Synchronous move
 8: for e ∈ ℓ^{-1}(l) do
 9:    if |σ| > 0 ∧ e ∈ enabled(G) then
10:       G' = execute(e, G)
11:       cost' = Align(G', σ', cost + K(l,e))
12:       if cost' < min then
13:          min = cost'
14:       end if
15:    end if
16: end for
    // Log move
17: if |σ| > 0 then
18:    cost' = Align(G, σ', cost + K(l,≫))
19:    if cost' < min then
20:       min = cost'
21:    end if
22: end if
    // Model move
23: for e ∈ enabled(G) do
24:    G' = execute(e, G)
25:    cost' = Align(G', σ', cost + K(≫,e)))
26:    if cost' < min then
27:       min = cost'
28:    end if
29: end for
30: return min
```

Figure 5: Alignment algorithm from [2]

The first thing to observe is that this is a depth first algorithm. As we can see on line 11, 18 and 25 the recursive call is executed to completion before continuing working on the same level.

What the algorithm does, is in essence performing synchronous, log/trace

and model moves in order to find a alignment.

This algorithm is working depth first, hence the order of the moves does matter, as the search space is iteratively bound by what it has found before. The algorithm does keep the minimum costs for a alignment and by that later recursive calls can be terminated if there cost is not less than the minimum found cost.

It makes intuitively sense that the synchronous moves are to be exhausted first, as we are searching for a optimal alignment with fewest mismatches. That means we want to execute the same activity in the model as in the given trace. Log/trace moves are second, as model moves are kept last as they may be highly exponential. (As we observed can activities in a graph be executed in many different ways)

An important detail to note is, that the cost and by that what is defined as a "optimal" alignment is only dependent on the cost function. The cost function takes the type of move as input. That means even if model-moves can be highly exponential, they are not in itself "bad" if the cost function defines otherwise.

The last thing to elaborate is a optimization, which helps with the task of reducing the search space. Recalling from section 2, we observed that the marking and enabled events can mutate. However this mutation can also course a overlap between different sub-problems, as we can observe recurring markings by executing certain events.

In the context of this algorithm two identical traces with equivalent marking will lead to the same result, as we are working with a deterministic depth-first algorithm. By that we can omit to re-compute all possible traces, as we can determine that they lead to the same result. This procedure is formally defined in [2] and the central definition is called *Execution equivalent markings*.

This optimization is found in the implementation of the algorithm as a cache-like construct.

4.2 Merging components

4.2.1 Mapping from Alignment to TDM

Now after having introduced the concept of alignment and the algorithm which finds a "optimal" alignment, it is to be discussed how we can use these components for the test driven modeling approach.

First lets recall that the cost function defines what the alignment-algorithm characterizes as the optimal alignment, and by that also what trace is returned. Note that the in pseudo-code above the found optimal trace is not returned. However, similar to the optimization of the state space reduction, is this included in the implementation. Hence the implemented algorithm returns the minimal cost and the corresponding trace it has determined.

In order to make the component of alignment useful we have to state the requirements for its usage: The alignment-algorithm will be called in connection to a open test which contains a trace (which we wish to align) and a context, which defines the relevant activities for that test-trace.

Then by defining the cost function in a certain way, we can ensure that the algorithm determines whether the test-trace can be found in the model. To elaborate this in depth we will define the cost function as the following:

```
testContext

costFunction(typeOfMove, targetEvent)
    switch(targetEvent)
        case "synchronous move"
            return 0
        case "Log/Trace move"
            return Infinity
        case "Model move"
            if (testContext.includes(targetEvent))
                return Infinity
            else
                return 0
```

To summarize the assigned cost, synchronous moves are allowed.

Moves in the trace are not allowed and are assigned cost infinity, that means the alignment-algorithm will return recursive calls instantly which contain a trace/log-move.

Moves in the model are only allowed if the activity which is to be executed is not in the context of the test.

In order to understand how this cost function ensures that if a alignment is found (the algorithm does not return infinity), this alignment is a *valid* trace. We will recall the alignment examples from section 4.1.2 regarding the example graph from Figure-4.

Valid in this case means, that the found trace is corresponding to the test-trace when projected onto the context. If such a trace is found by the algorithm we know that the test will pass if its polarity is positive (vice versa for a negative test).

T	A	D	C
M	A	>>	C

The above example does leave the graph accepting. Though will the algorithm with the above defined cost function never return this alignment, as trace/log-moves are not allowed (return infinity). These are the moves where the >>/no move symbol is in the lower half of the table. Hence its denoting a no move in the model-trace. Recall that this trace, denoted by M, is the sequence of activities which are executed in the model in order to obtain a alignment. This is also the trace the algorithm returns, together with the minimal cost, if a "optimal" alignment is found.

When having the above in mind it does intuitively make sense that these trace-moves not are allowed, as this means the found trace will leave one of the test-trace activities out. If that is the case we would not be able to use this result for our open tests, as we cannot check whether the given test-trace exist.

It is to be noted that synchronous moves are always allowed, as we need those to find a trace in the model which does represent our test-trace. Synchronous moves are the core of having two traces aligned, in this case the test- and model-trace.

However the interesting case is the one of model-moves. Consider this complete alignment which refer to the graph of Figure-4:

T	A	D	>>	C
M	A	D	B	C

Further assume that this alignment was found when model checking the graph in order to evaluate the positive open test:

$$\big([A, D, C]\{A, D, C\}\big).$$

This alignment is indeed a "optimal" alignment as any alignment which is returned from the algorithm with the given cost function is in such senses "optimal", as both the positive and negative open test only need to evaluate whether the test-trace exist in the model. By that any found alignment with the described cost function is optimal.

Besides being "optimal" does this alignment contain a model-move. As the activity B is not in the context of the test, a model-move with this activity is allowed. This distinction is important, as when we project the found alignment-trace $[A, D, B, C]$ onto the context $\{A, D, C\}$ we derive the wished test-trace $[A, D, C]$. From this follows that the positive test must hold.

However if we would change the context of the test, such that the positive open test is defined as the following:

$$\big([A, D, C]\{A, B, D, C\}\big)$$

Then the algorithm would not be able to return the above alignment. This is because the model-move with B is not allowed as it is contained in the context. This makes sense as the trace $[A, D, B, C]$, which we would get, projected onto the context $\{A, B, D, C\}$ will stay the same and by that not living up to the given test trace $[A, D, C]$.

In essence, activities which are in the context can only be executed by synchronous moves. These activities, which are marked as important, need to be aligned with the test-trace. This also ensures that the algorithm does not return any traces which contains duplicates or *wrong* order of activities from the context in its model-trace. *Wrong* in this case means that it is not equal to the test-trace.

However all activities which are not in the context can be executed/model-moved arbitrarily often, as when projected onto the context we omit to look

at those.

It is to be noted that the cost function captures this property, as if the algorithm returns with cost 0 (a trace is found), then we know this trace only contains synchronous moves or model-moves with activities not in the context. Thereby it ensures correctness of the important activities, and by that we do not even have to project the found alignment-trace from the algorithm onto the context. That is because we know that if a alignment is found it must live up to the test-trace.

It is to be noted that the above is defined formally in [2]. Another thing to mention, which is addressed in the paper, is that the pure alignment algorithm would work with a computed upper bound for the cost in order to ensure its efficiency. However this is not possible in this use case, as we are working with a cost of infinity. To tackle this challenge there is a formal definition which keeps tracks of the *Reachability by model-moves* in [2]. It is to be mentioned that the definition is exploiting the fact that the only way executing a sequence, in order to enable more activities of the context, is by model-moving non context activities.
However it is still to be noted that the alignment algorithm works best and correct in most cases if its depth is bound. It is suggested in [2] that a depth of 100 seems to works efficiently while ensuring correctness. However when needed, a run with no bound is acceptable as well, if one wants to invest a couple of seconds per test.

4.2.2 Alignments influence on the project - Importance of syntactical checks

In order to understand how the model-checking solution with the help of the alignment algorithm did change the focus and course of the project we have to look at the underlying assumptions of the test-driven modeling approach and the system Slaats. et al have defined in [1].

The test-system did barely scratch the model-checking task. There was a clear focus on defining syntactical checks, such that the model-checking task only should become prevalent in certain cases. The idea was to handle the model checking with a rudimentary algorithm, as the importance of syntactical checks where believed to be much greater.

However this assumption was challenged when looking at the model-

checking for negative open tests in a more detailed fashion. It is to be noted, that whereas positive tests do not have to exploit the whole search space of a graph all the time, as they can return as soon as they found any trace satisfying the test, negative test must exploit all possibilities all the time. This is because negative tests are ensuring, that there does not exist any trace in the model which is satisfies the test-trace.

Recall the behaviour of re-executing events and by that having changes in enabled events and the marking of a graph. This behaviour might lead to loops which might enable new traces and by that keeping track of all traces and there possibilities may produce a greater overhead then expected. This is not a discourse containing a rudimentary model checking algorithm anymore, it evolved into a proper task for itself.

The question, whether this thesis-project should focus on the model-checking task for itself or if there should be implemented a soft version of model-checking while also including the syntactical checks, came up. Regardless, we began to realize that we had to reduce our expectations of a full-fledged test-system.

The solution to the model-checking task is described in the previous section. When realizing that we could use the work of alignment [2] and by that having a effective way of exhausting (to a high degree) the state space of a DCR-graph, the importance of model-checking and the syntactical checks shifted. As we now would be able to do the general model-checking in a effective manner, it became less important to minimize its frequency with the help of having the syntactical checks.

It is to be noted that for a optimal test-system syntactical checks still have there reason. For the scope of this thesis-project we chose to lay the focus on implementing model checks with the alignment algorithm and having a useful Ui for handling tests. The syntactical checks are kept for future work (section 8).

5 Implementation

5.1 Structure

In order to look at the application from a implementation perspective it is to be noted that there are three work spaces.

- **electron**: Which contains the functionality regarding the interaction with the OS, such as saving and loading files.
- **types**: Containing all the type definitions used in the application.
- **web**: hereunder:
  - **components**: Which contains the web-application functionality, as the different views and the modeling environment.
  - **helpers**: Code to support functionality as the swarm/co-op modeling feature or calculations used in the application.
  - **styling**: Does contain css files for the layout of the application.

The extensions I was to implement are found under the components in web and in the type workspace. It is to be said that the type workspace only contains a single type file, which was extended with type definitions such as a formal DCR-graph or a Test type.

The component folder did contain files which represents the views or states of the application and how they are handled. There is not much to elaborate here, as the application does contain a front-page/menu and the modeling canvas, which is represented by the *Designer* component. This single component does by far contain the most functionality regarding the modeling of a graph.

This file is also where most of the work for this project was done. The reasoning behind this is that the data of the graph is stored here. In React components, such as the *Designer*, can have internal states. These states contain data used in the component, often when a state is changed, a re-rendering of the corresponding data is triggered. This data/these states can be connected to an Ui-element (often JSX elements), which then is to be updated.

As the formal DCR-graph has to refer to the state of the modelled graph and the tests have to refer to this formal representation of the model, it seemed reasonable to place this functionality here.

It is to be noted that the component folder has been extended with new files, such as components for the user interface regarding the usage of tests and the files contained in the supplied alignment module.

5.2 Creating a formal DCR-graph

The creation of the formal DCR-graph is a essential part of the test system. Before any test-cases can be evaluated this graph has to be created, it is used for model-checking and will be used for syntactical checks.

The implementation is located in the *Designer* component. The creation is triggered by a click on a button. Note that the choice of having this as its own action, and not a underlying function call when triggering the execution of tests, is a conscious decision. It was observed that when having it under the same button as the test execution, there were runs were the application was executing the test before the initialization of the graph was finished. This might be due to underlying low level behaviour of the machine when executing instructions. Additionally it also seemed to be convenient to have these functionalities distinguished in this raw state of the application. Recall that the syntactical checks, which might be implemented in the future (section-8), do have to save and look at different versions of the formal graph. In order to keep track when developing this functionality it might be easier to have separated buttons, rather then one full fledged trigger which contains a lot of different tasks.

In order to show a reasonable amount of detail the following sequence diagram will show which functions are involved when creating a formal DCR-graph. Afterwards there will be a elaboration on small challenging details.

Note that this a a non-formal more functional sequence diagram which is only used for illustrating the implementation. By that buttons are denoted in blue, functions are characterised with yellow and local-states do have the color green.

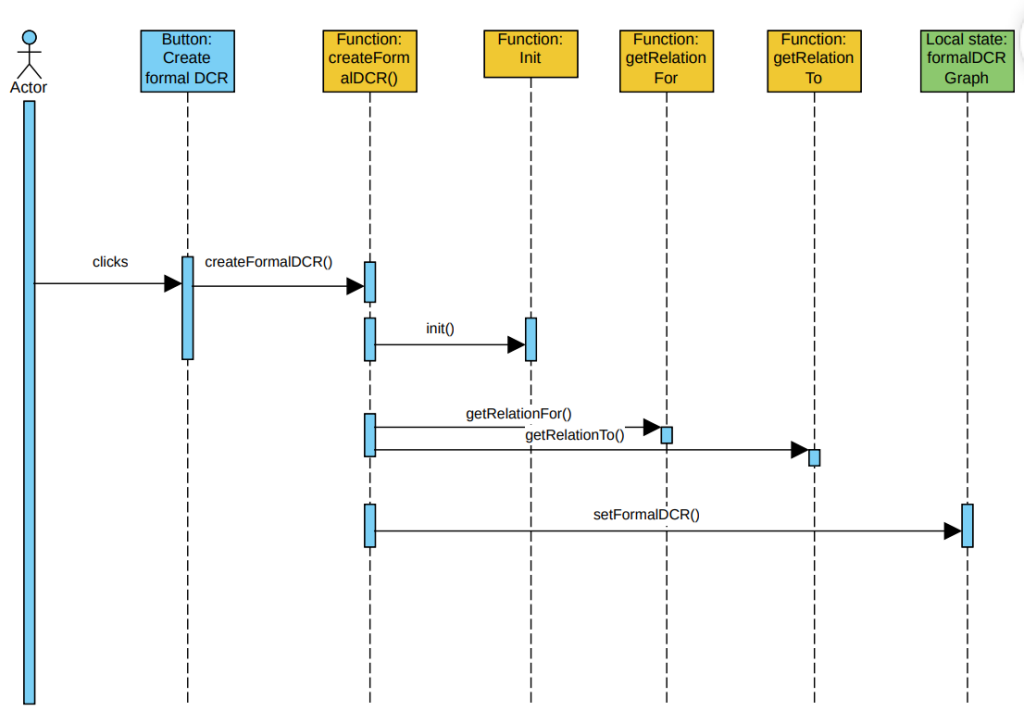

Figure 6: Sequence diagram of creating a formal DCR

To summarize the sequence of function calls which are triggered when the user presses the button "Create Formal DCR". The function *createFormalDCR* does as first call *init*, which initializes everything needed for the alignment algorithm, hence the model checking. This is done here, as a natural follow up on creating the formal graph is to execute the test cases.

Next the functions *getRelationFor/To* are called. These functions are used to extract the information from relations of the graph. This will be elaborated in a bit. For now its only important to mention, that these functions do not return anything, they simply manipulate the given object. This object is initialized in the calling function (*createFormalDCR*).

At the end of the flow the function *setFormalDCR* is used to set the DCR-graph object, created during the function call, to the local state which contains the formal DCR. This local state and its data is lying the the *Designer* component.

In order to understand the described procedure closer, we have to take a look at the type definition of a formal DCR-graph:

```
256  export type Event = string;
257  export type Label = string;
258
259  export interface Marking {
260    executed: Set<Event>;
261    included: Set<Event>;
262    pending: Set<Event>;
263  }
264
265  // Map from event to a set of events
266  // Used to denote different relations between events
267  export interface EventMap {
268    [startEventId: string]: Set<Event>;
269  }
270
271  export interface DCRGraph {
272    events: Set<Event>;
273    conditionsFor: EventMap;
274    milestonesFor: EventMap;
275    responseTo: EventMap;
276    includesTo: EventMap;
277    excludesTo: EventMap;
278    marking: Marking;
279  }
280
```

Figure 7: Type definition DCR-graph

As we can se on line 271-279 does the formal DCR-graph contain a set of all events. This is straightforward to initialize, we simply run through all the events in the model and do safe there ID's in this set. Be aware that a type *Event* in this case is only a string. This definition is a part of alignment algorithm and it was necessary to change the type signature for the whole

application in order to incorporate this module (this is discussed in detail in 5.3). Anyway in order to mention it here, recall the grasping of the structure of the application in 4.1.1. It was stated that the Ui has its own layer in the application. So do the types representing the Ui-events and relations. The formal graph simply refers to these objects ID's.

Returning to the type definition, the marking (line 278) does simply contain sets of events for there corresponding marking. This is also linear to initialize and it done in the same run through all the events.

A little more trickier is the *EventMap* type. In essence this is a dictionary which has the start event ID as key and a corresponding set which contains all the events this type of relation is pointing to.

However the important detail lies in the word "for" and "to", by those the relations are divided into two groups. And with that also how the the *EventMap* dictionary is used.

EventMaps used for "for" relations do have the end event of a relation as key and its set contains all the events which is a condition/milestone for this event. In order words searching in this dictionary with a given event ID as key, will give you all the events which are possible direct prerequisites for this event.

In contrast a *EventMap* used for the "to" relations, has the start event of a relation as key and the set contains all the events this relation is pointing to.

This important distinction is something I had to come back to during the project. It is a example of a typical challenge in this thesis. Keeping the theory in mind while taking implementation decisions and being fairly new in the domain. I had to re-think some decisions and understanding as I had the alignment module to incorporate. This module was of course also build upon assumptions, which had to be aligned with the extensions which where to be implemented.

Another example of this kind was, that there for each event must be an instance in the dictionaries. Even if the events do not have any relations of that kind, the alignment algorithm requires this. This is done when running through all the events simultaneously, here the marking and the event set is created as well.

Hence the *createFormalDCR* function does run through all the events of the model once and does run through all the relations once, where the *getRelationFor/To* are called.

5.3 Incorporate alignment

When a formal DCR-graph is created, tests can be executed. At this point in time tests are only evaluated by model-checking, which is done with the alignment-algorithm.

Minor Things which where to consider when the alignment algorithm was incorporated:

- As mentioned in section 4.1.2 the correct execution of the milestone relation was to be handled. The alignment algorithm is based upon functions which simulate the execution of a DCR-graph. Thereby functionality as executing a event, getting all enabled events or checking whether the graph is accepting is supported. These functions can be manipulated in order to fit different versions of the DCR-graph notation. In order to implement the milestone relation a additional check in the **isEnabled** function was implemented. Namely checking whether a milestone event is included and pending.

- Overlapping type signature: The alignment module had its own type definitions. Some of those where overlapping with the ones from the application. This lead to the type signature of all user-interface elements changing, namely they are now explicit marked to be "Ui". All non duplicate types where added to the type file of the application.

- Different versions of java script: The application was using a older version of js then the alignment module. Updating the workspace did not cause any additional trouble.

It is to be noted that all files regarding the alignment module has been added to the *components* folder in the workspace *web*.

The execution of tests gets triggered via a button. Then the *modelChecking* function is called. Have a look at the following functional sequence diagram:

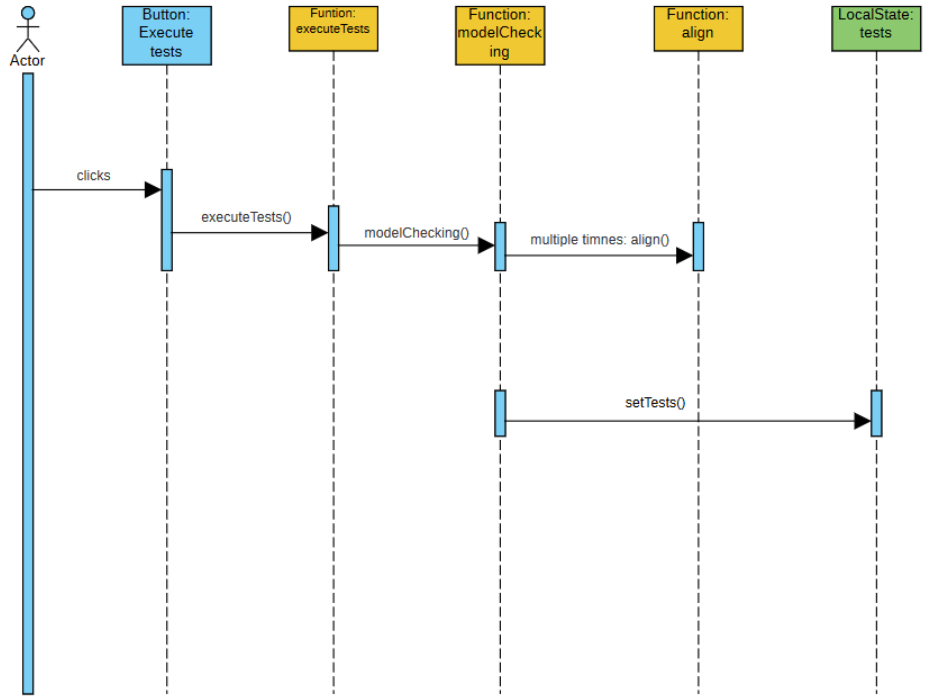

Figure 8: Sequence diagram - model checking

This is a straight forward sequence. The user presses a button and the function which is handling the execution of tests is called. From here syntactical checks would be initialized as well but at this point in time only the *modelChecking* function is called.

This function is located in the *Designer* component, where also the tests are stored as local states. The function then iterates through all tests. In each iteration a local cost function is defined, which refers to the tests context.

Then the alignment algorithm is called for each test-trace with the corresponding cost function. Interesting to note it that when evaluating the result of the alignment algorithm the test object itself stores whether it passes or not.

This is reasonable as the test objects are stored as local state in the *Designer* component. The *modelChecking* function iterates through these objects and when a test objects pass-identifier changes, we instantiate a new object. A important distinction about Reacts behaviours is that, if one

modifies a object directly it will not be updated on the screen automatically. In order to keep the user-interface and the internal states in sync, React checks whether the objects are the same, hence the identifiers are checked.

This is a pitfall I have overcome in a refactoring session. At that point the Ui-components for the tests did only show the test results when toggled manually on the screen. In order to fix this issue I copied the local state, manipulated that copy and set this new instance as the local state. React discovers that the identifier of the local state is changed and re-renders the corresponding Ui.

5.4 Tests and a useful Ui

5.4.1 Test object

In order to understand how test objects are defined we look at the figure below.

```
302
303  export interface Test  {
304      readonly id: string;
305      trace: UiEvent[];
306      context: Set<UiEvent>; //set
307      polarity: string;
308      deleted: boolean;
309      passes: number; // -1 = default not executed, 0 = false, 1 = true
310  }
311
```

Figure 9: Test object

The implementation of a open tests contains: a unique ID, a trace which is defined as a array of Ui-events, a context which is a set of Ui-events as there cannot be duplicate in a context, a polarity (negative or positive) and a number representing whether the test passes or haven't been executed yet.

When the user wants to define a test, events can be added to or deleted from the active trace/context through the right-click menu. To illustrate the implementation look at the user interface:

Figure 10: Defining a test

The *Designer* does have local states representing the "active" trace and context. These are represented in a component which fetches all the functionality needed for creation of tests, this interface component is returned in the designer. The components visual representation is shown in the figure above.

The user can only define one test at a time. When the test is specified and polarity is chosen the user can submit the test with the corresponding button. By that a object of the *Test* type is instantiated and added to the list of tests, which also is a local state. These are the tests which are iterated through when model-checking. Afterwards the "active" test is reset.

5.4.2 User interface for usage of tests

When a test is submitted the test is shown in the *TestBar* which is returned as a JSX element in the *Designer* and is its own component.

It is to be noted that the user interface for the usage of tests is defined by a hierarchy of multiple components. These have all been added to the component folder of web.

In order to supply a intuition for the implementation observe the next two images.

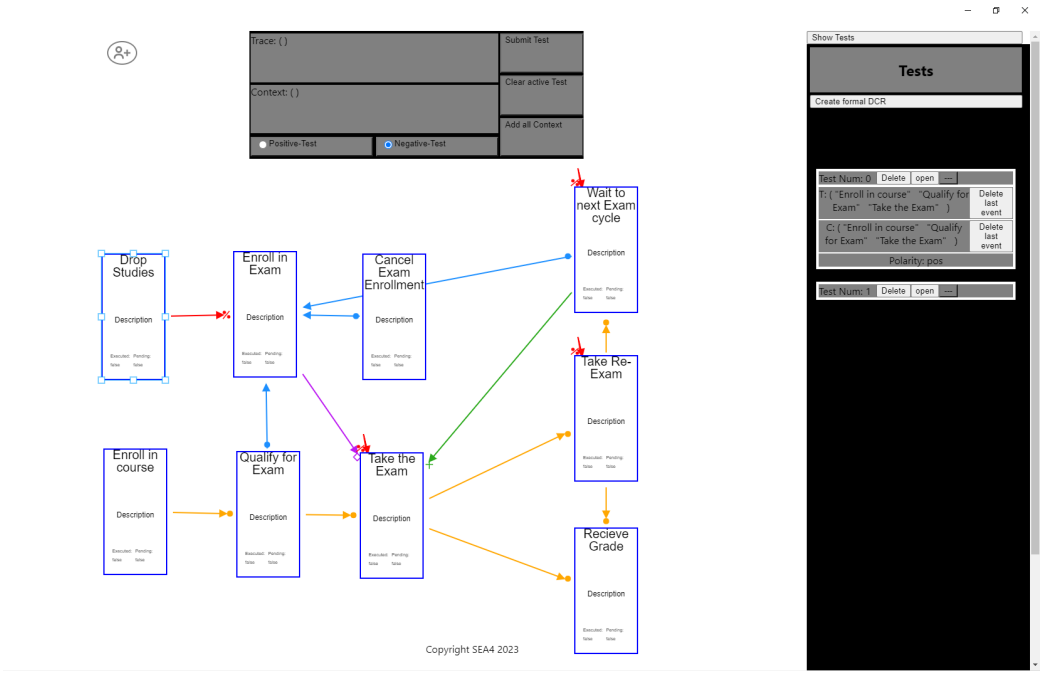

Figure 11: Image of application

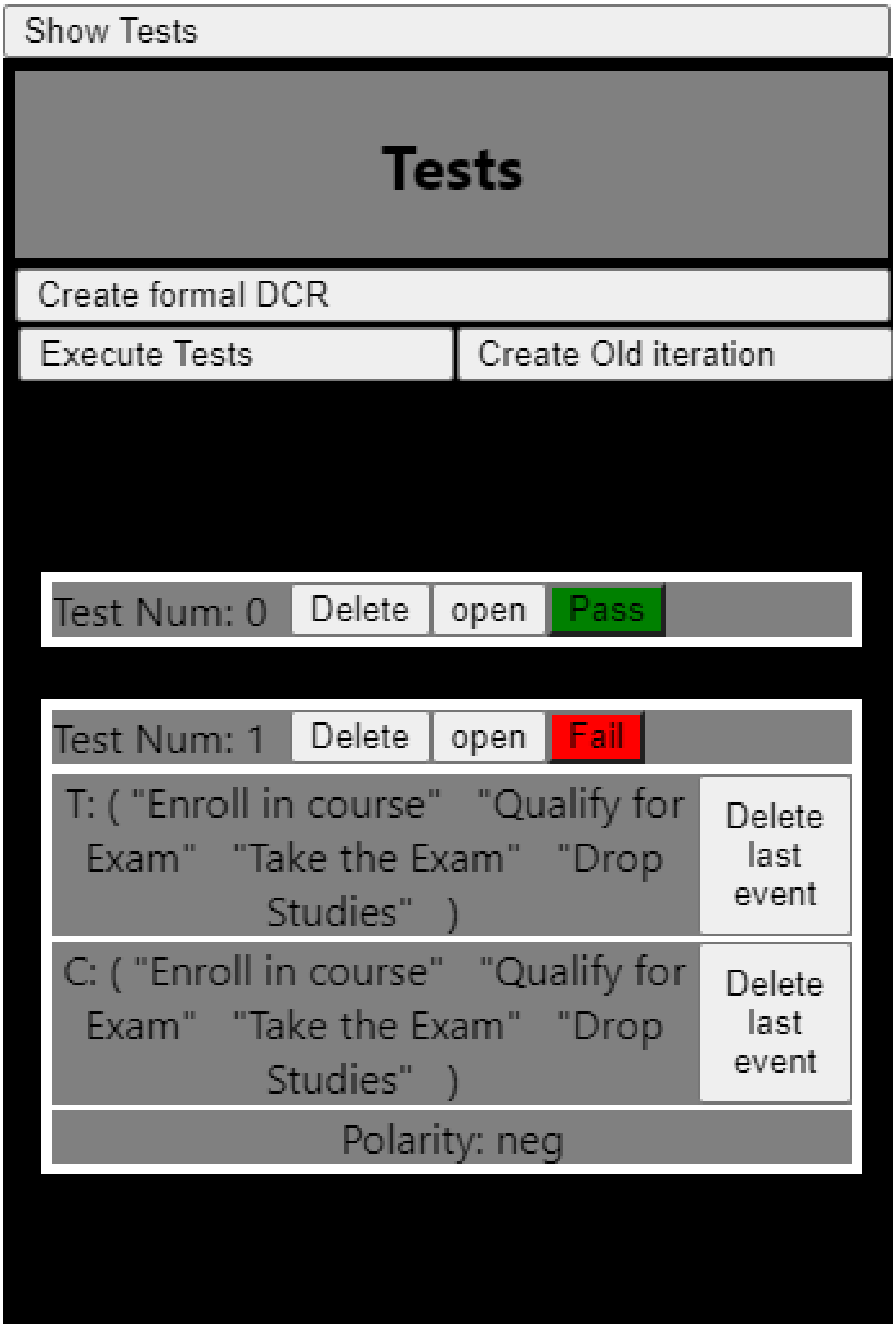

Figure 12: Image of TestBar

It is to mention that the button "Execute Tests" and "Create Old iteration" only are shown when there has been created a formal DCR (They can not be found on the first image). This is a interface design decision in order to prevent any errors. Additionally making it intuitive for the user, that a formal DCR must be instantiated before being able to execute tests or saving a "old" iteration. The later is a feature leading up to the future work (8) section and will be elaborated there.

Observe that the interface-components representing the tests not only contain visual representations but also functionality such as deleting or unfolding tests.

To note is, that when deleting a test in the user-interface the local state in the *Designer* component is changed. This is done by passing props to down the component hierarchy. These props can be functions or local-states. The below screenshot shows the props passed to the *TestBar* component.

```
/>
<TestBar
  formalDCRCreated={formalDCRCreated}
  open={testsOpen}
  setOpen={setTestsOpen}
  tests={tests}
  deleteTest={deleteTest}
  deleteLastTrace={deleteLastTrace}
  deleteLastContext={deleteLastContext}
  executeTests={executeTests}
  createFormalDCR={createFormalDCR}
  createOldIteration={createOldIteration}
/>
```

Figure 13: Props passed to TestBar

Here the *testsOpen* and *formalDCRCreated* prop represents Boolean val-

ues which decides whether the *testBar* is unfolded and whether the discussed buttons are shwon.

The *tests* prop represents the test objects stored in a local state, this state is given, as the Ui component representing the tests needs a way to refer to the tests.

The others are functions defined in the *Designer* component which contain different functionalities, such as deleting a test.

In order to understand how a function is passed down to a lower order component in the hierarchy, it seems convenient to show a piece of design where the hierarchy becomes obvious.

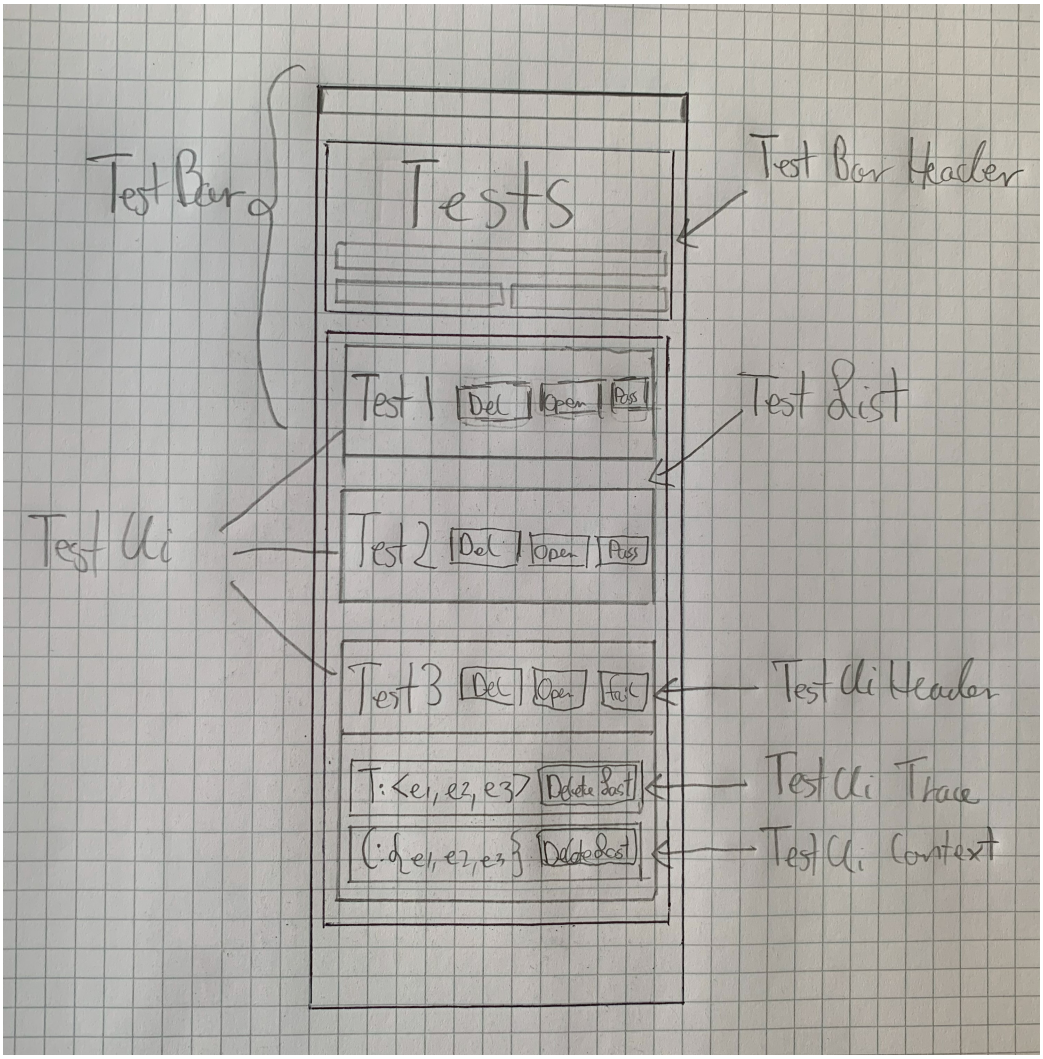

Figure 14: Ui design

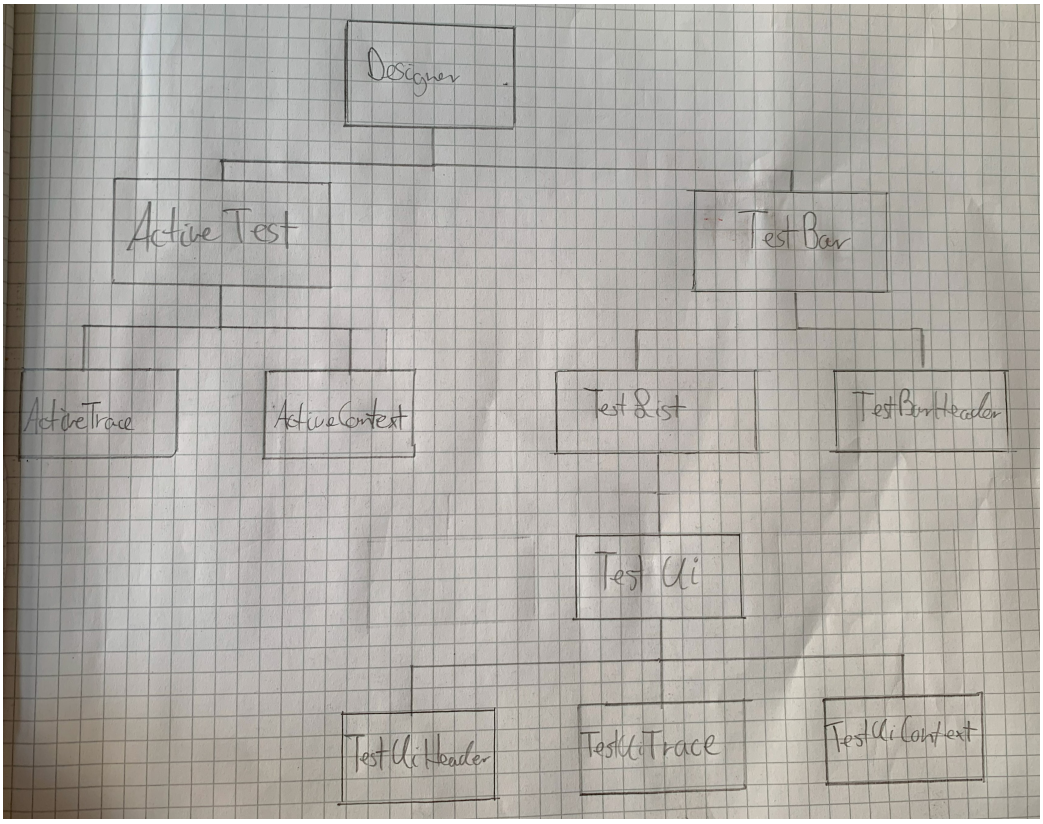

Figure 15: Hierachy of test Ui

In order to show how such props which are propagated down the hierarchy are used we consider the *TestUiHeader* component. Here is the corresponding code snippet:

```
21  interface Props {
22      givenTest: Test;
23      i: number;
24      show: boolean;
25      setShow: (val: boolean) => void;
26      deleteTest: (val: string) => void;
27  }
28
29
30  const TestUiHeader = ({givenTest, i, show, setShow, deleteTest}: Props) => {
31      return(<Main>
32          <> Test Num: {i}  </>
33          <button onClick={(e) => deleteTest(givenTest.id)}> Delete</button>
34          <button onClick={(e) => setShow(!show)}> open</button>
35          { (givenTest.passes == -1) ?
36              <Default> ---</Default> : ((givenTest.passes == 0) ?
37                                          <NotPassing> Fail </NotPassing> :
38                                                      <Passing> Pass </Passing>)
39
40          }
41
42
43      </Main>)
44  }
45
46  export default TestUiHeader
```

Figure 16: TestUiHeader

As we can see are the props extracted in line 21-27. When the button which deletes the test is triggered the function *deleteTest* is called with the corresponding test ID. This call then gets propagated all the way back to the *Designer* where the function does manipulate the local state.

When that has happened the *TestList* component will re-render its test as, the local state is refers to is changed.

Furthermore can we from the code-snippet above see how the internal pass-identifier of a test is having influence on what is rendered. In figure 11 and 12 we can see that test interface is changing depending on whether it the passes or not. Line 35-40 in the above code-snippet does show how this is done. This is a simple construct called conditional rendering (a else-if statement).

The other components use a similar structure to extracts props, use propagated functions or rendering based on conditions.

5.5 Minor refactoring

In order to prepare for future work, especially the task of implementing syntactical checks, it was essential to implement the saving of tests. This was done in a refactoring session. The application will now also save the tests corresponding to the model when the file is saved.

For the scope of this project it is only to be mentioned, that this was done by declaring a new object type which contains the graph and the tests. This could then be passed to the existing functionality.

Now it should be possible to follow the TDM approach by creating a "old" iteration from the model and the corresponding tests. This iteration is kept and when extending the model, the new and old graph is compared in order to see whether syntactical checks apply.

A last refactoring goal which is kept for future work with this application, is the user-interface. Even though at this point in time the application has a decent and useful interface for the test-system, some design work might be done for improving the user-experience. Hereunder could the representation of the activities in a test trace or context be improved. This may include CSS work for visual representation but also the idea of having each event in a trace represented by its own component/instance. This might include features as clicking on these events and deleting them from a trace.

6 Learning and Underlying challenges

The main work of this thesis can be defined by a process of gathering knowledge in order to be able to do justified implementation work.

A similar process is described in Naur's article [4] where he is reasoning about that the main task of programming is building the theory behind the program. The *theory* in this process captures a theoretical understanding of the program. Hereby knowledge is gathered on how the program is applied in a real world scenario, its connection to its domain specific context. Furthermore it is also key to understand what the single parts of the program are doing and understanding how the program might be extended.

This definition is a fine way to capture my learning process throughout the

project. I was *building theory* [4] while working with the implementation. An additional the starting point for my learning was the paper which described the formal test system [1].

Hence the first task of the thesis was understanding the domain of business process modeling, specifically DCR-graphs. So that the formal definitions defining the syntactical checks could be understood. However this knowledge became less relevant when the course of the project took a shift towards the model-checking task with alignment. This was the next *theory I had to build* [4]. Even if not working with programming itself, understanding how the supplied algorithm would work and solve the model checking was a rewarding learning process.

Even if some of the formal knowledge gathered in the start of the project might have fallen of my mind again due to limited work-capacity, the understanding of reasoning by using formal mathematical definitions is something that I have come closer to in course of this project.

In addition to this learning, it was a necessary task to learn and grasp a fundamental understanding of front end development and specifically for this project the library React. Even if this is not certainly captured by Naur's definition [4] it is definitely closely connected to understanding the structure of the supplied application.

Grasping the structure of the application in order to understand a way of extending it is fine aligned with the *theory-building process* from [4]. This was another main task of thesis-project and has offered a learning gain in the specific domain of the application itself and the conversion from formal definitions to implementation.

This is leading to the next general aspect of this thesis, namely the approach of a proof of concept, or more broaden "Theory vs Practise".

It was a typical challenge to have the theory in mind while taking decisions during extension of the application. There occurred mismatches between what I thought of a good decision and what actually was a good decision regarding the domains context. Example wise defining certain data-types in the "right" way. Sometimes it was hard to take a decision in the context of DCR-graph when only having worked with this domain a couple of months.

Especially when it was time to fetch the supplied alignment algorithm with the application. Now there occurred minor clashes on how I had thought it was a good idea to implement things compared to PhD. student Axel, which

worked with this domain for quite a long time. Hence taking a foresighted implementation decision with the theory in mind was not always the most straightforward thing to do.

A example of this would be the relationship dictionary as discussed in the foregoing section.

However overcoming this challenge by diving into details of decisions and being particular about the specifics was a rewarding process in terms of learning. Having to work with concrete theory and justified assumptions from third parties ensured a precise handling of the tasks at hand.

7 Evaluation

7.1 Use-Case

In order to evaluate the work of this thesis I will demonstrate the application and its usage for a real-world work-flow. It is to be noted that in a business set-up the models may be considerably bigger and more complicated. The model discussed is a toy-example and was chosen in order to show some interesting properties and functionality - it should not be seen as a robust example of how the notation may be used in the industry.

The selected use-case demonstrates a students workflow in order to take an exam at KU. The model captures the process from enrolling in a course, qualifying and taking the exam. When not considering special cases as severe illness, every student has at maximum three attempts in order to pass an exam. This behaviour is captured in the model and we elaborate on the details in a bit.

In order to demonstrate details of the DCR-graph notation and how open tests in combination with those can ensure specific constraints, we will be discussing the use-case and its model.

After discussing details of the model it will be showed how open tests are designed. All that is done with the tool developed during this thesis-project. Thereby will this section also serve as a demonstration of how the scientific-prototype may be used.

7.2 The Model

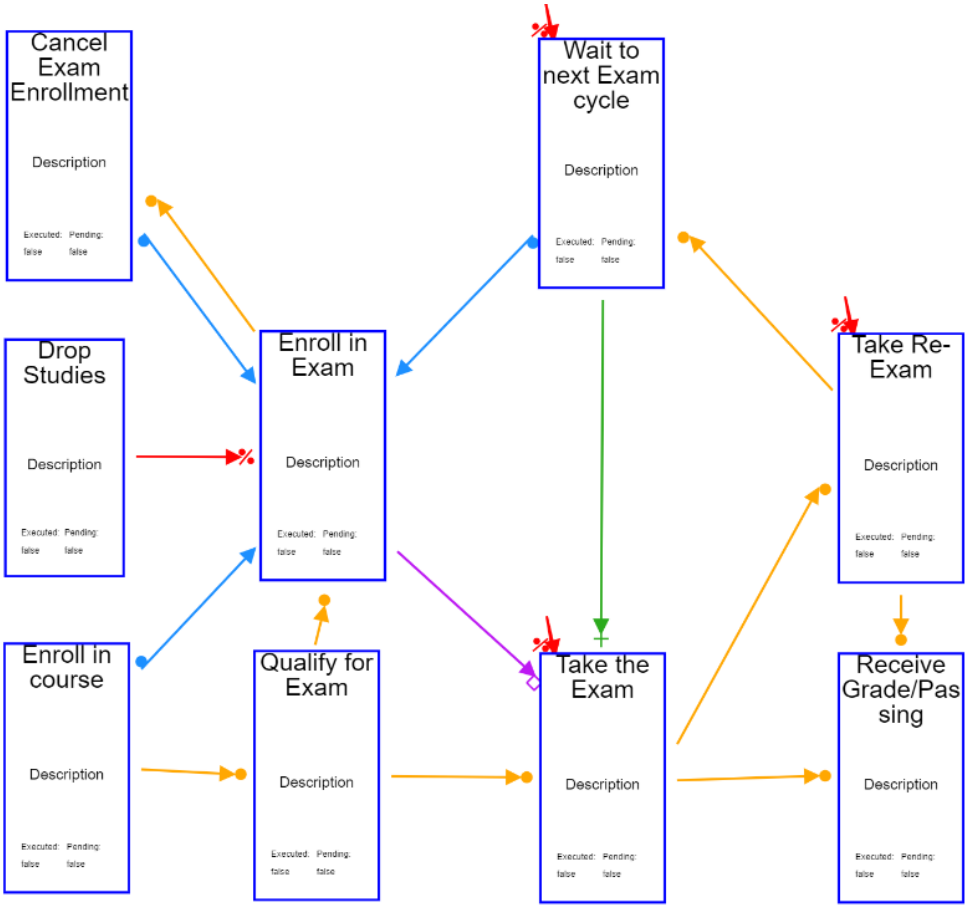

Figure 17: Use case model

Observe that there are some linear conditional dependencies in the model. The straightforward way for a student to pass a exam, hence receiving the grade is through enrolling in course, qualifying for the exam, taking the exam and then passing. This is modelled via the condition relations (orange).

However when enrolling in a course, exam enrollment becomes imminent after qualifying for it. This is modelled with the response relationship (blue). As soon as this response is triggered the **Enroll in Exam** activity is pending and the milestone is activated. By that enrolling in a exam is necessary

before taking the exam.

Note that there is a pitfall in the model which allows us to evade this requirement. Observe the exclude relation from **Drop Studies** to **Enroll in Exam**. If we execute this activity and the exam enrollment is excluded, the milestone relation is no longer activated. By that it would be possible to execute the activity **Take the Exam** even if there has not been executed an enrollment for the exam.

The TDM approach and the open tests with a specified context can help discovering pitfalls as the above. Misconceptions or simply errors in the model can be revealed and solved. This example is perfectly covered by test number 0 and 1 (7.3.1). Note how a additional single activity in the context changes the outcome of the test, by considering the pitfall described above (event: **Drop Studies**).

Ideally constraints are defined with tests to ensure the correctness of a model. However a more subliminal, implicit use-case is that open-tests can be of support when navigating through a model. Tests can provide a more thoroughly perspective of the state-space.

Having such a tool may especially be a help when considering more severe processes, where such pitfalls can cause security lacks.

Returning to the model, we know discovered that the exclude relation from **Drop Studies** to **Enroll in Exam** causes a side-effect. This may be insignificant for this particular example, as when dropping out of your studies it seems irrelevant to take a exam or considering any activity related. However in another setting a loophole around requirements may cause severe damage.

The reason for the exclude relation is explained by the goal of having a *accepting* (4.1.2) model. As soon as the **Enroll in Exam** activity becomes pending it is not possible to have an accepting graph if this activity is not executed or excluded. Hence in order to make it possible to drop out of one studies at any point in time this event will exclude the possibility of having the exam enrollment pending. This enables traces such as shown in test number 2 (7.3.2).

The last detail of the model worth discussing is the fact that every student can at maximum take a exam three times. The model does insure this

by having the events **Take Re-Exam** and **Wait for next Exam cycle** excluding themselves. These can never included again. Test number 3 and 4 (7.3.3)show the correctness of this property.

The **Wait for next Exam cycle** activity does have a include relation to **Take the Exam** and a response relation for **Enroll in Exam**. This ensures that when waiting for a next exam cycle one is only able to take the exam again when enrollment is done again as well. Test 5 (7.3.4) and 6 (7.3.5) show this property and they will be elaborated on in a bit.

7.3 The Open Tests

In order to avoid a unnecessary overhead, only tests are presented which underline the above discussed properties. Thereby will it become clear how positive and negative tests can be used to specify and to ensure certain behaviour in a model.

7.3.1 Test Number: 0-1

Test number 0 and 1 are two negative tests which are designed to show the pitfall regarding the dropping of one's studies. Both contain the same trace:

Enroll in Course, **Qualify for Exam**, **Take the Exam**, **Receive Grade**

Observe that the activity **Enroll in Exam** is not contained in the trace. These are negative tests and by that want to check that the given trace does not exist.

The contexts of both tests do contain all the activities in the test trace plus the activity **Enroll in Exam**. This is important, as the tests motive is to ensure that one can not take the exam and receive a grade if not enrolled in the exam. The test is meaningless if **Enroll in Exam** is not represented in the context.

Recall the property that important activities are contained in the context, so that they cannot be model-skipped. Hence they are important for the projection of the found trace to the context, which should lead to the given test-trace.

The outcome of the negative tests 0 and 1 is differing. This is because test 0 does consider the event **Drop Studies** in its context. Test number 1 does not and by that it is possible to find a trace which projected onto the context give us the test-trace. This negative test is failing as the loophole of dropping the studies and then taking the exam is exploited.

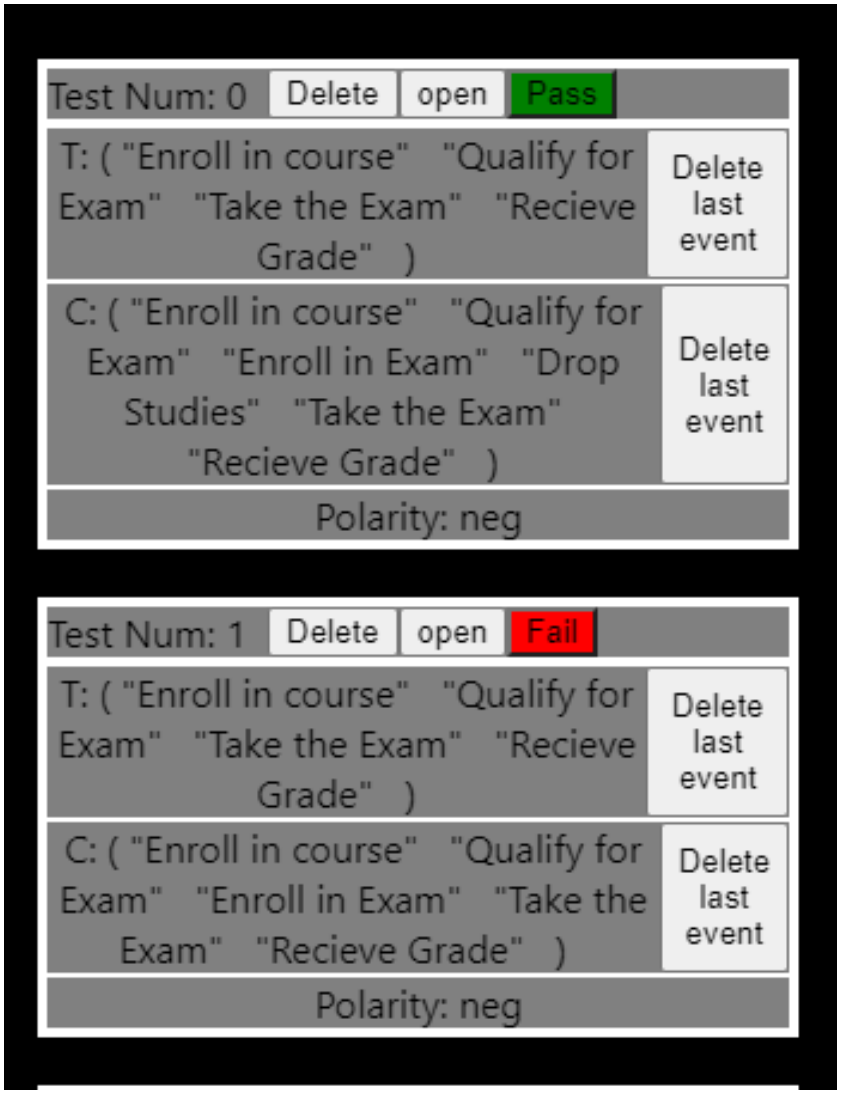

Figure 18: Outcome test 0 and 1

7.3.2 Test Number: 2

Test number 2 is a straightforward positive test which goal it is to show that we can abort our studies at all time and still leaving a accepting graph/trace.

The trace and the context are identical and are showing the possibility of taking the exam, failing the re-exam and after waiting for the next exam cycle dropping the studies. It is passing as shown below.

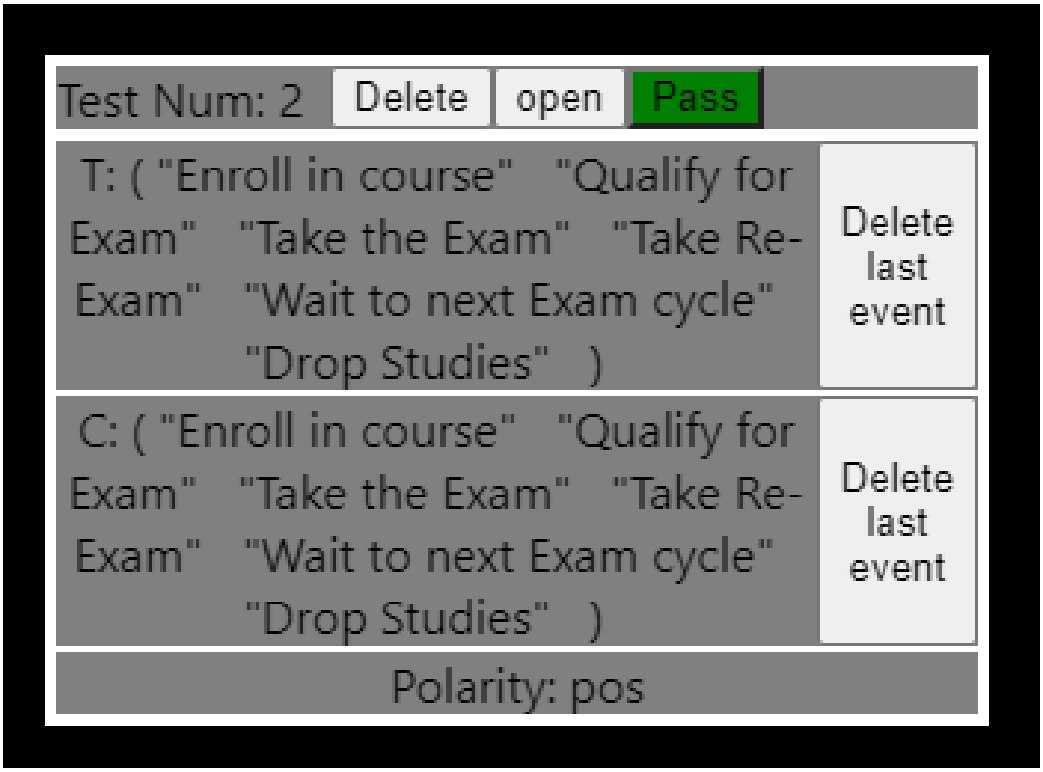

Figure 19: Outcome test 2

7.3.3 Test Number: 3-4

These are two negative tests which are checking that an exam at max can be taken three times. Any subsequent shot on taking a exam or re-exam should not be possible. As we can see in the figure below is it not possible to find a trace violating this constraint.

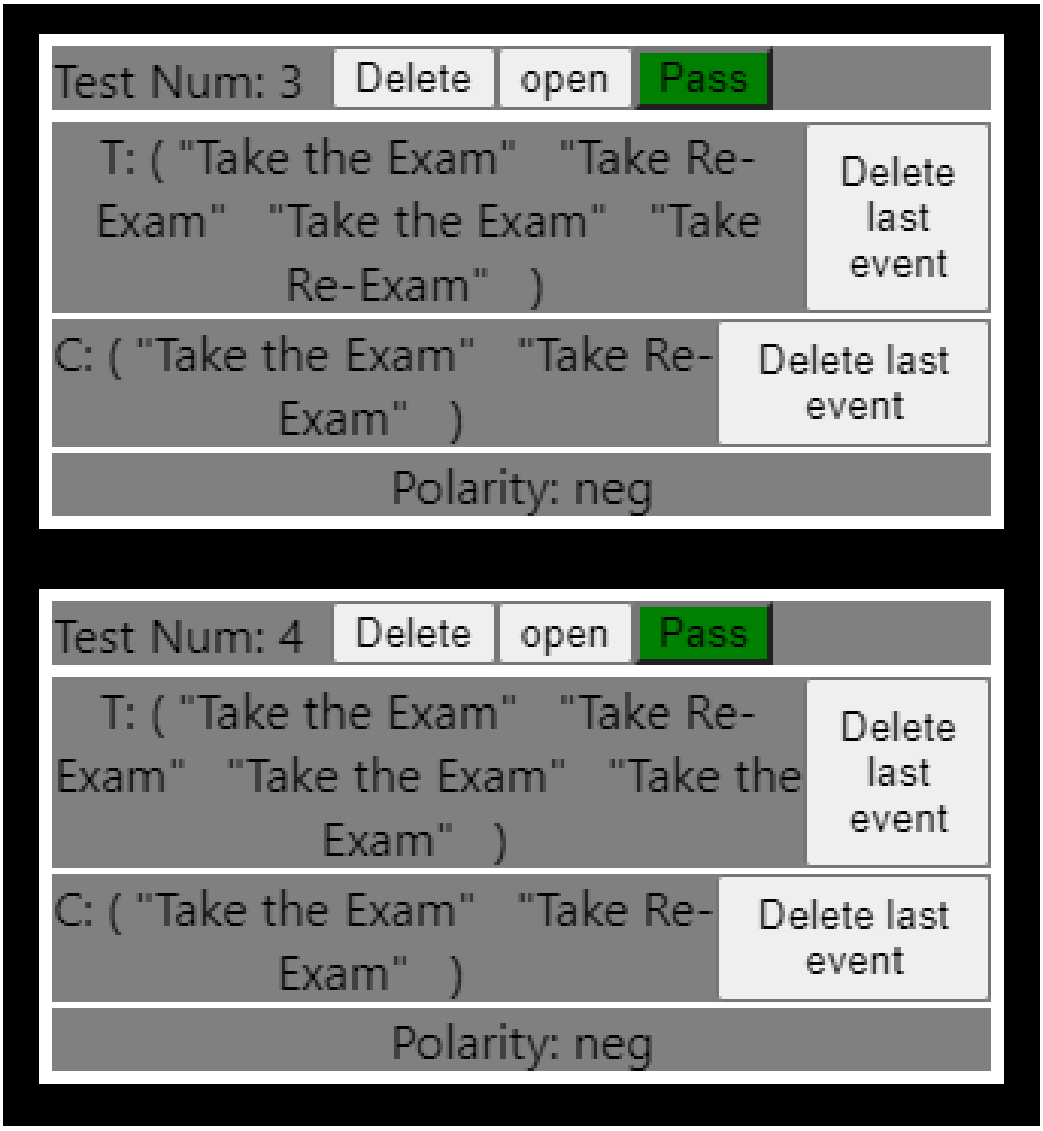

Figure 20: Outcome test 3 and 4

This makes intuitive sense as the **Waiting to next Exam cycle** event has an exclude relation to itself and no in-going include relation. The same is true for the **Take Re-Exam** event. By that these can only be executed once. Additionally the Exam event does also exclude itself when executed. It is only possible to include the event again by executing the **Waiting to next Exam cycle** event. As this is only possible one time is becomes clear

that the property of maximum taking three shots for a exam is ensured.

7.3.4 Test Number: 5

Test number 5 is a negative test which shows the property that it is necessary to enroll in the exam when taking it a last third time.

In order to demonstrate the implicit usage of open tests it is interesting to note that we omit all activities in the trace which are not directly relevant for what we want to ensure.

The test has the trace:
Enroll in Exam, **Take the Exam**, **Wait to next Exam cycle**, **Take the Exam**

The context contains the following events:
Enroll in Exam, **Take the Exam**, **Wait to next Exam cycle**, **Drop studies**

This negative test passes, as there can not be a trace which executes the exam a second time without skipping the prerequisite (milestone) outgoing from the **Enroll in Exam** event. This can either be done by dropping the studies or executing the pending enrollment activity. That is why those activities are contained in the context.

7.3.5 Test Number 6:

Test number 6 is working with the same property of the model. This time it is a positive test which checks that it indeed is possible to take the exam a third time after waiting for the next exam cycle, implied that the **Enroll in Exam** event is executed in each cycle. The outcome and specific definition is shown below:

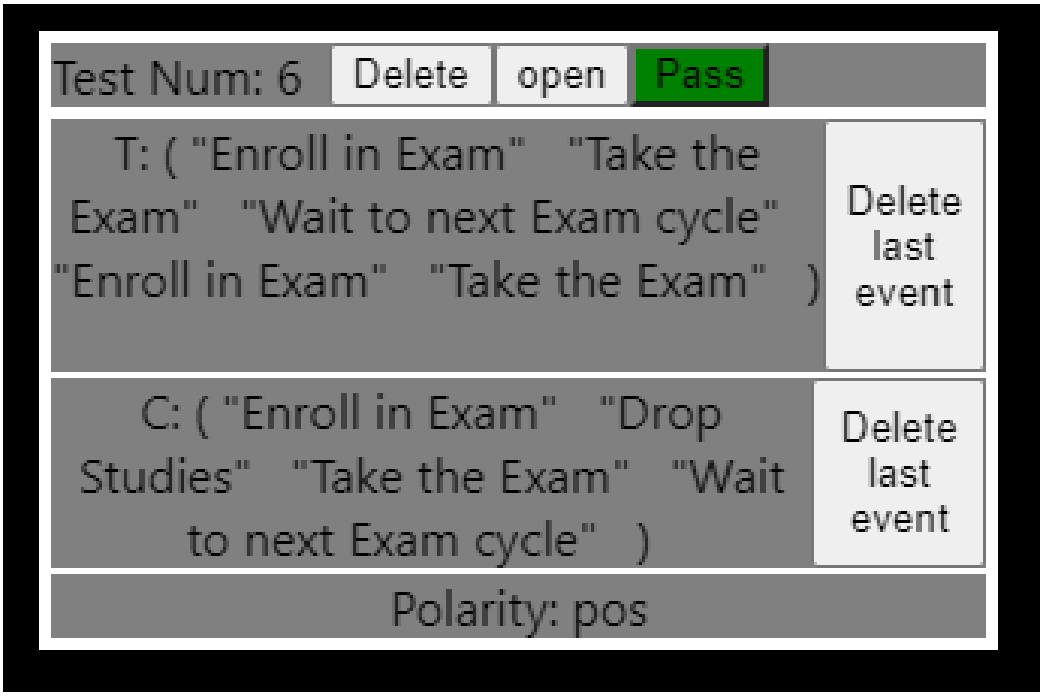

Figure 21: Outcome test 6

7.4 Concluding the use-case

As shown in this section the scientific prototype has a good usage for the TDM-approach, where constraints can be defined by tests. It was observable that both negative and positive open tests have there use and work correctly. As the tests have shown does the application simulate the DCR-graph notation as intended.

The fact that the open tests can be used to maneuver and explore the state space of a graph opens a new conclusion. Assuming that expert users in any business process notation will be unlikely to introduce pitfalls as the one in the discussed model, new users might introduce such possible dangerous behaviour more often.

TDM with open tests may give non-expert users a tool to ensure correctness of their model while boosting their learning curve, as they can dive deeply into certain functionality of the given notation. It is to be mentioned that the tool also has its benefits for experts users and may ease their work.

8 Future Work: Syntactical checks

As observed throughout the project, the syntactical checks or the static analysis for the model are yet to be implemented in a future iteration.

In order to do so one has to refer to the formal DCR-graph. By that it would make sense to place this functionality in the *Designer* component where the internal state representing this graph is stored.

The static analysis concepts introduced in [1] need to refer to the "old" and "new" graphs discussed in 3.1. These concepts also introduce a new form of graph, called "dependency graph". It should be said that storing all these object in the same component would be reasonable. However as the *Designer* component already is overloaded, it is left to a future iteration to evaluate whether it would make sense to move all the formal representations to another component (examplewise the *testBar*).

The saving of tests is implemented which enables one to create test-iterations which contain a formal model and the corresponding tests. At this point in time this is manually done by clicking on a button. However when the graph is saved, these test-iterations are not saved. This would simply produce a unnecessary overhead, as the formal graph can be instantiated in linear time at any given point and the test are stored by default.

Therefore the application (in this development state) relies on the user to know when to create an "old" iteration. It will be remained open whether is would be sensible to make this a automated process. This simply on decisions regarding the future implementation.

However it is to be noted that there was observed some struggles during this project when trying to implement automated functionality which have a clear structured order of execution. Such as the need to create a formal DCR-graph before executing tests. It seems like function calls and instructions are not always executed in a synchronised step by step one threaded manner. This can of-course makes sense depending on underlying behaviour, heads up.

A last thing to consider in future work is the user-interface design. In section 5.5 there are mentioned some concrete ideas. However dedicated time for designing the user-interface would be sensible, as this fell short of priority

in this thesis-project.

References

[1] Tijs Slaats et al. “Open to Change: A Theory for Iterative Test-Driven Modelling*”. In: *Business Process Management* (2018), pp. 31–47.

[2] Axel Kjeld Fjelrad Christfort and Tijs Slaats. “Efficient Optimal Alignment between Dynamic Condition Response Graphs and Traces”. In: (2023).

[3] Hildebrandt et al. “Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs”. In: *Post-proceedings of PLACES 2010. EPTCS* 69 (2010), pp. 59–73.

[4] Peter Naur. “Programming as Theory Building*”. In: *North-Holland Publishing Company, Microprocessing and Microprogramming* 15 (1985), pp. 253–261.