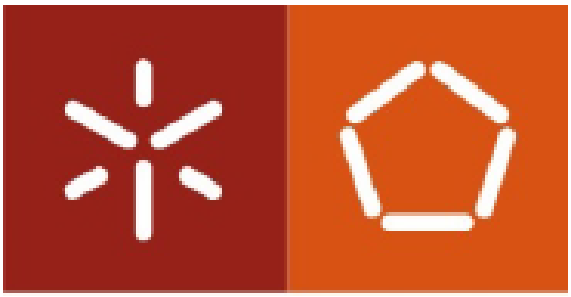

University of Minho

Integrated Master's Degree of Software Engineering

Software Engineering Laboratories

Modelling and Formal Specification of Web Application

Carla Cruz a80564

Ricardo Pereira a73577

August 23, 2021

Contents

1 Introduction **2**

2 Approach **3**

3 Requirements **4**
3.1 Informal Requirements 4
3.1.1 Regular User Interaction 4
3.1.2 Administrator Interaction 5
3.2 Formal Requirements 6
3.2.1 Regular User Interaction 6
3.2.2 Administrator Interaction 9

4 Design **13**
4.1 Mockups 13
4.1.1 General Design 13
4.1.2 Regular User Design 15
4.1.3 Administrator Design 18
4.2 ConcurTaskTrees Environment 21
4.2.1 Regular User Design 22
4.2.2 Administrator Design 22

5 Analysis **24**
5.1 *SWAP* model behaviour 24
5.1.1 Regular User Interaction 24
5.1.2 Administrator Interaction 29
5.2 *SWAP* model facts and assertions 38
5.2.1 Regular User Interaction 38
5.2.2 Administrator Interaction 39

6 Conclusion and Future Work **40**
6.1 Future work 40

References **41**

1 Introduction

Nowadays, people see a web application as a single system where, in the great majority of cases, a user introduces his credentials and has access to all of his data. Obviously, this kind of systems are much more complex than that and building one can be a very hard task to complete. Actually, we can even say that the construction process never ends, since it is always evolving and there are always mistakes and faults that must be corrected. The reason why these errors happen is directly related to the freedom that the languages in which the applications are built gives us. Anyone who has ever used a web application must have encountered at least one bug, no matter how small or important it was. This becomes even more interesting with this fact: thousands of new applications are released **every day** [1].

In any branch of engineering it is necessary to plan and draw the final product. If we think in building, let us say, a doghouse, immediately, there is a need to doodle what you want to build so that we can see what will be needed or not. What material will be needed? How many? What are the dimensions? What if in the future we want to change something in it? There are **a lot of questions that only appear when the project is being discussed and dissected** and the answers must all be registered so that mistakes do not happen during the doghouse construction. The problem gets bigger if we consider a building. In software engineering, applications are like buildings in which bricks are pieces of programmed code and where it is possible to break and put them together again without any cost. That is the problem that we are facing: **programmers use this flexibility to build and dismantle at their own will** the applications, trying different approaches and implementing one that worked in the great majority of those fast tests.

An ordinary person would ask "is it that easy to build an application?" and a developer would probably answer "that depends on the type of application you want, one full of errors and working sporadically or the other that will always work and will eventually have one problem or another". Clearly, the first option will be built up more quickly than the second because **a fully operational application requires a prior and thorough study of the requirements** and of the design solution to be built, in another words, it requires time to think about all the possible states that it can have. Yet, time is something that companies do not have because they think it will be wasted and they rather use it in building the app.

2 Approach

The problem discussed before was what motivated us to find a formal tool that would allow us to model a web application in order to make the process of building these types of systems more effective. There are a few tools very well known by the formal methods community that would allow us to model them very accurately such as: ***TLA+***[2], ***NuSMV***[3], ***Alloy***[4], ***Electrum***[5], ***ConcurTaskTrees Environment***[6], etc. We already had the opportunity to try them all, but given the circumstances we decided to test just two of them.

So there are many ways to build a formal model like this but we decided that the best would be to take an existing application and specify it. This way we would be approaching both aspects of applications in general and aspects of that application in particular. The one we are talking about is called *SWAP* and **its purpose is to provide students of educational institutions a platform that allows the exchange of shifts** of certain courses to which they are enrolled. As expected, this forces the platform to have some kind of authentication so that the student can know the shifts in which he is enrolled, the proposed exchanges and the requested exchanges. It virtualizes the discussions students would need to have to exchange a shift, something that in reality can be messier than it appears to be. Also, it ends up being fairer since the application establishes an order to all exchange requests. *SWAP* is definitely an excellent application to model because **it has some aspects that the great majority of web applications has too**. As an example, the log in aspect is something that cuts across most applications but on the other hand shifts exchange is not. With that said, the strategies used here may also be useful in modeling other applications. Furthermore, we will also introduce some additional features that we think this application could have.

Earlier, we said we would use two of that set of tools to do some experiments and find the one that best fits to the actual problem. These were *Electrum* - an extension to the *Alloy Analyzer* which is a structural modeling language based on first-order logic and that allows its users to generate and prove invariants, among many other features, all of this using **set theory** - and *ConcurTaskTrees Environment* - an environment for editing and analysis of task models very useful to support design of interactive applications such as the one we are studying. The first one is very versatile because **it allows us to specify as much as we want** (as well as a programming language) and even though the visual component is very rudimentary, it exists, unlike *NuSNMV* and *TLA+*. In comparsion, *ConcurTaskTrees Environment* is something much more visual that shows what and how tasks are being done, therefore the modeling process is not as liberal as in *Electrum*. These were the reasons that led us to choose *Electrum*, **a tool that produces models based on mathematical properties**. Yet, a *CTTE* model was also developed in order to model user interaction with this application.

3 Requirements

Along with what the great majority of experts say about building functional and trustworthy software, requirements analysis is one of the first things that should be done and also the one that **defines the relation between the client and the seller**. Having well-defined requirements helps everyone to know specifically what kind of product is going to be built, and **the more accurate they are, the more similar will be the wanted and the actual outcome**. With that in mind, the following section describes the most important aspects of the wanted outcome, those that cannot fail and that are extremely important for the idea to be well materialized.

3.1 Informal Requirements

The following requirements are divided by the type of problem they fit in. As an example, the interactions of the user with this web application, in terms of what he is supposed and allowed to do, represent a major and general problem. This same problem is detailed in 3.1.1.

3.1.1 Regular User Interaction

- A regular **user should log in before starting to use the main tools of the web application**. To do so, the **institutional email address** as well as the **associated password** must be introduced by the user.
- If a user forgets his password he should be able to ask for help to **recover** it. This help could be a clickable button that redirects the user to a web page where **he is asked for his institutional email address** to send an email with a link. This link leads to a web page where the user should introduce again is institutional email address and define the password he wants from now on (in this process, **he must confirm the chosen password**). The submission is either successful or unsuccessful.
- A successful login should lead the user to a **web page with a list with the courses he is registered in** - the *main* page. All entries will have for each course, information about it, as well as the **course name**, **the current shifts in which the student is registered**. A drop-down menu where he can choose the wanted shift for any course (if the wanted one is available) will be available in *courses* page.

    1. As said, this drop-down will have **all the shifts that are available to exchange for the shift the student is registered in**, but this means that someone must have already expressed their willingness to switch shifts, otherwise there wouldn't be any exchange available.
    2. So, this dropdown must include in its options a button to insert the wanted shift if it is not available.
    3. In the first state of the application, note that for all of the users there shouldn't exist any other option than the one that allows to insert a new shift, once no one has done it before.
    4. For each course, there should exist the **current/actual shift** the user is registered in, the **incoming exchanges** proposed by other users and the **requested exchanges** proposed by the actual user. As an example, if a user in course "A" enrolled in shift "1" submits a request to exchange with someone enrolled in shift "2" (course "A"), everyone enrolled in shift "2" will have in their incoming exchanges shift "1" and the system adds shift "1" to the requested exchanges of the user who submitted the request. If some other user accepts this request, the requested exchanges of the student who started this whole process loses shift "1", the actual shift changes to "2" and the actual shift of the student who accepted changes to "1". Also, incoming exchanges of any user enrolled in shift "2" will lose shift "1", only if no one else has requested the same exchange (shift "1" for shift "2").

5. If a user accepts an exchange or an exchange he requested is accepted then **all of his requested and incoming exchanges must be reseted** since all of them were related with the previous "current shift".
6. Further more, **there will not exist repeated drop-down options**, that is, if there is a student "A" that is in shift "1" and wants the shift "2", a student "B" that is in shift "2" and wants "1" and a "C" that is also in "2" and wants "1" (assuming that "B" and "C" were the first to express the will to exchange), student "A" will have only one option to switch to shift "2" along with all the other shifts options. Then, if student "A" decides to accept an exchange for shift "2", he will change with the first student that submitted the request to switch shift "2" for "1".
7. The previous feature (no repeated options) implies that **there will be no student names associated to the exchange**.
8. Once someone submits a request to exchange a shift, as soon as someone else accepts it (there must appear a warning message if the user really wants to do it), **the change is immediately done**. This avoids sending a confirmation message to the student that submitted the request (which ends up being a waste of time).
9. Exchanges acceptances must be done once at a time, this means that the user must choose which shift exchange he wants to accept and accept it and only after that he would be able to accept other exchanges.

- This user should have a **notifications menu** where notifications will appear about exchange requests related to the shifts he is registered in.
- There are **students that have a special condition** - those that are student workers. As it is covered in the great majority of institutions, they have the right to choose the shifts that they will attend and for that reason, there should be a feature that allows these students to choose shifts in a certain period of time, something that is decided by the application administrator. This students should be able to choose these shifts before the allocation for the rest of the students occur, since the first ones have priority.
- There should exist an **initial page that precedes the login page** to introduce what is *SWAP* and only then, the user would be able to visualize the login page, after clicking **sign in button**.
- After logging out, the application must confirm that the user session has ended, followed by the **initial page**, which can also be seen as the "final page".

3.1.2 Administrator Interaction

- The administrator should log in before starting to use the application.
- If the administrator forgets his password he should be able to recover it, clicking a button that redirects the administrator to a web page to do this.
- The administrator should be able to see the available courses and to add shifts to the course as needed.
- The administrator should be able to see all the students. He can choose one and see the information about him like his name, the courses he has, and the shifts of courses that he is registered to.
- The administrator should be able to change the shift of a student and this action should increment or decrement the number of students on each shift.

- The administrator has a list of students with special conditions and submits this list on the platform to be recognized on the application. Then the administrator activates an option that makes students with special conditions able to choose the shifts that they want on their courses.
- After the choice by the students with special conditions and this information gets on the app, the administrator should activate the option that attributes the shifts to the rest of the students.
- The administrator should be able to see the history of the shift changes.

3.2 Formal Requirements

This requirements are nothing less than the previous client requirements organized and written in *Electrum* language, the one we choose to build this model. The idea is to **turn those informal goals into "codable specifications"** that can be related to each other and define all possible states application can have.

As we already know, in this type of formal tools, we tend to specify more and more, which is good - **the more specified the model is, the better it represents the system** - but thanks to this, it ends up having lots of lines of code. To fight this problem, we decided to divide the specification and build a model for each subsection of *3.1 Informal Requirements*. In each model, we will explain the signatures, the variables inside them (that can also be seen as relations) and the predicates we have established. For example, *signing_in* is the type of predicate that occurs the most in this model. They are firstly defined by the previous state of the elements and secondly by the next state of the elements, in another words, first we specify the preconditions and then we must say what happens to all of the model variables, otherwise *Electrum* will assign random values to all of them. Note that we do not need to specify the preconditions of each variable, but doing so makes this model more formal.

3.2.1 Regular User Interaction

Course (signature)

This signature represents a *Course* which can have multiple relations. It must have one that indicates the current shift, so if a *Course* exists, it will always have one and only one current shift. A *Course* can also be related to some (or none, since we are talking about a set) *requested_exchanges* and/or some *incoming_exchanges*.

Page (signature)

In this model, we will not have multiple web pages like it happens in a web application. Instead, we will see a *Page* as something that represents the *SWAP* application, where a bunch of elements appears and disappears according to the address bar state. If the address bar is set to *login* state, then the login button will be "visible" as well as some other pages elements like username and password. Here are the variables of this signature:

- ***addr_bar*** - this relation indicates the address bar state (only one state is allowed) that will always exist;
- ***username*** - represents a text box where the user inserts is username that can be *empty*, *valid* or *invalid* and is an element that will only be visible in the log in page;
- ***password*** - this variable has the same behaviour as username;
- ***login_button*** - represents the log in button;
- ***forgot_button*** - represents the button to access the account recovery page;

- ***send_email_button*** - represents the button to confirm the need of help that will be sent to the inserted username (the institutional email);
- ***is_logged*** - a visual element that tells us if the user is logged in or not;
- ***signin_button*** - button to set the application in log in page;
- ***courses_button*** - button to set the application in *courses* page;
- ***logout_button*** - button to set the application in log out page;
- ***main_button*** - button to set the application in *main* page;
- ***accept_button*** - button to accept an incoming shift exchange;
- ***courses_list*** - shows all of the user's available courses.

AddrBar(enum)

Enumerates all the possible states that the address bar can have.

ButtonState (enum)

Enumerates all the possible states that a button has available.

FieldState (enum)

This *enum* is used to represent the states of elements like text boxes.

Booleans (enum)

Represents the states that a boolean can have - *true* or *false*.

Shifts (enum)

All the available shifts are here defined, so if it becomes necessary to increase or decrease the shifts, this is the *enum* that should be changed.

actual_shift_remains_the_same (predicate)

Prevents current/actual shift from changing from one state to another. If the intention is to run a model with requested exchanges and incoming exchanges constantly in change (as well as it happens in reality), then the last two lines must be deleted. This predicate is a complement to others where this happens.

incoming_exchanges_acceptance (predicate)

Acceptance of one and only one exchange requested by another user. This predicate is called after accept button is clicked and it tells us:

- there must exist some incoming exchanges in the actual course;
- the next state value of the actual shift for that course must be in the previous state values for the incoming_exchanges;
- there are no requested or incoming exchanges in the next state;
- all the other courses but the one that has changed must remain with the same information they had before.

force_requested_and_incoming_to_exist (predicate)

Forces all Courses to have requested and incoming shifts. This is another predicate that is used in others.

no_exchanges (predicate)

A predicate that tells us that a *Course* does not have any relation. In the application, that would mean the list of courses with no shifts.

init (predicate)

This predicate defines the initial state of the model. Reading the specification we can tell that the address bar is in "initial" state and there are no other elements available besides sign in button which is unclicked. *no* word indicates that his following element is not active.

signing_in (predicate)

Translates to *Electrum* the act of clicking in sign in button. The only change happens in the state of *signin_button* variable that is set to *clicked*.

initial_login_transition (predicate)

initial_login_transition reproduces the transition between initial and login pages, where some elements appear and other disappear. So, when we are inside login page we have the adress bar in *login*, an *empty* username, an *empty* password and two log in and forgot unclicked buttons. The *actual_shift_remains_the_same* is the predicate referenced above that is used here to ensure actual shift does not change.

logging_in (predicate)

When the user is logging in, it will certainly fill the username and password text boxes which will be valid or invalid and he knows it by clicking in log in button. Otherwise, at this point, the user can also say he forgot is credentials and click forgot button to recover them. The predicate has an *or* operator that distinguishes this two scenarios.

wrong_login (predicate)

The *wrong_login* predicate is called when inserted username and/or password are invalid; in this case the log in page is reseted.

login_main_or_recover_transition (predicate)

There are two ways of getting out of log in page: by going to recover page (if the user previously clicked in forgot button) or to main page (if the user inserted valid credentials and clicked log in button). Going to the recover page implies a username text box to insert the email to which will be sent the reset link and a button to confirm that intention. In comparison, the transition to main page makes courses and respective shifts visible to the user.

recover_credentials (predicate)

When the user is inside the recover page, he must introduce an email that will be *valid* or *invalid* and click in send button.

recover_inital_or_recover_transition (predicate)

After clicking send button, if the email address inserted is *invalid* the application will reset recover page in order to introduce the email address again. If the user inserted a *valid* email address, then the application will be redirected to initial page.

inside_main (predicate)

This predicate defines what happens when the user is inside main: he can either log out or go to courses pages, where he can perform exchanges, if any is available. Again, *or* operator is allowing one of these two options to be chosen by the user.

main_courses_or_logout_transition (predicate)

As said before, from main, user can log out or go to courses page, and here is where one of these transitions occurs. Logging out means there will not be any button available and going to courses means one more button becomes available: the accept button.

before_courses_changes (predicate)

This represents the application state when it changes to courses page.

before_shifts_exchanges (predicate)

If some incoming exchanges are available in some course (something that is being checked in the line that has "some" quantifier), the user may accept one of them and this means clicking accept button in order to perform the exchange.

after_shifts_exchanges (predicate)

"after_shifts_exchanges" is called when accept button is in the state "clicked". The only elements that change here are those related with the course where the exchange happens and the accept button that is set to "unclicked", everything else stays the same - this is done by calling "incoming_exchanges_acceptance", which was explained before.

after_courses_changes (predicate)

After the courses page returns to its "normal" state the user can log out or return to main page and he decides it by clicking in one of the respective buttons.

courses_logout_or_main_transition (predicate)

Similarly to what we have seen before, this predicate translates the transition between course pages and log out page or main page, depending on what button he clicked before.

inside_logout (predicate)

This represents the application state when it changes to log out page.

logout_init_transition (predicate)

It is very common to see web applications that get stuck after logging out. This predicates ensures this can not happen, redirecting *SWAP* to its initial state after logging out.

skip (predicate)

Finally, we reached the last predicate: *skip*. It may not seem, but this predicate is just as important as the others, since it reflects when nothing happens and application remains indefinitely in the same state.

3.2.2 Administrator Interaction

Page (signature)

In this model and to represent the administrator interaction, we create the signature Page as something that represents the *SWAP* application. This Page could have many elements (for example buttons and table) and this appears and disappears according to the address bar. If the address bar is set to the Main state, then the buttons of the navbar will be "visible" appearing as unclicked. Here are the variables of this signature:

- ***addr_bar*** - the address bar state is defined by this variable and will always exist.
- ***username*** - this represents a text box where the administrator inserts his email. This text box can be empty, valid, or invalid.
- ***password*** - this text box, like the username, can be empty, valid, or invalid.
- ***is_logged*** - this variable tells us if the administrator is logged in or not.
- ***login_button*** - represents the login button.
- ***forgot_button*** - represents the forgot button and when it is pressed, goes to the Recover Page.
- ***send_email_button*** - represents the button who sends an email to recover the password.
- ***signin_button*** - represents the button who take the administrator to the Login Page of the application.

- ***courses_button*** - Button to set the application in Courses Page.
- ***history_button*** - Button to set the application in History Page.
- ***specialS_button*** - Button to set the application in Special Students Page.
- ***studentList_button*** - Button to set the application in Students List Page.
- ***chanShift_button*** - Button to change the current shift.
- ***profile_button*** - Button to set the application in Profile Page.
- ***addShift_button*** - Button to add a shift.
- ***addList_button*** - Button to add the List of Students with special conditions.
- ***atvShift_button*** - Button to activate the shifts.
- ***history_table*** - The table gives the history information.
- ***students_table*** - The table gives the students information.
- ***special_table*** - The table gives the special students information.
- ***courses_table*** - The table gives the available courses information.
- ***profile_table*** - The table gives the profile of one student.
- ***logout_button*** Button to set the application in Initial Page and make the administrator logs out.
- ***main_button*** - Button to set the application in Main Page.

After the variables defined, these need to have the values that can be attributed to them. So we have the enumerates which are used to represent the states of elements.

AddrBar (enum)

All the possible states of the address bar: initial, recover, login, main, courses, specialStudents, studentsList, history, profile

ButtonState (enum)

All the possible states of button: clicked, unclicked

Boolean (enum)

This enum represents the states that a boolean can have: true, false

FieldState (enum)

This represents the states of elements, for example, text boxes: empty, invalid, valid

TableState (enum)

The table can exist or not, so we represent the state of the table with this enum: noexist, visible

Predicates

init

This predicate defines the initial state of the model. The address bar is in the "initial" state and the Sign-in button is unclicked. There are no other elements available.

signing_in

This predicate represents the act of clicking in the sign-in button (the variable is set to click).

initial_login_transition
This represents the transition between initial and login pages. Inside the login page, the address bar changes to log in, and the fields of page login are empty. We also have the login and the forgot buttons.

logging_in
In this predicate, the administrator can fill the username and password text boxes which will be valid or invalid. Otherwise, at this point, the administrator can click on the forgot button to recover the credentials.

wrong_in
This predicate is called when the username or password (or both) are invalid.

login_main_transition_or_recover
On this stage, the administrator can go to the recover page (if clicked on the forgot button) or to the main page (if the credentials are valid and clicked login button).

recover_credentials
At the Recover Page, the administrator needs to introduce his email (valid or invalid) and click the send button.

recover_initial_or_recover__transition
If the email address is invalid the application will reset recover page in order and the administrator needs to introduce the email address again. Otherwise, the application will be redirected to the initial page.

courses_click
This predicate represents when the administrator clicks on the courses button.

courses_transition
After the administrator clicked on the courses button, he goes to the Courses Page.

students_click
This predicate represents when the administrator clicks on the students button.

students_transition
After the administrator clicked on the students button, he goes to the Students Page.

history_click
This predicate represents when the administrator clicks on the history button.

history_transition
After the administrator clicked on the history button, he goes to the History Page.

add_shift
This predicate represents when the administrator clicks on the addShift button. He can add shifts to the courses, so this predicate is able when the application is on Courses Page.

atv_shifts
This predicate represents when the administrator clicks on the atvShifts button. He can activate the shifts of the students.

atv_shifts_transition
After the administrator clicked on the atvshifts button, the students are able to see the shifts of their courses.

change_shift
This predicate represents when the administrator clicks on the changeShifts button. With this action, he can change any shift of any student.

profile_click
This predicate represents when the administrator clicks on the profile button.

profile_transition
After the administrator clicked on the profile button, he is able to see information about the student he picked.

return_to_main
This predicate represents when the administrator clicks on main the button.

return_to_main_transition
With this predicate, the administrator returns to the Main Page.

specialStudents_click
This predicate represents when the administrator clicks on the specialStudents button.

specialStudents_transition
After the administrator clicked on the specialStudents button, he can see the list of special students or he can submit the list.

specialStudents_subList_click
This predicate represents when the administrator clicks on the addList button.

specialStudents_subList_transition
With this predicate, he adds the add to the server, and then he could see the list of students with special conditions.

specialStudents_atvShifts
This predicate represents when the administrator clicks on the atvShifts button.

logout_click
This predicate represents when the administrator clicks on the logout button.

logout_transition
After the administrator clicks on the log out button, he goes to the initial page.

skip
This predicate reflects when nothing happens and application remains indefinitely in the same state.

4 Design

The design of the application is something really important to the client once it is the only thing he sees. Nevertheless, **a beautiful and nice design does not define the quality of an application**. In fact, we can even have the entire design drawn and apparently operational without having anything working in back end. So, those who develop the application must guarantee that the relation between back end and front end is consistent.

4.1 Mockups

Thinking in these aspects, we decided to gather everything we had related to design: the mock ups and the current *SWAP* pages appearance. Initial, log in, recover and log out pages are those that remain equal to all users of this application and thanks to that we will not differentiate them, belonging to general design.

4.1.1 General Design

The first one is the initial page (figure 1), which does not exist yet. This is one of the proposed changes and the objective is to make the application more attractive to its users. We can see a sign in button and some introductory text.

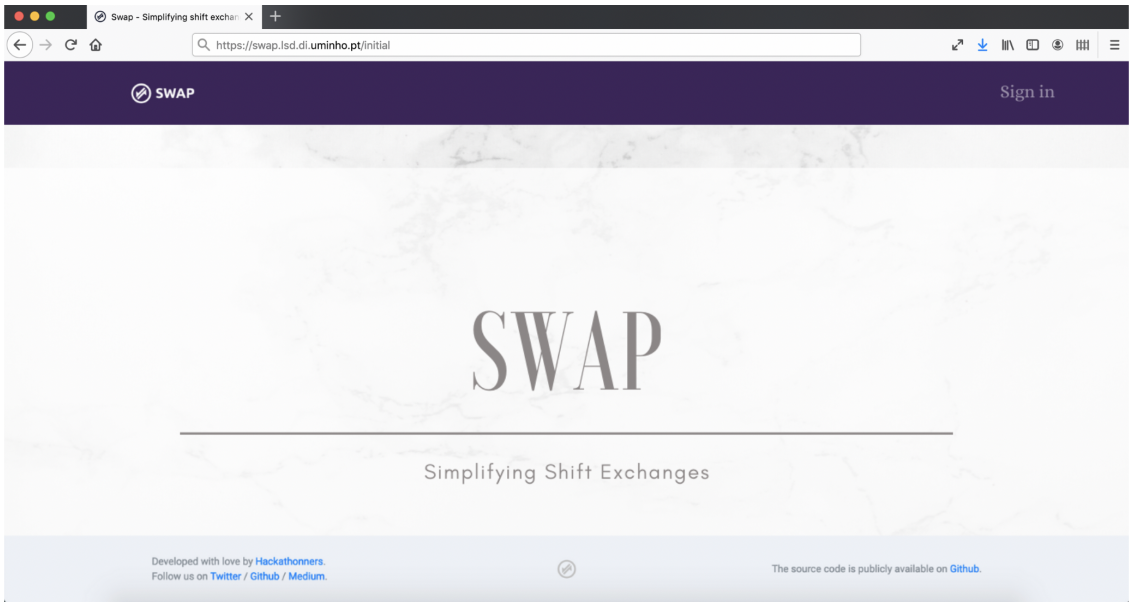

Figure 1: *SWAP* initial page.

Secondly, we have the recover page (figure 2), where the user may want to insert his email to which will be sent a reset link. When the email is inserted, he can press the reset button to confirm his intention.

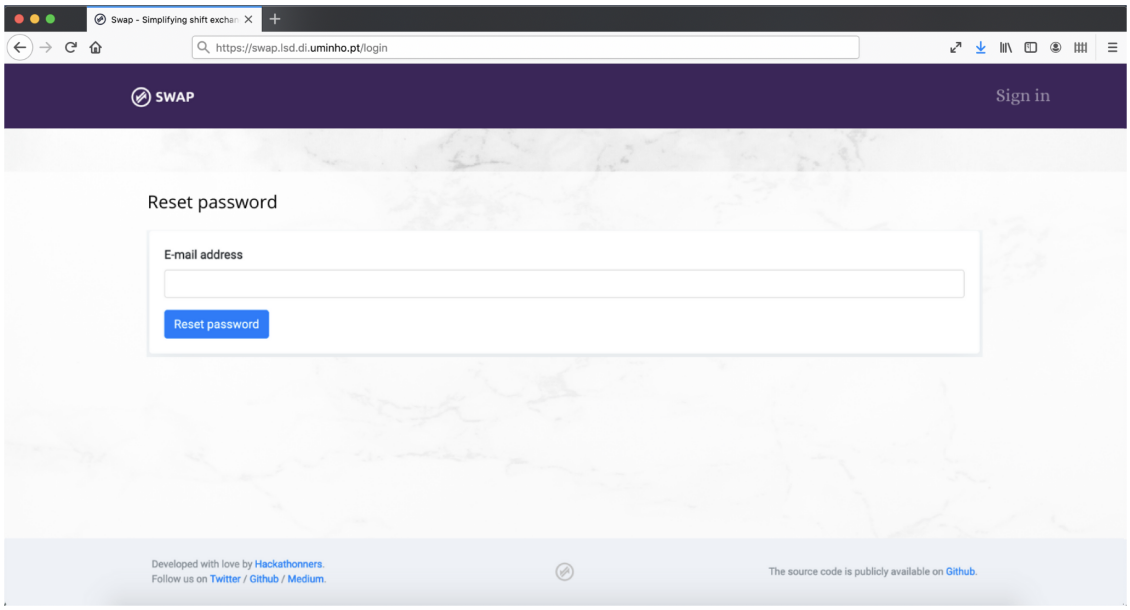

Figure 2: *SWAP* recovery page.

The log in page (figure 3) already exists. We have multiple options here which are: introduce an email, introduce a password, click in log in button or click in forgot button that would redirect the application to the page described before.

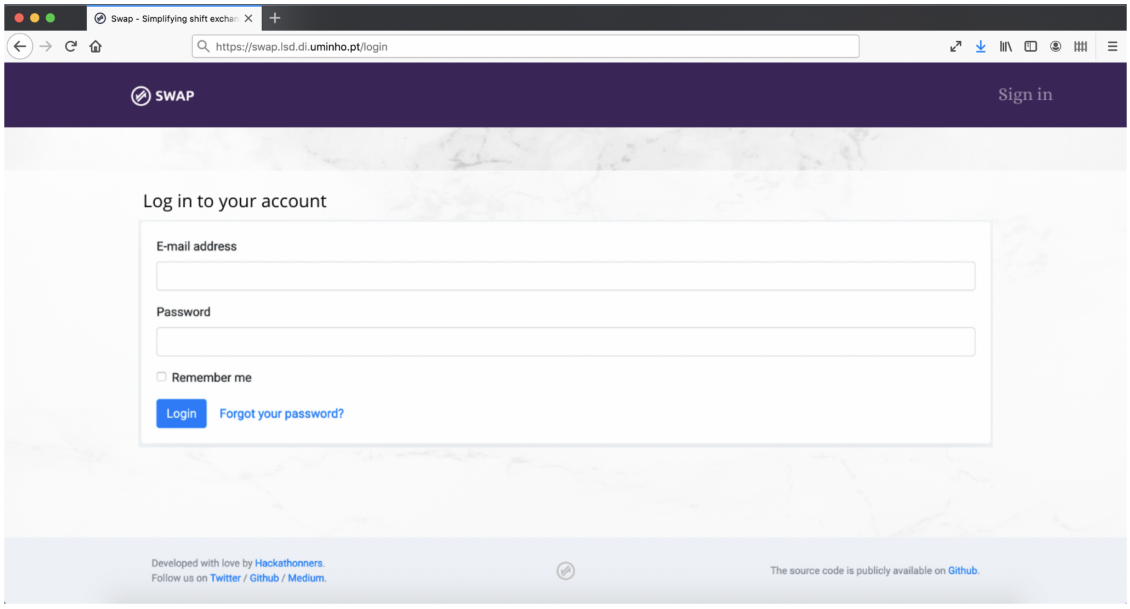

Figure 3: *SWAP* log in page.

If an user decides to log out, the next figure (4) shows what he would see: some text informing him of that action. This page must be followed by the initial page, something that is explicit in the requirements.

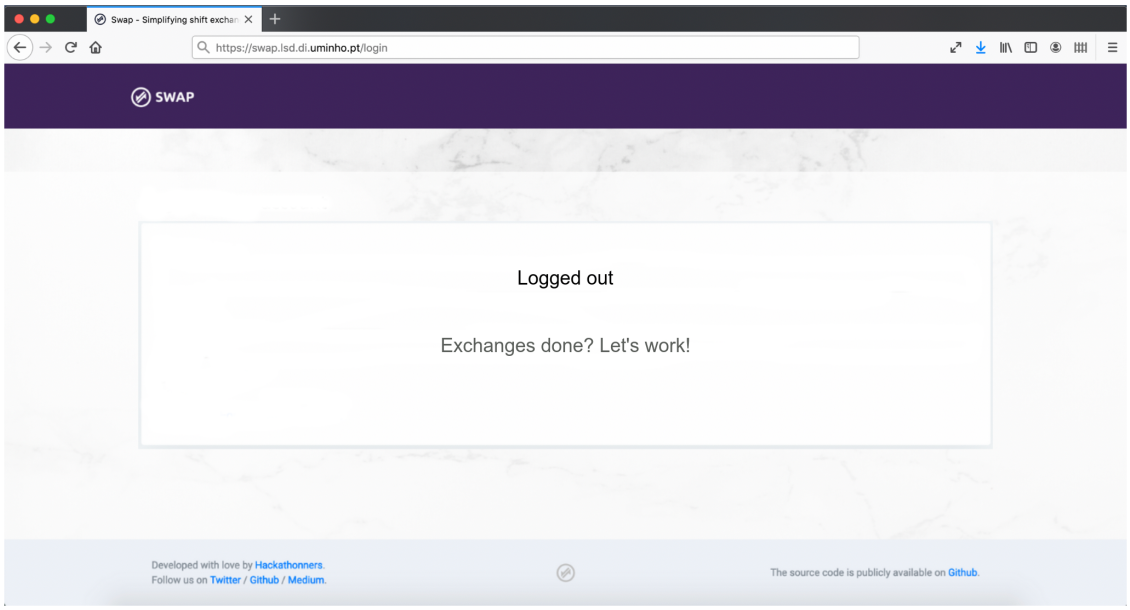

Figure 4: *SWAP* log out page.

4.1.2 Regular User Design

These regular user mockups represent the application pages after a successful log in and before an eventual log out. So right after logging in the user must see the main page (figure 5).

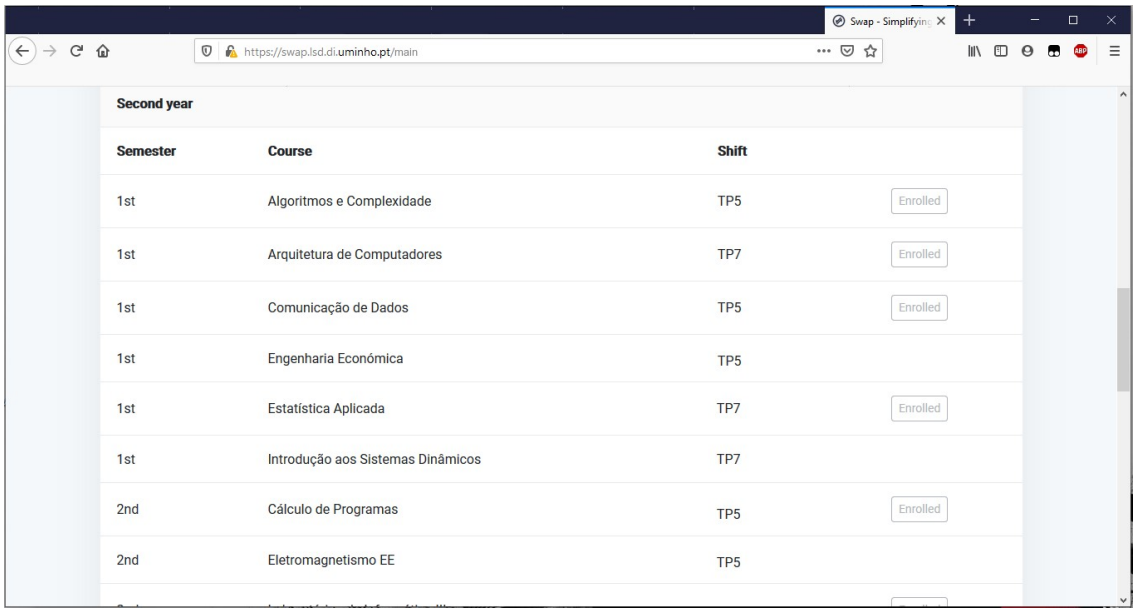

Figure 5: *SWAP* main page.

The courses page must be presented every time the user clicks in courses button. This is the page where the user is able to perform some exchanges. The next figure (6) refers to the incoming exchanges that other regular users may request. We see a drop-down menu to choose between

incoming exchanges and requested exchanges. If the user chooses incoming exchanges, then he will have another drop-down menu with the available incoming exchanges and a button to accept any shift from that list.

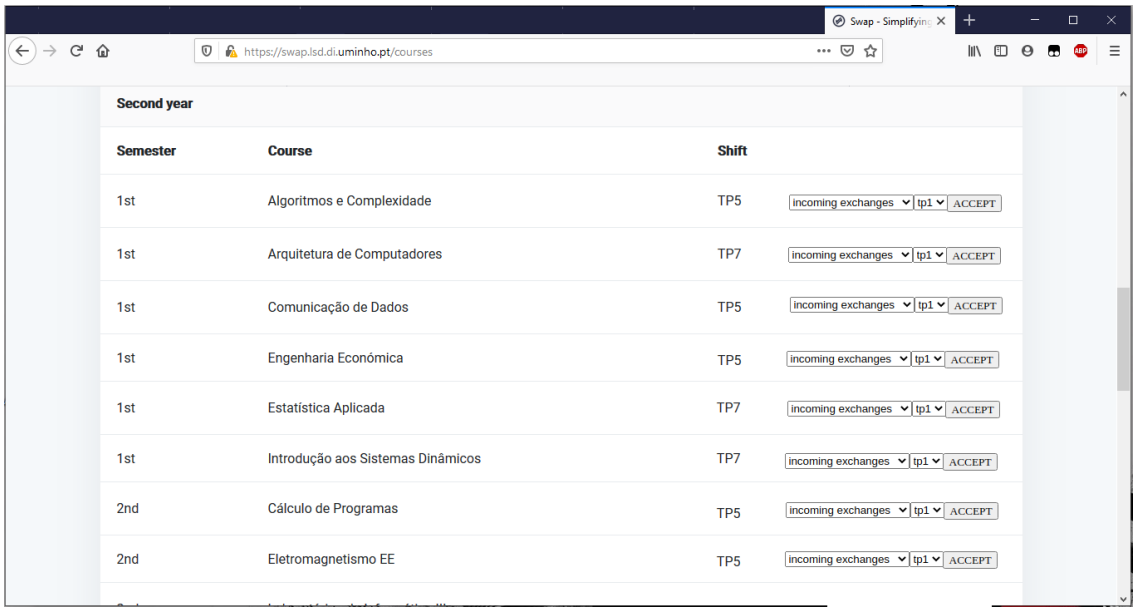

Figure 6: Accepting incoming exchanges in *SWAP* courses page.

Then, we have the figure 7 that refers to requested exchanges, those requested by the user. If he choose this option, it will appear a list with the shifts he requested before and the last entry of this list will be the option to do another request. If he chooses this last option, then another drop-down menu with all of the shifts he can request will appear. After choosing one he can send the request by clicking in the *request* button.

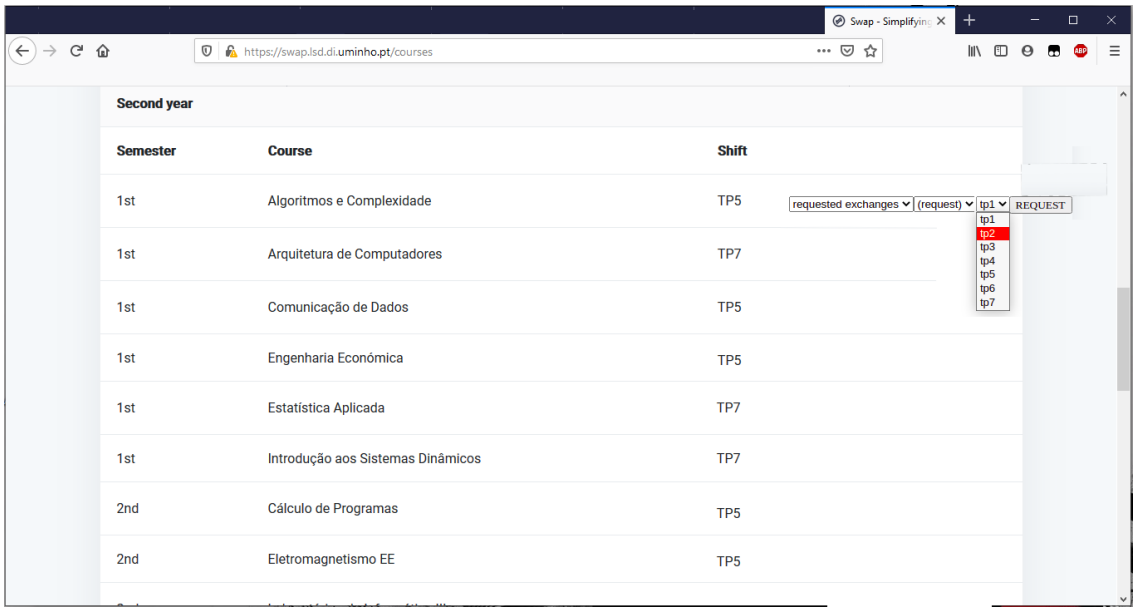

Figure 7: Requesting exchanges in *SWAP* courses page.

4.1.3 Administrator Design

After the administrator logs in, he sees the main page of the application.

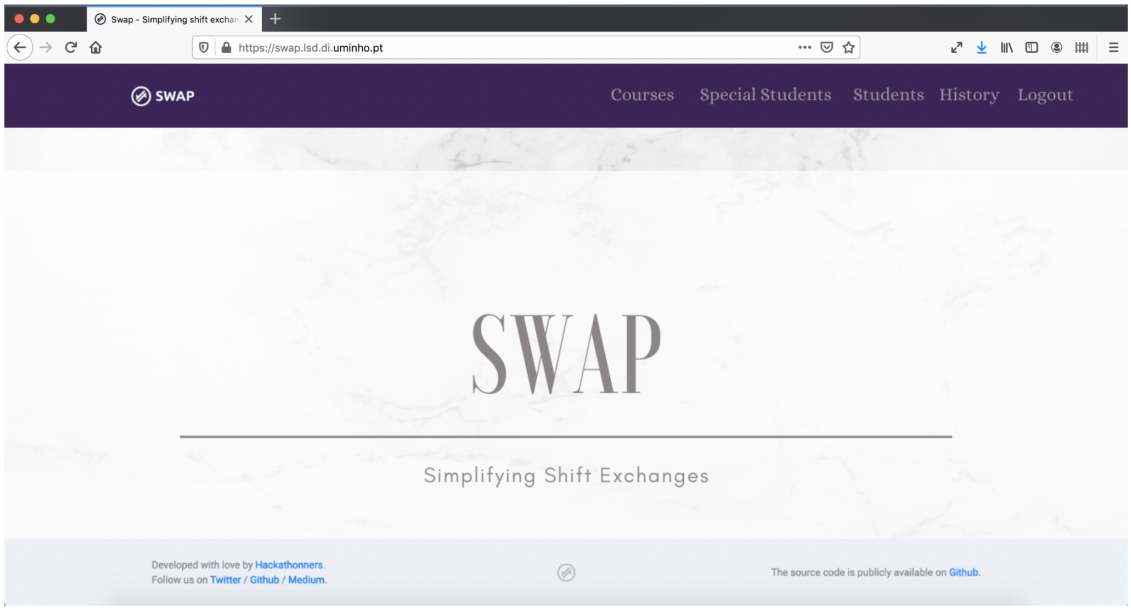

Figure 8: *SWAP* Main Page

If the administrator clicks on the courses button, he sees the current courses available and he can add shifts to the courses.

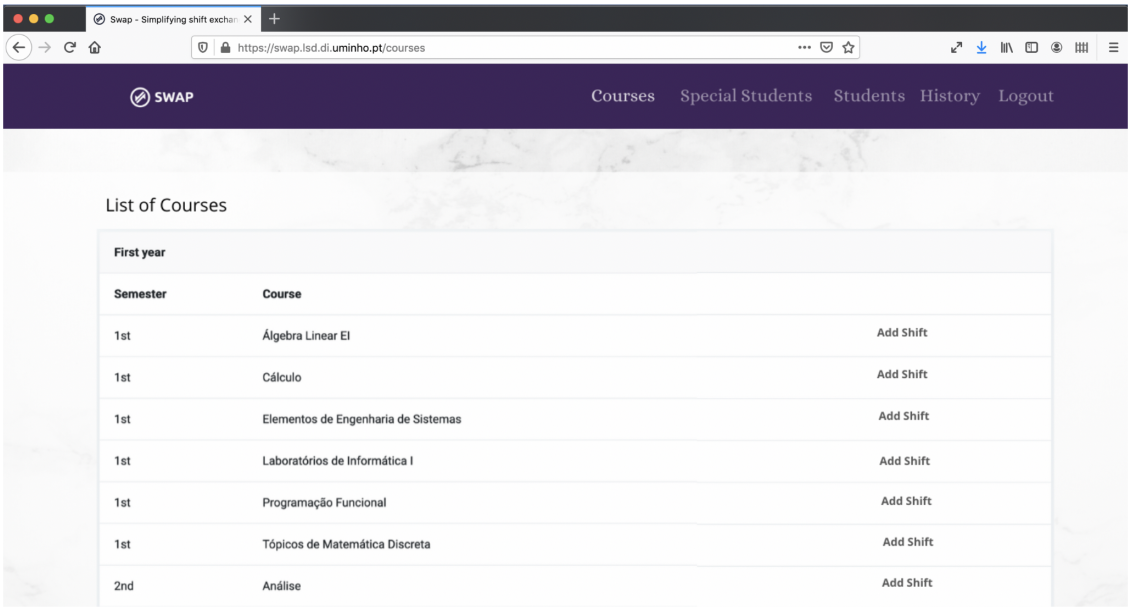

Figure 9: *SWAP* Courses Page

If the list of students with special conditions is uploaded to the server, the administrator can

activate the shifts of these students.

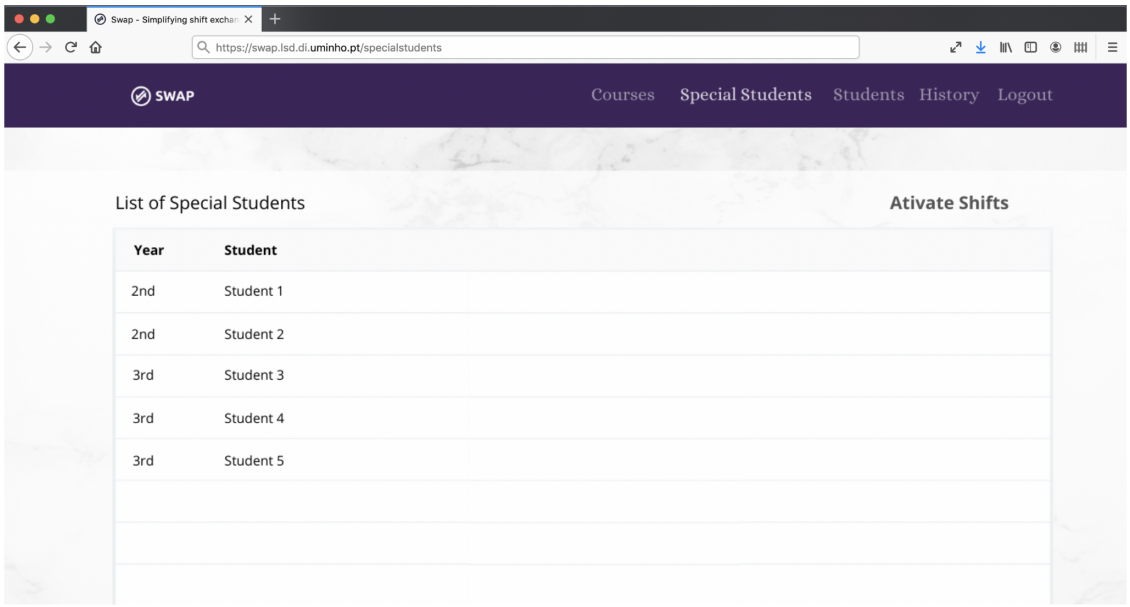

Figure 10: *SWAP* Students Page with Special Condition - if the list is uploaded

If the list of students with special conditions is not uploaded to the server, the administrator can submit this list.

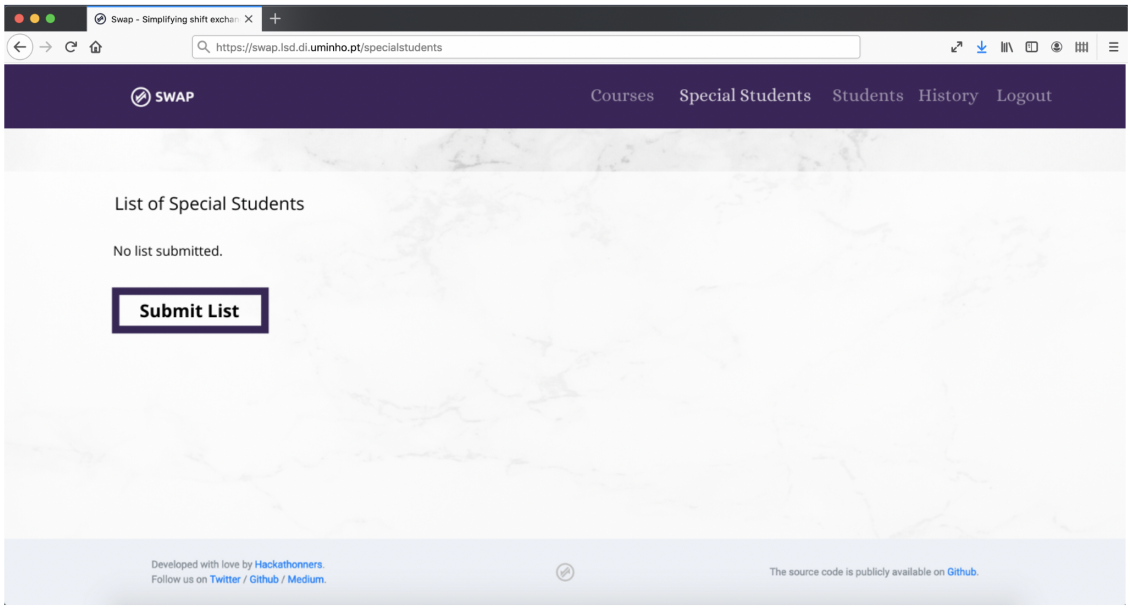

Figure 11: *SWAP* Students Page with Special Condition - if the list is not uploaded

The administrator can activate the shifts on this page and by choosing a student he can see his profile.

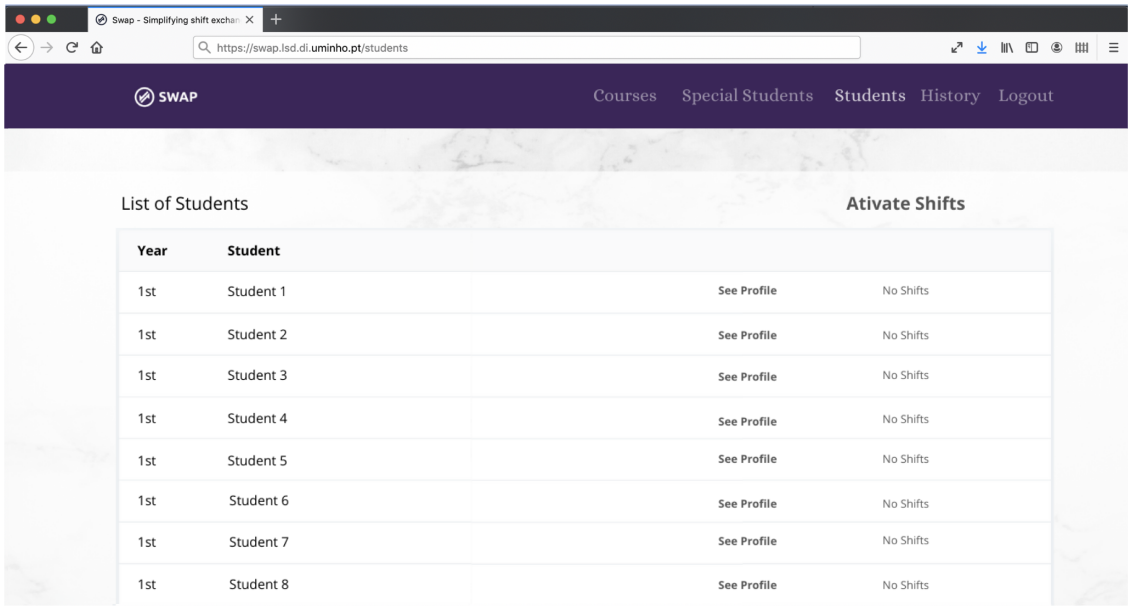

Figure 12: *SWAP* Students Page before the administrator activates the shifts

The administrator can change the shifts of the students and by choosing one of them he can see his profile.

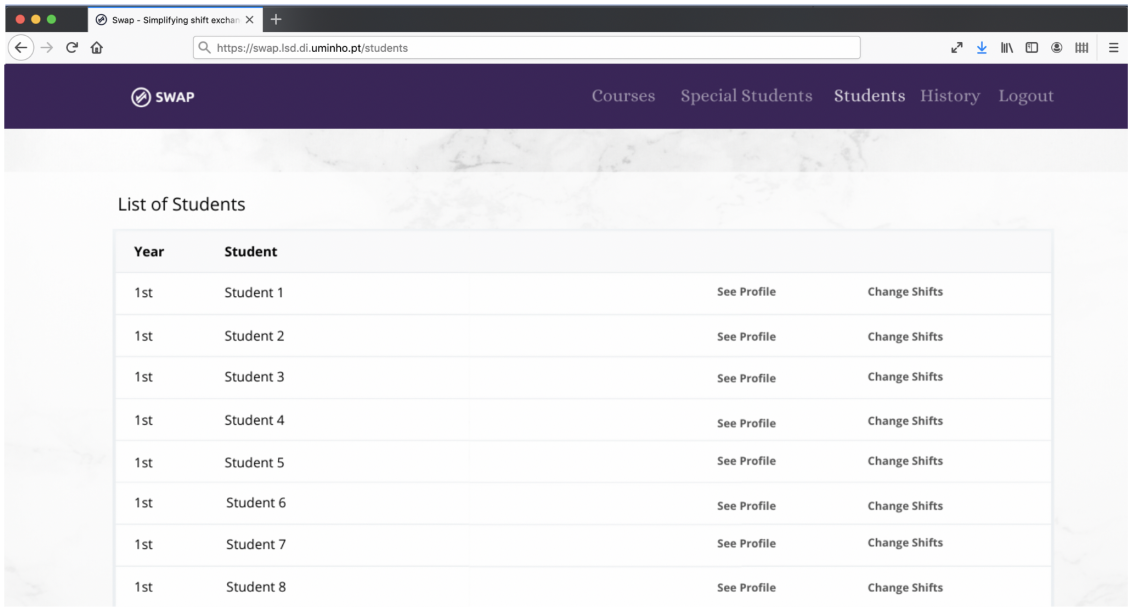

Figure 13: *SWAP* Students Page after the administrator ativates the shifts

After the choice of the administrator on seeing the profile of one student, he is able to see all the information about him.

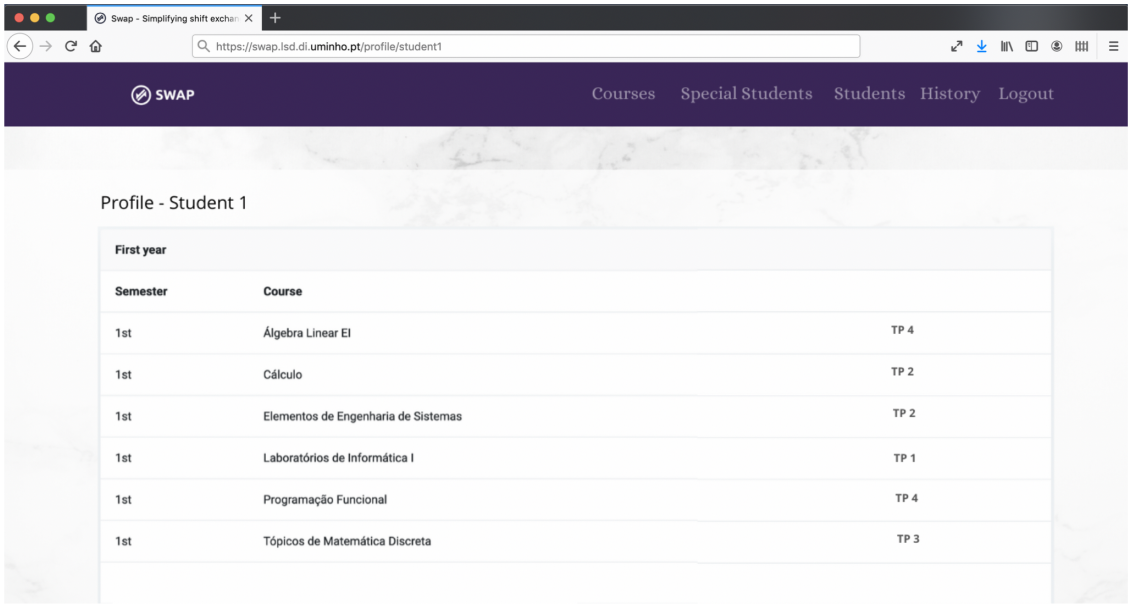

Figure 14: *SWAP* log out page (current design).

As we refer previously, the students can change the shifts so the administrator can see the changes they made.

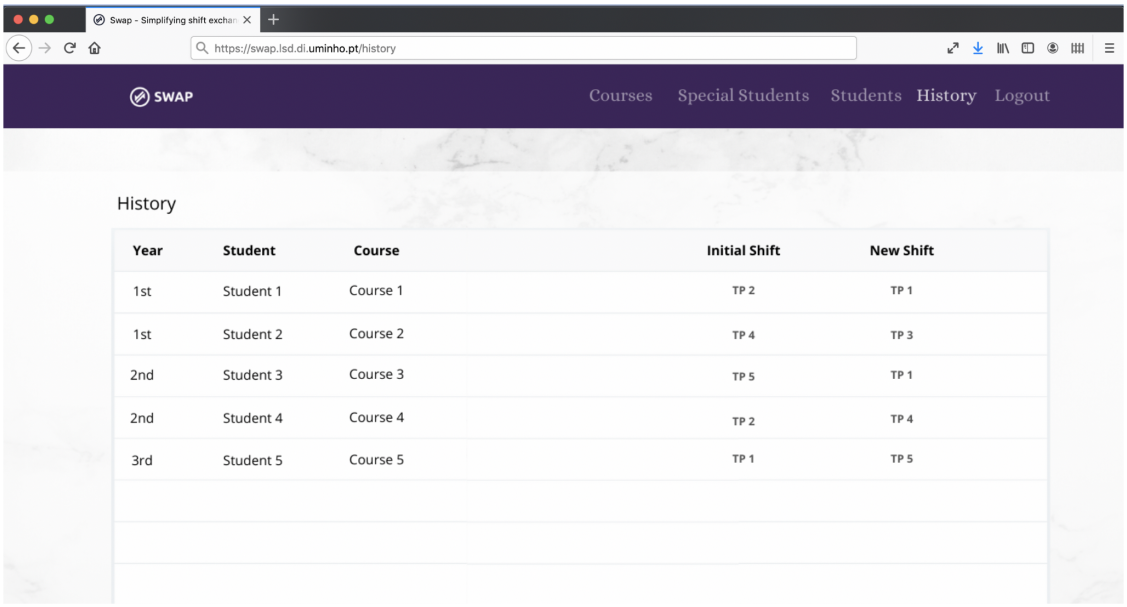

Figure 15: *SWAP* Student Profile Page

4.2 ConcurTaskTrees Environment

Like we said before, the design is a really important aspect that must be part of our main concerns. It is not just about appearance but also about user interaction, since he decides what

he can or can not do when he is looking at the page. This interaction is precisely what *ConcurTaskTrees Environment* deals with and bellow we have a very brief *SWAP* modeling in this tool.

4.2.1 Regular User Design

From the point of view of the regular user we can see that there are different types of actions, some are complex and others are simple. So, we have the *SWAP* application which is the main abstraction that can be divided in two other actions. The user can decide to recover the password, where he has to insert an email address to which the system will send a link to recover it. An alternative to that would be the login where he inserts the username and the password (not by this specific order) and lets the system process the credentials. After that, the user logs in and is sent to the main page. Here he can alternate between this page and courses page. As we can see, "Courses info" element has an asterisk that tells us he can execute this step as many times as he wants. When inside of courses page, he can choose a course, choose a shift and submit an exchange request that will be processed by the system or can simply verify exchange requests he may have. As soon as the user logs out, he can not undo this operation and needs to login again to do something else.

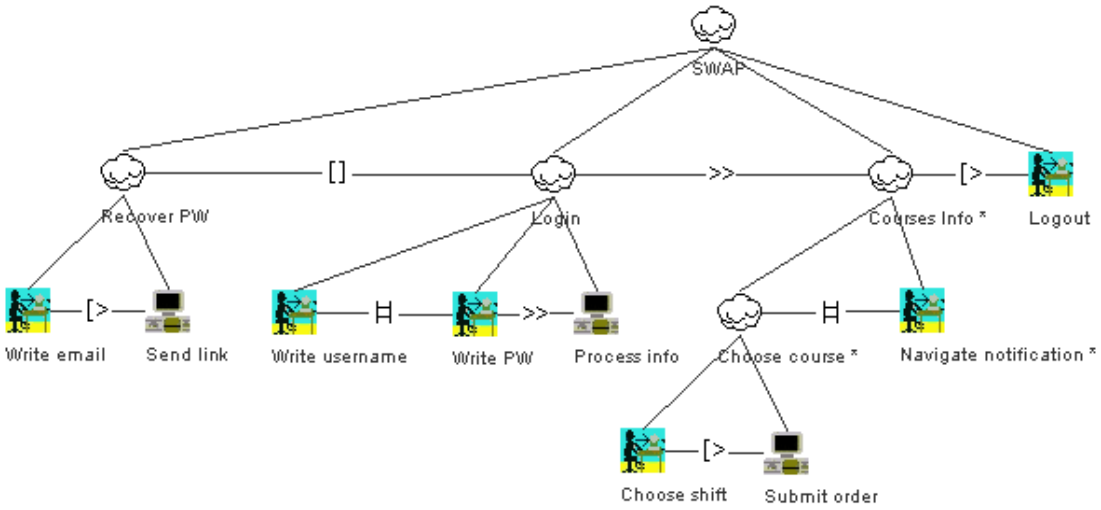

Figure 16: *ConcurTaskTree* of Regular User

4.2.2 Administrator Design

After knowing the requirements that the administrator has and to achieve these in the future, it was necessary to perform task modeling. For this, we used the *ConcurTaskTrees Environment* which allowed us to have a more detailed view of what would have to be modeled and the behavior of the application.

When the administrator opens the application, he can recover the password or log in. After this, the administrator can use the application. He can see the courses list and add shifts if he wants. He can also see the students with special conditions and submit the list if this list is not uploaded to the server. After this, the administrator can activate the shifts and see the students list. On this page, he can change the shifts and see the profile of the student he chooses. If students make changes in their shifts, the administrator can see the changes that they made. When the administrator logs out, the simulation stops.

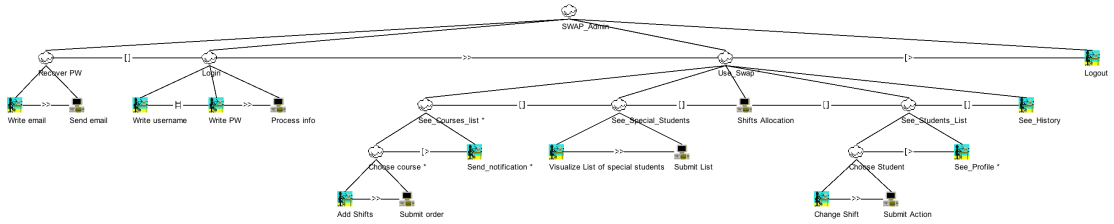

Figure 17: ConcurTaskTree of Administrator

5 Analysis

The main goal of modeling a system is to **ensure that every aspect has been studied and registered before**, in order to make the development easier. By removing the need to think deeply about the problem makes the **development process faster and less chaotic** since the programmer only has to apply everything he knows about the language he is using. He applies this knowledge to build an application that is completely structured and analysed which is much better than thinking and building something at the same time. It is very common to see discussions and criticisms to something that is being built but at this point, if anything needs to be changed means someone's time has been wasted and **the probability of inserting errors into the system increases**.

The intention of this section is to corroborate everything that has been said about the modeling of applications using formal tools, namely web applications. Below, we have a comparison of what was modeled with what was built (subsection 5.1) and the verification of some invariants (subsection 5.2). To each case we will present visual traces of the states given by the *Electrum* tool, a very powerful feature that can even be customized.

5.1 *SWAP* model behaviour

5.1.1 Regular User Interaction

As described in the requirements, the application must have an initial page other than log in, one that briefly introduces and informs the user what can be done with it. So, as we can see in the following screenshot we have the initial page with a sign in button, just as it is in the mockup shown in figure 1.

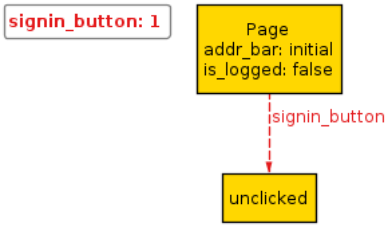

Figure 18: Sign in button unclicked.

The next state, expectedly, remains equal to the previous, since *Electrum* prefers to choose a state where nothing changes (*skip* predicate). Forking the first state will force *Electrum* to apply some other predicate different than *skip* and in this case the only one that fulfils the actual preconditions is *signing_in*. The sign in button is setted to *clicked*.

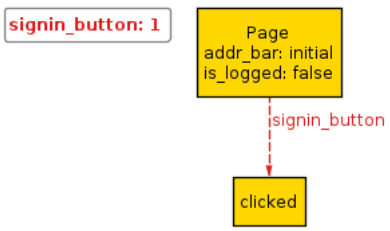

Figure 19: Sign in button clicked.

As we click in the right arrow, the new previous state becomes the old next state and the new next state becomes what is defined in the predicate that *Electrum* decides to apply. Again, it decides to call *skip* and forking it leads us to another trace (another predicate is called), where

in the next state the address bar is in login and the username and password fields are *empty* (a transition from *initial* to *login* happened) - figure 3.

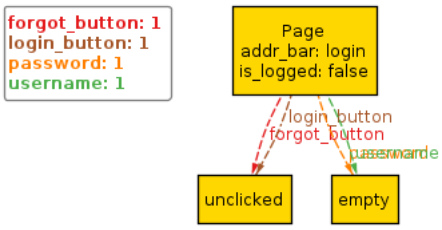

Figure 20: Unfilled login page.

Proceeding in the trace and forking it makes us face three situations: the user inserts *valid* credentials and logs in (assuming that he has filled them in), or inserts *invalid* credentials (at least one) and the the log in page is reseted (*unclicked* buttons and *empty* fields) or clicks forgot button to recover his account.

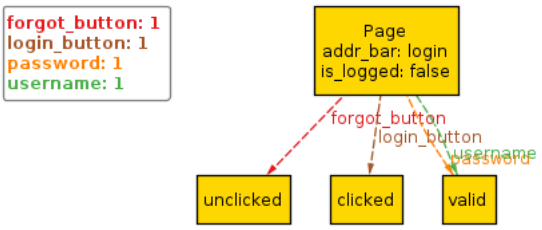

Figure 21: Valid credentials and log in button clicked.

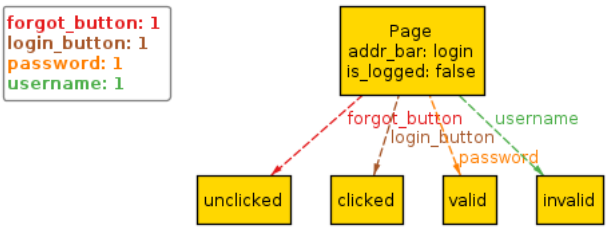

Figure 22: Invalid credentials and log in button clicked.

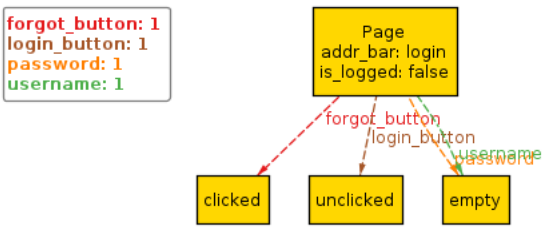

Figure 23: Forgot button clicked.

Following the trace of figure 23 it is expected that the next state shows the address bar setted to *recover*, which means the user is in account recovery page, as shown in the following figure.

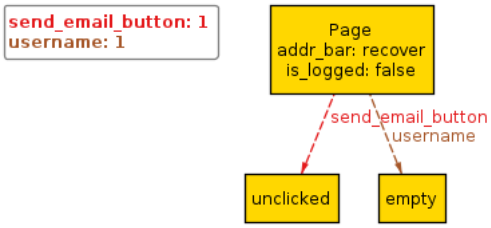

Figure 24: Recover page.

Now, the user can either insert a *valid* or an *invalid* username to which the reset link will be sent. In the first situation he is redirected to the initial page and in the second one the recover page is reseted so he can reintroduce the username (remember that the the username is his institutional email address).

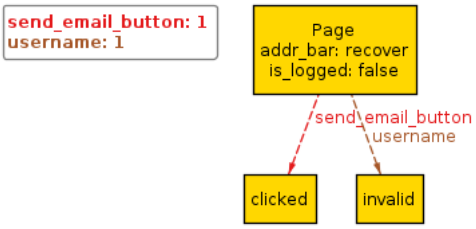

Figure 25: Forgot button clicked and invalid credentials.

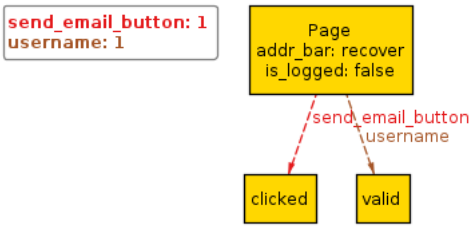

Figure 26: Forgot button clicked and valid credentials.

At this point we can say we reached the end of a trace because it starts to loop, since it returned to the initial state. Recalling the trace of figure 21, when the user tries to log in with *valid* credentials, proceeding and forking it, changes the page address bar to *main*, some new elements appear and others disappear. Among them we have, for example, courses and log out buttons and a list of courses, the ones in which he is registered. It is important to state that *Course* instances are always present in every trace, but since in the real application they must only appear in *main* and *courses* pages, we tried to reproduce this in the traces view, presenting them only when they are related to the page (remember that there are no relations between *Page* and *Course* when the called predicate contains a clause *no p.courses_list*). This state refers to figure 5.

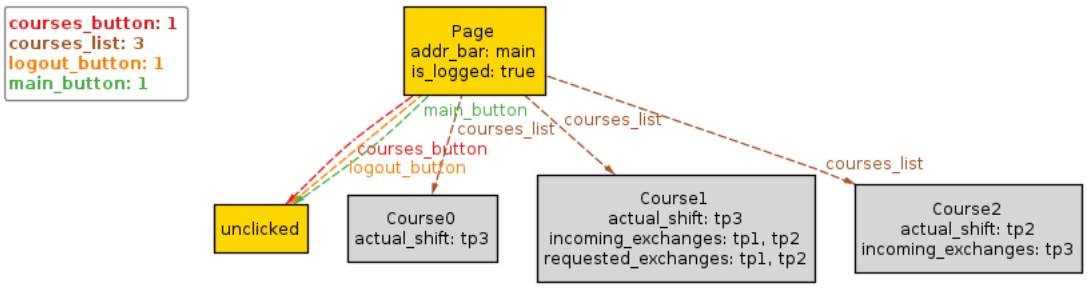

Figure 27: Main page with all buttons unclicked.

So far, we have seen a user that can access *SWAP* web application, click the sign in button and decide to log in or to recover his account if he needs to. Now, once in the main page, the user has two options: proceed to courses or log out page and the decision will depend on the button that is clicked. Below we have the traces that represent these alternatives.

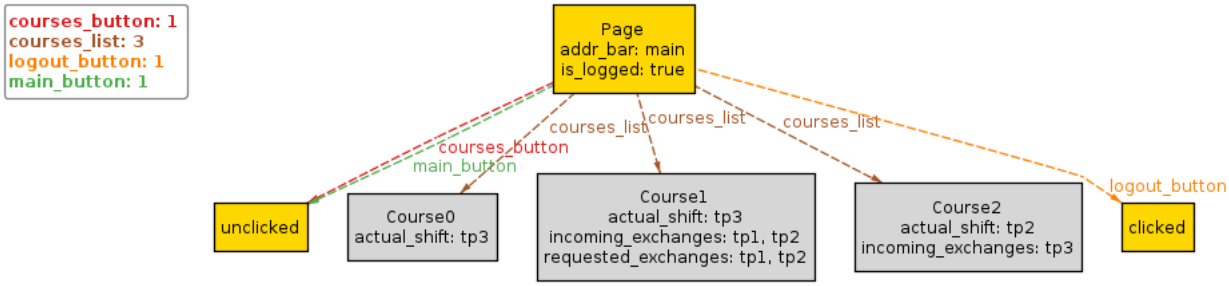

Figure 28: Main page with log out button clicked.

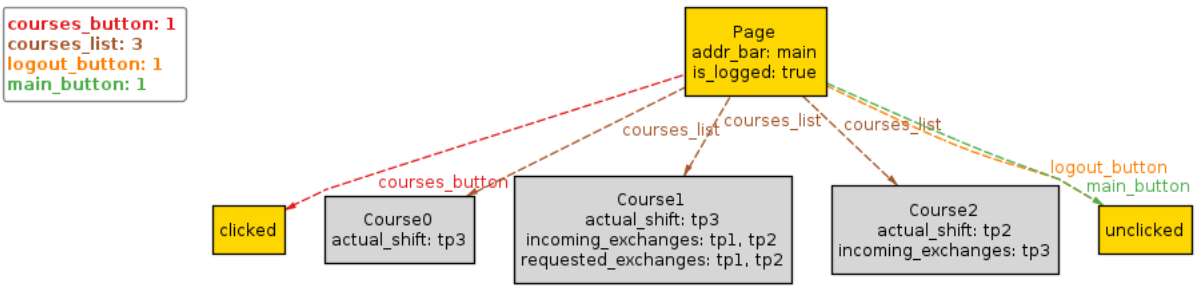

Figure 29: Main page with courses button clicked.

Let us imagine that the user chooses to log out. In this case the logout page appears and the variable *is_logged* is setted to *false*. There are barely no elements in such page and that is the reason why the following page instance only has that relation available (which in the trace theme was defined has an attribute of *Page* instances). Once the application is in log out page, it will return to the initial page right after that, finishing one more trace.

Figure 30: Logout page.

Recalling the trace of figure 29, the courses button was clicked and that means a transition to courses page must occur in the state after (figure 31), where new buttons such as accept appear.

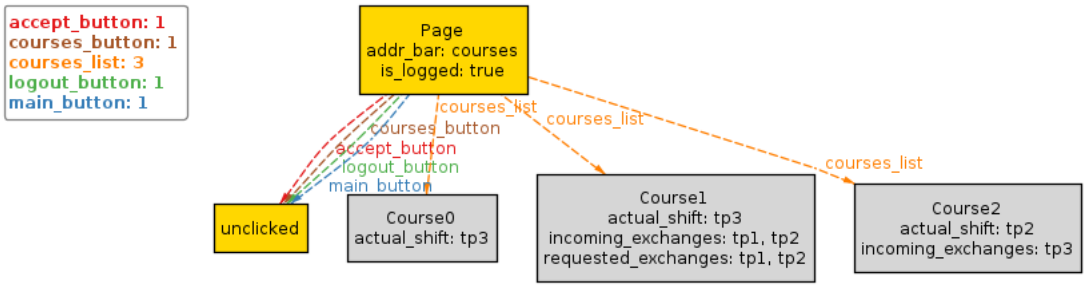

Figure 31: Courses page with unclicked buttons.

Right next to that, the user will probably click a button and the available options will be accept an exchange, return to main page or log out. We have already seen similar traces of the last two, so let us only talk about the first one. The respective mockup that refers to this situation is showed in figure 6.

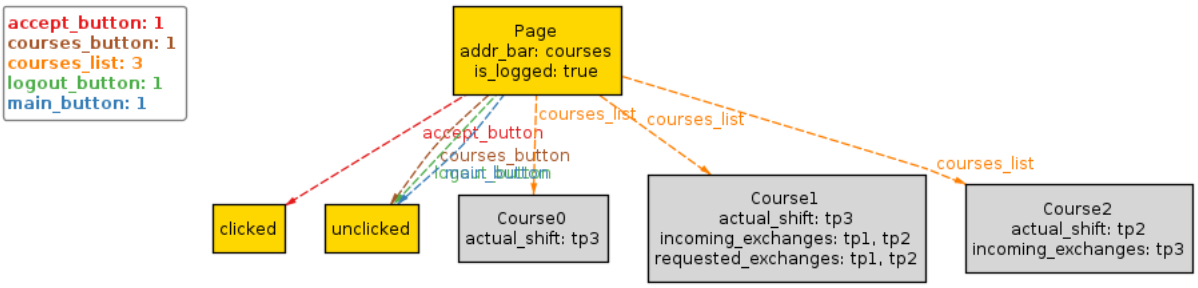

Figure 32: Courses page with accept button clicked.

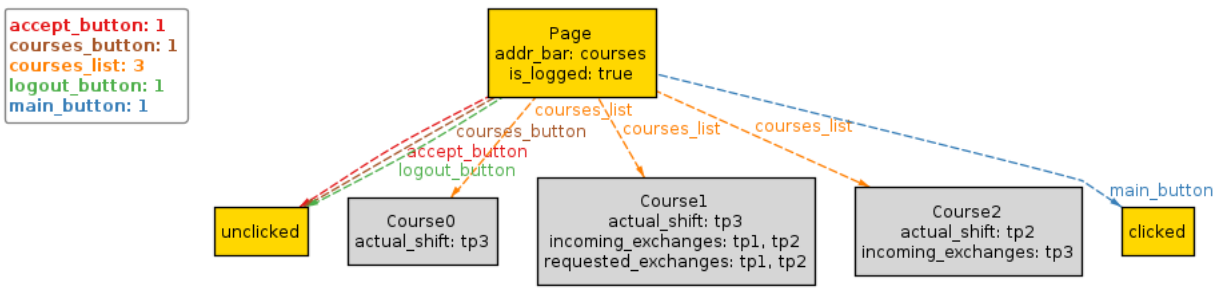

Figure 33: Courses page with main button clicked.

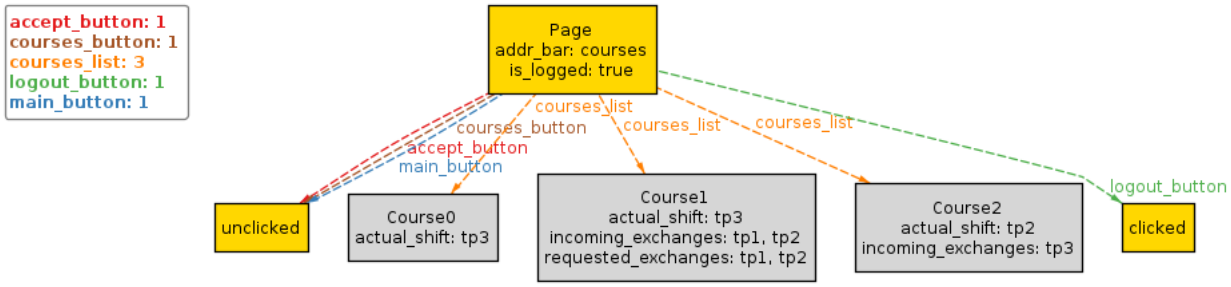

Figure 34: Courses page with log out button clicked.

As the user clicks in the accept button, some adjustments must occur according to the requirements written above. So, a course that has some incoming exchanges will lose one of these which will replace the current shift and will also clean its incoming and requested exchanges.

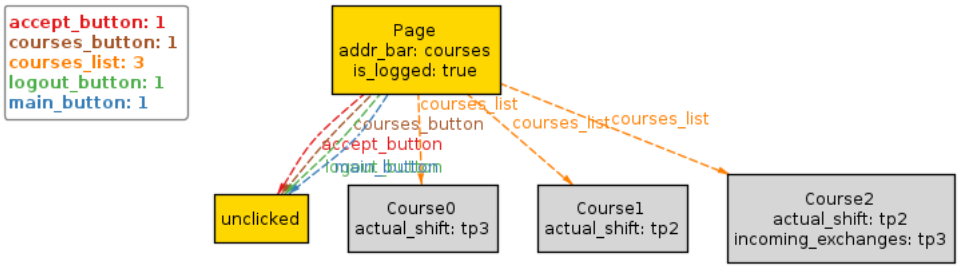

Figure 35: Accepting and performing exchanges .

From now one, the next steps that can be done will fall in one of the situations that were descibed above when the courses page had all of his buttons *unclicked*. We can now consider that possible traces are all reviewed.

5.1.2 Administrator Interaction

The model starts with the user logged out and with the sign in button for clicking as expected and their home page.

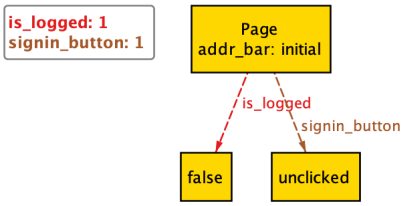

Figure 36: Sigin Button Unclicked

Presses the sign in button.

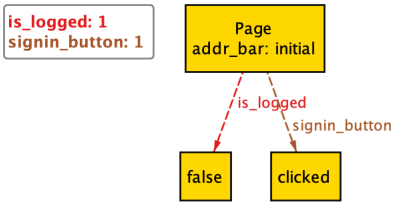

Figure 37: Sigin Button Clicked

After clicking on sign in, the administrator goes to the log in Page. Here the text boxes of the username and password are *empty* and the buttons *unclicked*.

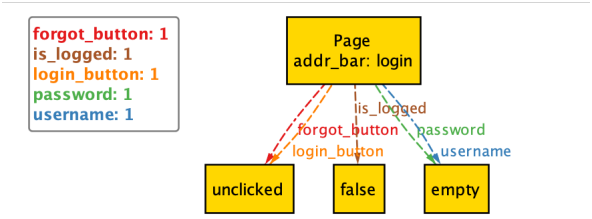

Figure 38: Login Page with empty text boxes

Fill in the fields with *valid* values and press the log button to start the session.

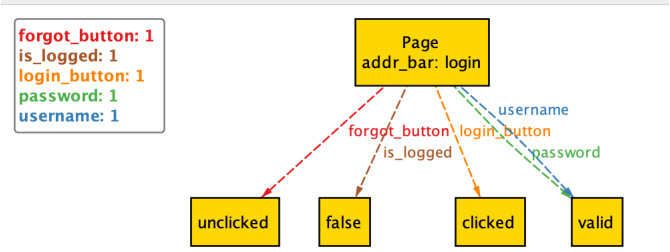

Figure 39: Login Page with valid username and password and the login button clicked

If both fields or only one are *invalid*, log in goes wrong and he needs to try log in again.

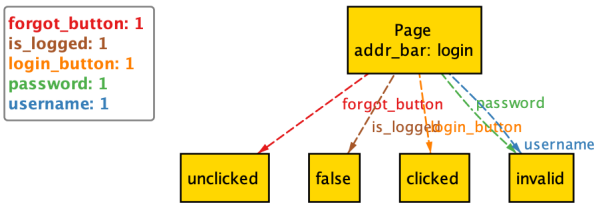

Figure 40: Login Page with invalid username and password and the login button clicked

The administrator can recover his password by clicking on the forgot button.

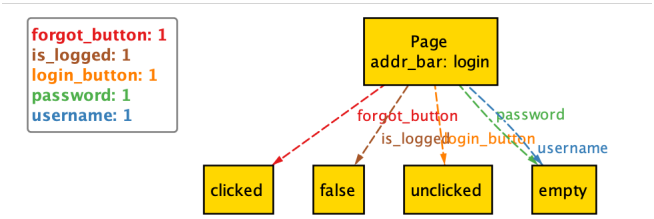

Figure 41: Login Page with the forgot button clicked

After clicking on the forgot button, the administrator goes to the Recover Page and needs to field the username text box.

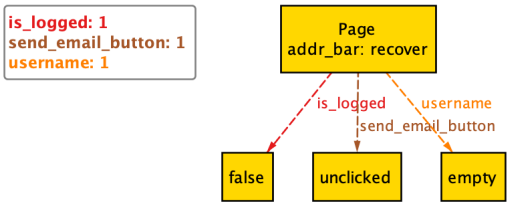

Figure 42: Recover Page with empty field

Filling the username text box with a valid email, he clicks on the send_email_button and it is possible to recover his password.

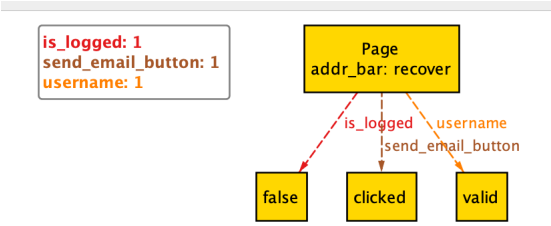

Figure 43: Recover Page with valid username and send email activated

After the administrator logs in, he goes to the main page and now is able to execute his properties ate the application.

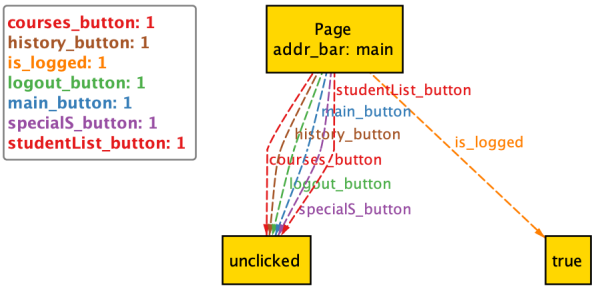

Figure 44: Main Page with navbar unclicked

After being in the main, the administrator can, for example, click on the courses button.

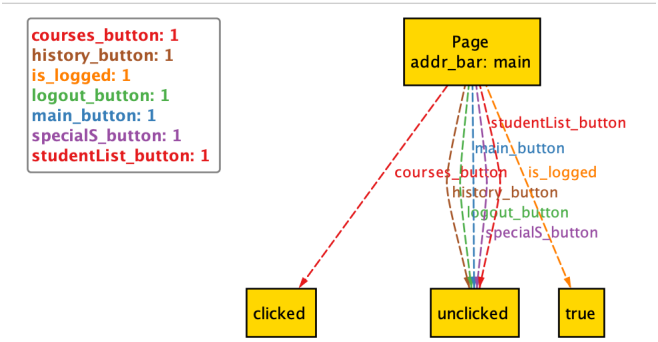

Figure 45: Main Page with Courses Button clicked

After this button is clicked, it is possible to see the list of available courses.

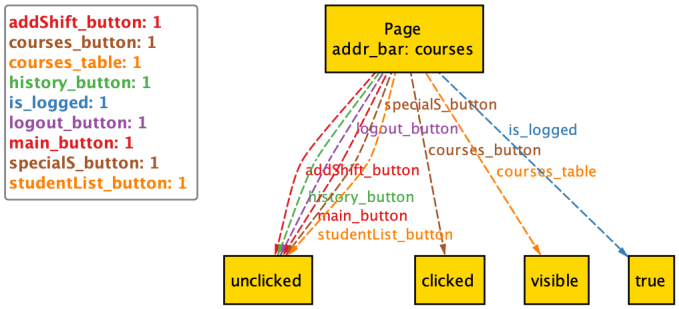

Figure 46: Courses Page with visible information

On the Courses Page, the administrator can add shifts to an available course, by pressing the corresponding button.

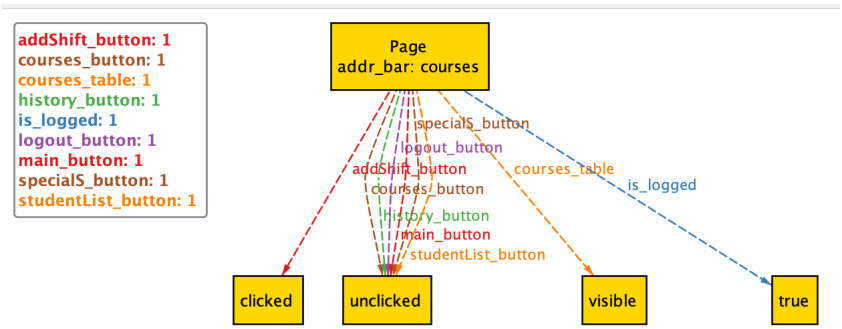

Figure 47: Add Shift on Courses Page

Starting from the main, or from another page, the administrator can, for example, go to the students page.

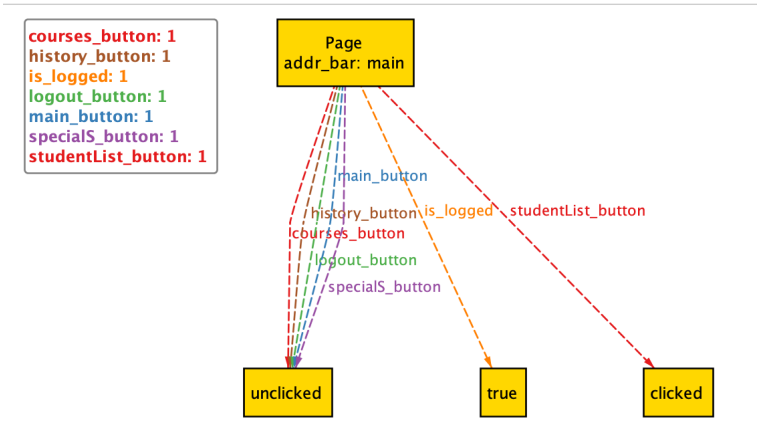

Figure 48: Main Page with Students List Button clicked

On this page, the administrator has two possible views. When the list of students appears, the shifts may not yet be activated, so it is possible to activate them. As soon as these have been activated previously or even after the activation action, the administrator can change the shift of a certain student.

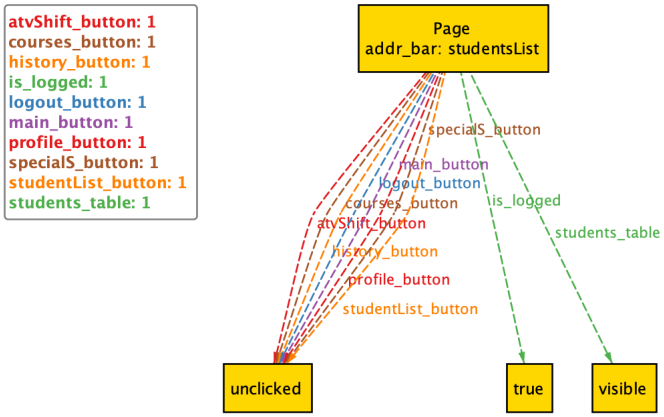

Figure 49: Students List Page with visible students information and with atvShift Button

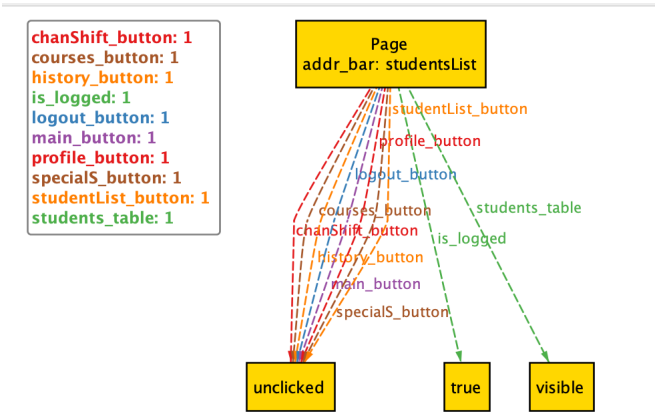

Figure 50: Students List Page with visible students information and with chanShift Button

The administrator is able to activate the shifts.

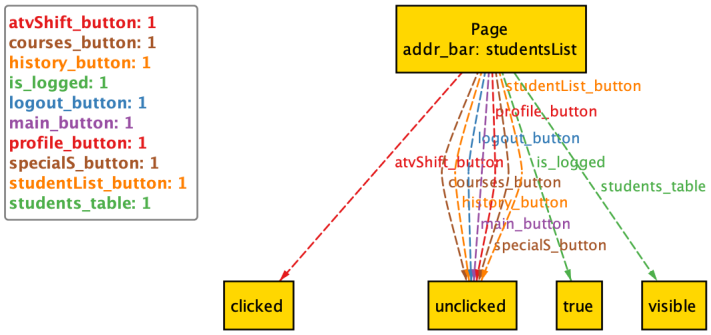

Figure 51: Students List Page activate shifts

In the other view or after activation, the administrator can then change the shifts.

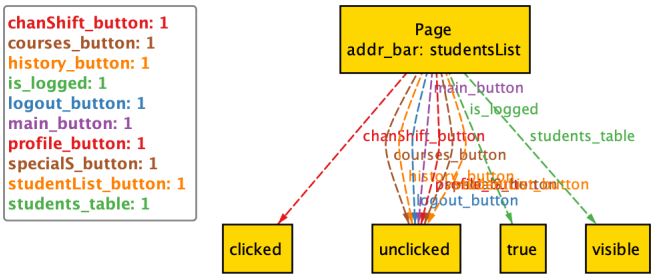

Figure 52: Students List Page change shifts

In this page Students List, the administrator can also view the profile of a student of his choice, clicked on the profile.

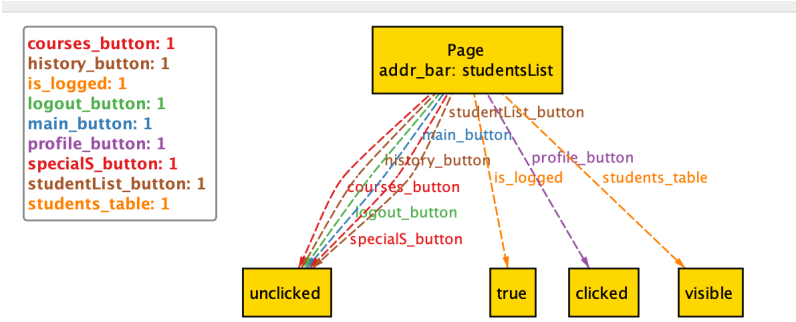

Figure 53: Students List Page with Profile Button clicked

Now it is possible to view all the information of the student that the administrator picked.

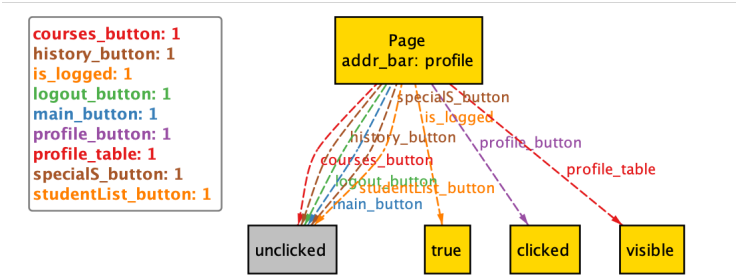

Figure 54: Profile Page with visible information

The administrator can also view the exchanges between students, by viewing the history, selecting this option in the nav-bar, from the main or another page.

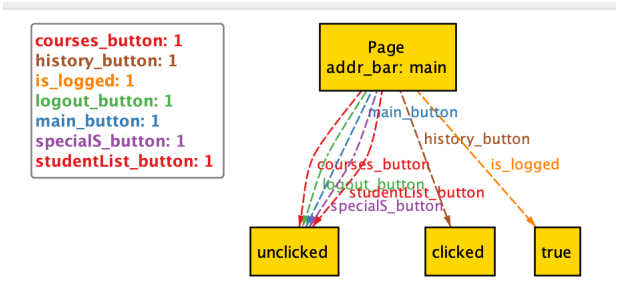

Figure 55: Main Page with History Button clicked

After pressing this button, the administrator is able to see:

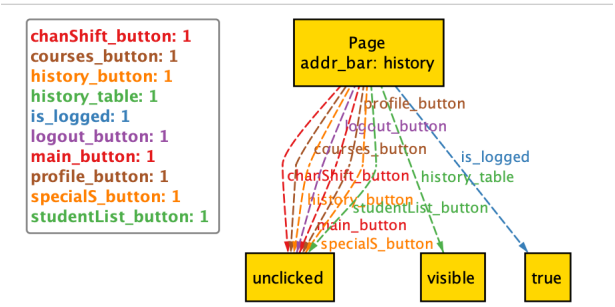

Figure 56: History Page with visible information

It is possible to see students with special conditions by clicking on the respective button.

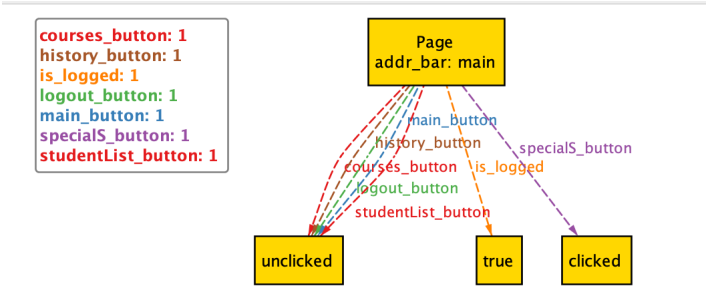

Figure 57: Main Page with Special Students Button clicked

The administrator can have two views because the list may or may not be submitted. Not being submitted, it is possible to see:

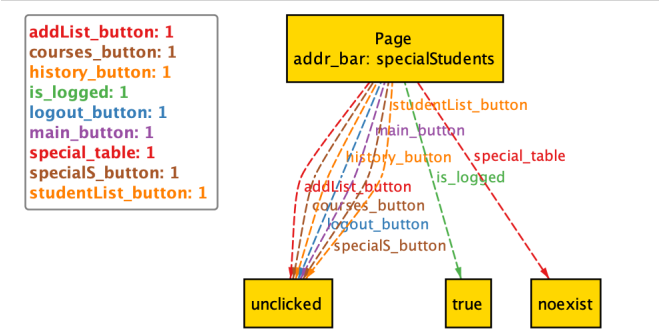

Figure 58: Special Students Page without list

In this way it is possible to add it:

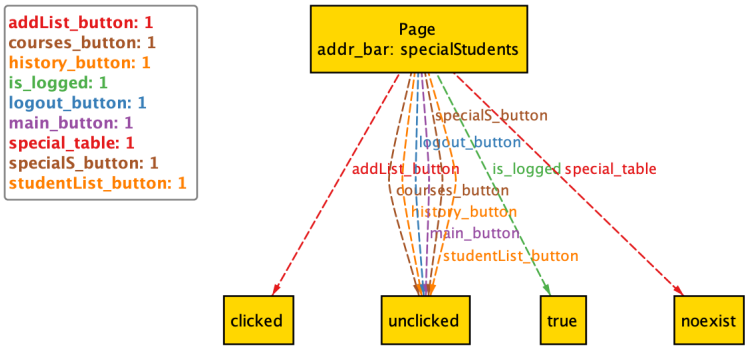

Figure 59: Special Students Page active add list

When the list is added or was already on the server, the Special Students Page is as follows:

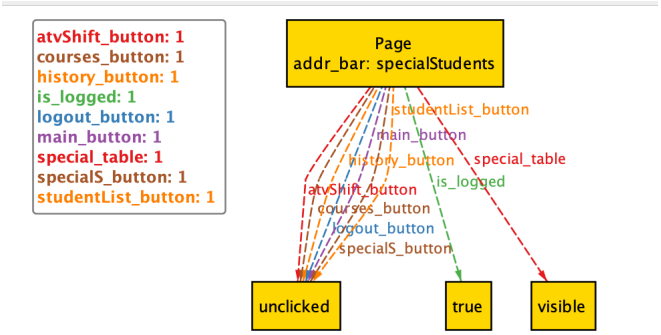

Figure 60: Special Students Page with list

The administrator can activate these students shift as well.

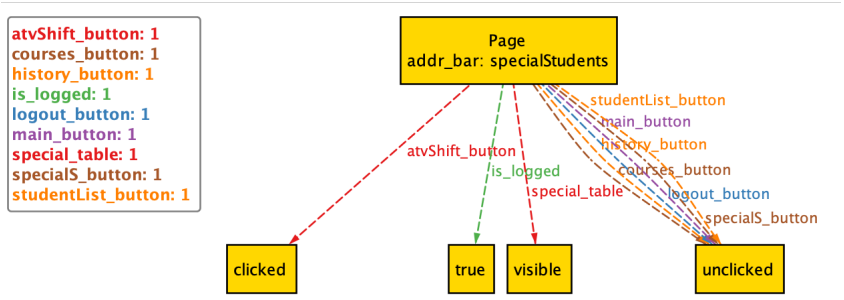

Figure 61: Special Students Page active shifts

The administrator can also, from any page, click on the main button, taking his to that page.

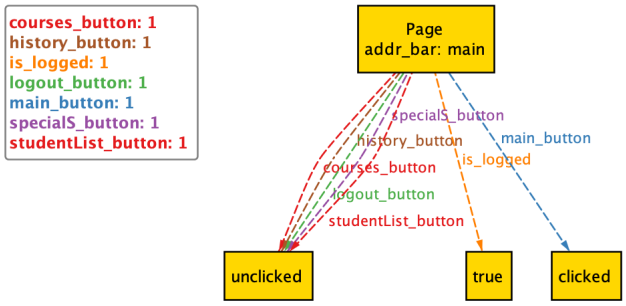

Figure 62: Click on Main Button

Finally, when the administrator wants, he can log out and return to the initial page.

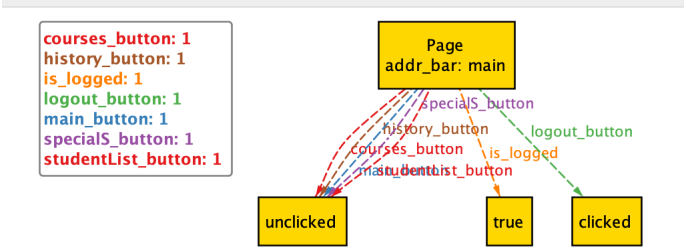

Figure 63: Click on Logout Button

5.2 *SWAP* model facts and assertions

The reason why these formal tools are considered so powerful lies on their **ability to check and assure some facts and properties that must always be true, under the dependence of math laws**. Furthermore, this type of verification can be done to all of the states of the model. The language used by *Electrum* establishes facts as invariants, a truth that remains as it is, true, no matter what. In another hand, it also establishes asserts but these are more flexible, since they allow us to proof that something happens or not.

5.2.1 Regular User Interaction

initialize (fact)

This is the fact that creates the initial state (*initialize*).

transitions (fact)

The fact that manages what predicates can be applied to the traces is called *transitions*.

exchanges_cant_have_actual (fact)

exchanges_cant_have_actual keeps the current shift out of the sets of incoming and requested exchanges for each *Course* instance.

not_stucked_in_logout (fact)

The fact that prevents the model from looping with *skip* predicate when *logout* page is reached. That would mean the model could get stuck in logout page.

logged_in_pages (assert)

logged_in_pages proofs that the user can not be logged in when the application is displaying some specific pages like *initial*, *login*, *recover* and *logout*.

logout_goes_to_initial (assert)

Checks that after *logout* page, the aplication returns to the *initial* page.

valid_login_goes_to_main (assert)

Something that is an absolute truth in the great majority of the applications is to log in with *valid* credentials. Every application with user session requires this assert.

one_and_oone_actual_shift (assert)

The *SWAP* application only allows one shift per *Course* instance. This assert checks that this is a truth.

cant_go_main_after_courses (assert)

Asserts also allow us to search for a very specific trace. Let us say that we want to find a trace where the user is in the *courses* page and changes to the *main* page. Writting that in *Electrum* and defining it has impossible will force this tool to find a counterexample that will certainly exist, since that is clearly false. This assert explores this aspect.

5.2.2 Administrator Interaction

These facts are important because they guarantee a certain type of behavior that the model must have. We defined some for the administrator but it would always be possible to define more and the more you define the better the behavior will be. Something that could be improved would be, for example, to define some more facts as they would make the code more readable and optimized. In this way we have:

initialize

This fact initializes the first state of the model (initial state).

transitions

This fact is very important because manages the model and decides which predicate will be applied to the traces.

logged_in_pages

This fact guarantee that after login, what are pages we can navigate.

login_transition_main

If we are logged in, this fact makes sure that we wrote valid credentials. After log in, the next page is the main page and we guarantee this property with this fact.

6 Conclusion and Future Work

In software engineering, there is a very long way to go when it comes to building safe and functional applications. A previous study and model of the product to be built is essential and is not just something that should be done; **it is something that must be done**. The only way to make it a reality is to change some minds by trying to turn it into a trend that lasts because it is guaranteed that **companies will find their code healthier** and definitely **save large quantities of money that is used to correct (minor) bugs**. If we say that the industry average is about between fifteen to fifty bugs per one thousand lines of code [8] the problem gets scary since projects commonly have thousands and thousands of lines.

So is it possible to model a web application? According to what we have shown in the previous sections, it is not only possible but also a prudent way to face this problem. We started by describing all the requirements that the client wanted and translated them to a very formal tool named *Electrum*. Obviously, this translation took some time but it was really helpful to understand and confirm that the problem we had in hands, was the same problem we had in our minds. A model like this is really useful to check, for example, in what conditions an user can log out; the programmer, in spite of gathering in that moment all the variables and aspects needed to program everything related to log out, he would have all of that information inside the model. Furthermore, a very positive aspect of modeling *SWAP* was the ability to find traces that reproduces its behaviour, traces that can be viewed. To a programmer this component means a lot, since there is a very organized and satisfactory solution that can be explored. Assertions also proved to be a very formal way of finding inconsistencies in the model because it is with them that we can check, for example, the existence of a specific trace.

Something that is also interesting to note is the capability to adapt this model to other applications once it shares some similar features with them. A programmer can take this model, run and test it and change it to his purpose without the necessity of building everything from scratch. It was also part of our intentions to write an understandable model that could be used by other interest engineers or interested researchers.

Formal modeling and specification can be (and already is) something very promising and useful for the development of web applications and we are already seeing this with big companies like *Amazon* that are modeling their systems with the help of formal tools. We really felt it was a very challenging and interesting project.

6.1 Future work

This kind of projects always opens doors to others considering the new ideas that each one of us had to improve it. An example is somehow, trying to run the model in the *Anima*[7] tool (developed at the University of Minho), which is a web application that takes an *Alloy* model and runs it with images, replacing those primitive instances appearance with something much more attractive. That would either mean finding a way to adapt it to *Alloy* or reimplementing *Anima* to work with *Electrum* language.

The current model is seen as an application where web elements come and go, in another words, a fixed page that is always changing. Contrarily to what was done, the model could also have different signatures of pages, those that are available in the application and the interaction would be done taking this approach into account.

As said before, the more specified the model is, the better it represents the system and having that in mind, it makes sense to specify it even more. We think it is important to mention that there is always a way to improvement, and as already said, we can optimize the code developed with certain surgical modifications or even implement new definitions that answer more questions or even make the solution a little more adaptable.

References

[1] statista.com [*Average number of new Android app releases via Google Play per month as of May 2020*]. 2019

[2] TLA+ [*https://lamport.azurewebsites.net/tla/tla.html*]

[3] NuSMV [*http://nusmv.fbk.eu/*]

[4] Alloy [*https://alloytools.org/*]

[5] Electrum [*http://haslab.github.io/Electrum/*]

[6] ConcurTaskTrees Environment [*http://hiis.isti.cnr.it/lab/research/CTTE/home*]

[7] Anima Tool [*https://ise.di.uminho.pt/anima/index.html*]

[8] *How Many Defects Are Too Many?* [*https://labs.sogeti.com/how-many-defects-are-too-many/*]

[9] *Use of Formal Methods at Amazon Web Services* [*https://lamport.azurewebsites.net/tla/formal-methods-amazon.pdf*]