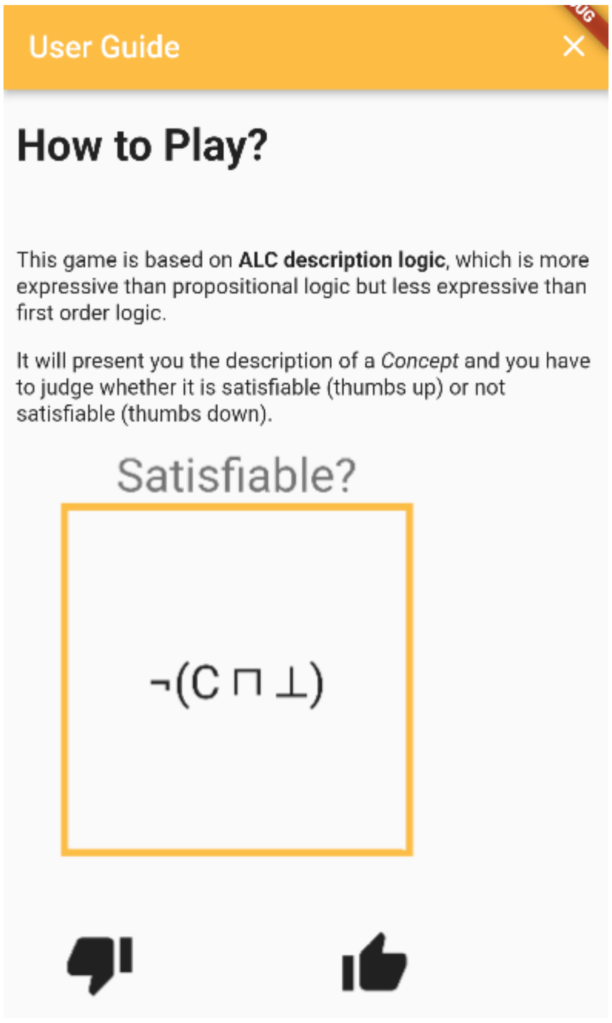
Logic App Development: Coursework Report

Felix Neubauer - 3634473

Introduction



User Guide

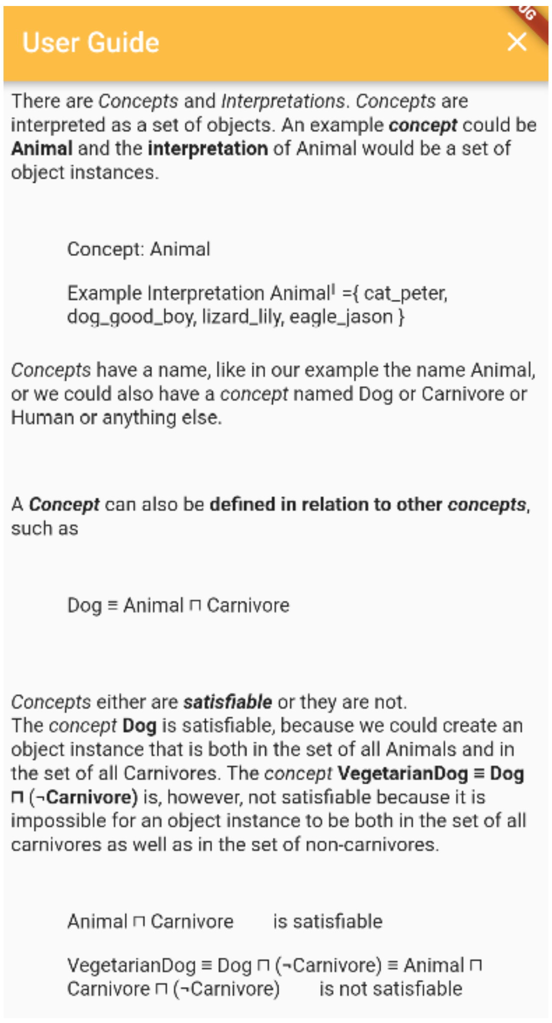
How to Play?

This game is based on **ALC description logic**, which is more expressive than propositional logic but less expressive than first order logic.

It will present you the description of a *Concept* and you have to judge whether it is satisfiable (thumbs up) or not satisfiable (thumbs down).

Satisfiable?

$\neg(C \sqcap \bot)$



User Guide

There are *Concepts* and *Interpretations*. *Concepts* are interpreted as a set of objects. An example ***concept*** could be **Animal** and the **interpretation** of Animal would be a set of object instances.

Concept: Animal

Example Interpretation $Animal^I$ ={ cat_peter, dog_good_boy, lizard_lily, eagle_jason }

Concepts have a name, like in our example the name Animal, or we could also have a *concept* named Dog or Carnivore or Human or anything else.

A ***Concept*** can also be **defined in relation to other *concepts***, such as

Dog ≡ Animal ⊓ Carnivore

Concepts either are ***satisfiable*** or they are not.
The *concept* **Dog** is satisfiable, because we could create an object instance that is both in the set of all Animals and in the set of all Carnivores. The *concept* **VegetarianDog ≡ Dog ⊓ (¬Carnivore)** is, however, not satisfiable because it is impossible for an object instance to be both in the set of all carnivores as well as in the set of non-carnivores.

Animal ⊓ Carnivore is satisfiable

VegetarianDog ≡ Dog ⊓ (¬Carnivore) ≡ Animal ⊓ Carnivore ⊓ (¬Carnivore) is not satisfiable

Licenses
All components developed by me use the MIT license.

Overview

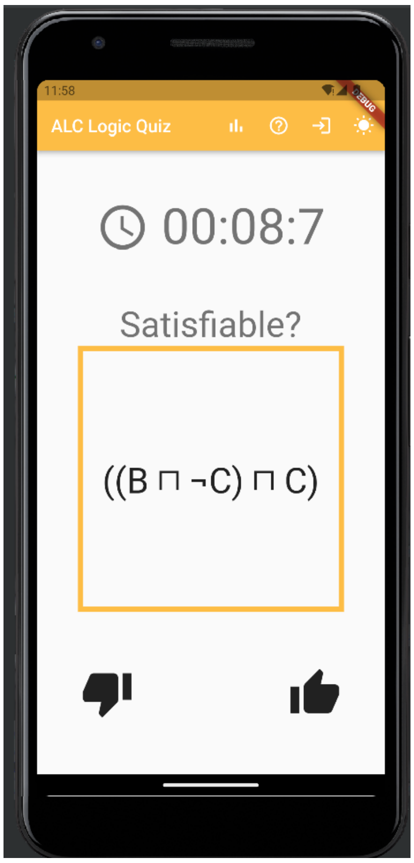


Figure 1

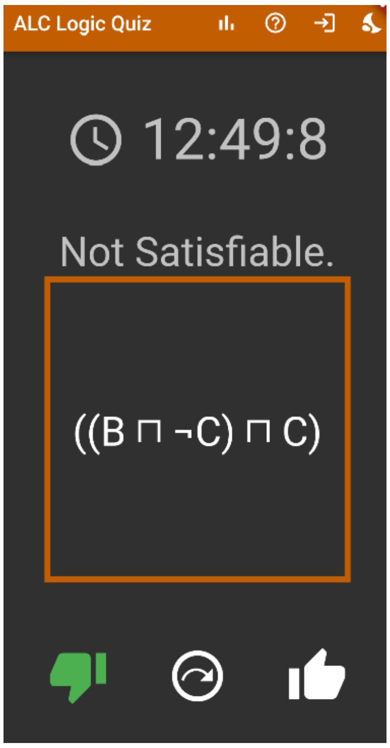
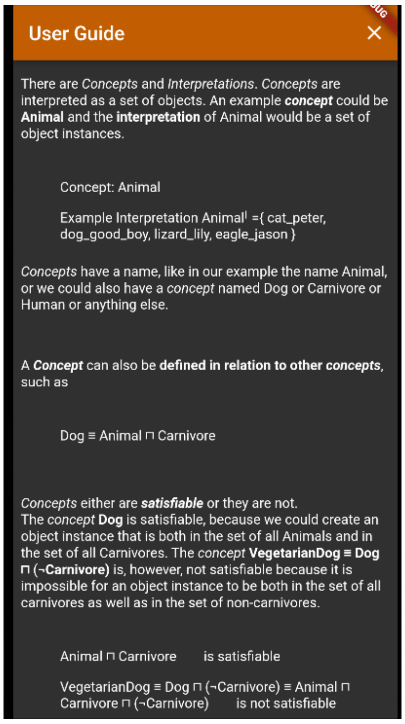


Figure 2



User Guide

There are *Concepts* and *Interpretations*. *Concepts* are interpreted as a set of objects. An example ***concept*** could be **Animal** and the **interpretation** of Animal would be a set of object instances.

Concept: Animal

Example Interpretation Animal^I ={ cat_peter, dog_good_boy, lizard_lily, eagle_jason }

Concepts have a name, like in our example the name Animal, or we could also have a *concept* named Dog or Carnivore or Human or anything else.

A ***Concept*** can also be **defined in relation to other *concepts***, such as

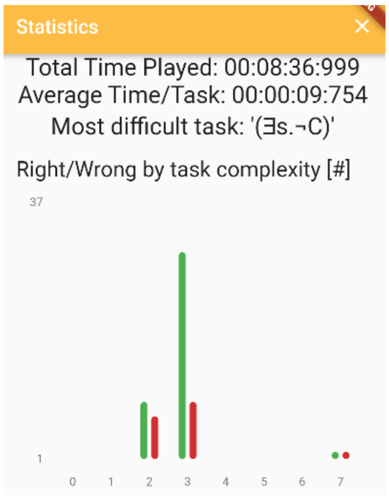
Dog ≡ Animal ⊓ Carnivore

Concepts either are ***satisfiable*** or they are not.
The *concept* **Dog** is satisfiable, because we could create an object instance that is both in the set of all Animals and in the set of all Carnivores. The *concept* **VegetarianDog ≡ Dog ⊓ (¬Carnivore)** is, however, not satisfiable because it is impossible for an object instance to be both in the set of all carnivores as well as in the set of non-carnivores.

Animal ⊓ Carnivore   is satisfiable

VegetarianDog ≡ Dog ⊓ (¬Carnivore) ≡ Animal ⊓ Carnivore ⊓ (¬Carnivore)   is not satisfiable

Figure 3



Statistics

Total Time Played: 00:08:36:999
Average Time/Task: 00:00:09:754
Most difficult task: '(∃s.¬C)'

Figure 4

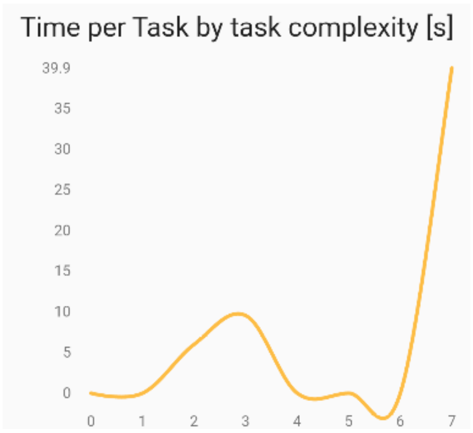
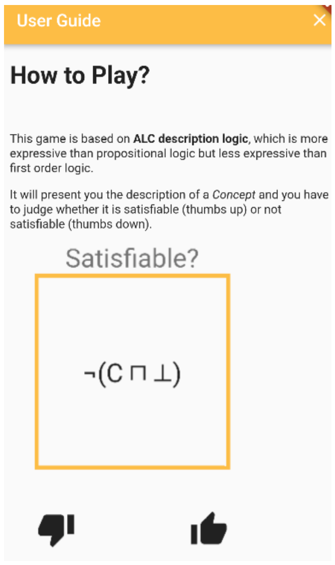


Figure 5



User Guide

How to Play?

This game is based on **ALC description logic**, which is more expressive than propositional logic but less expressive than first order logic.

It will present you the description of a *Concept* and you have to judge whether it is satisfiable (thumbs up) or not satisfiable (thumbs down).

Figure 6

Frontend

- Main Screen (Figure 1 and Figure 2)
  - AppBar to navigate to other screens, such as User Guide and statistics and to toggle between light mode and dark mode
  - Timer that counts how much time the user needs for the task
  - The Task is presented: ALC Logic Concept that is either satisfiable or not

    - Answer Buttons for the user. If the user presses a button, the timer is paused, and the correct answer is be presented. The user can press a button to continue to the next task (Figure 2)
- Statistics Screen (Figure 4 and Figure 5)
    - Shows the statistics of the user, including average time per task, most difficult task and plots regarding time needed or right/wrong ratio by task complexity
- User Manual (Figure 6 and Figure 3)
    - Explains the user how to play the game
- Other functionality
        - When the app starts, it will try to connect to the backend and download the tasks from the backend. If this connection fails, it will re-use the tasks from the latest time the app was running. This is either the tasks from when the app was connected with the backend last time (local storage of downloaded tasks), or if it has never been connected to the backend, it is 10 inbuilt default tasks.
        - User data, such as statistics, are persisted
        - Whenever a task is completed, the app will try to send the user statistics to the backend. As user management is not fully implemented in the frontend due to time reasons, a default user account is used, which is hardcoded into the frontend
        - Task selection uses a learning technique: the tasks provided by the backend are sorted by complexity. The frontend will always randomly select one of the first five tasks that have not yet been solved. A task is marked as solved, when it has been answered correctly more often than it has been answered wrongly. Once all tasks are completed, the app will just randomly select any of all tasks.

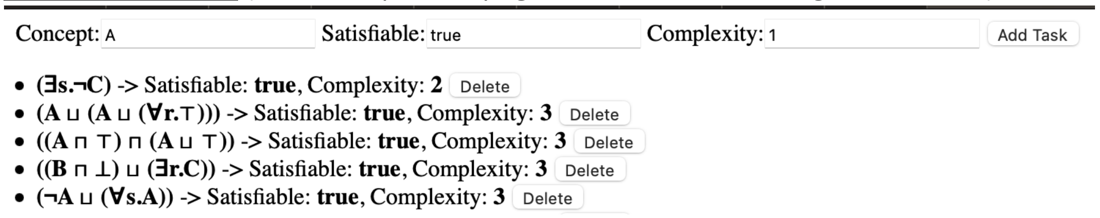
Server-side

- JSON-based REST API
    - GET /tasks: shows list of all tasks
    - GET /users: shows list of all users, for testing purposes
    - PUT /user/<name>: create or update user with given information, includes statistical information. Will only accept user update if it has correct password hash. Example request body:

```
{
  "name": "felix",
  "passwordHash": "my_password_hash",
  "tasksStatistics": {
    "A:true:0": {
      "Attempts": 4,
      "Successes": 4,
      "totalTimeNeeded": 40.1,
      "Task": {
        "Concept": "A",
        "Satisfiable": true,
        "Complexity": 0
      }
    }
  }
}
```

    - GET /user/<name>?password_hash=<password_hash>: returns the given user if the password hash is correct

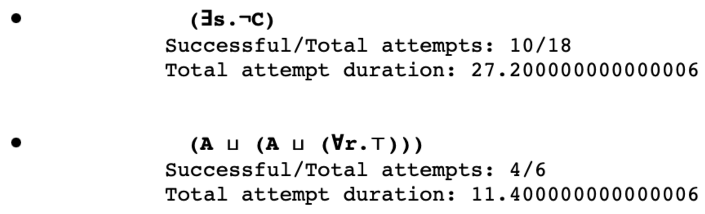
- GET /admin: html web page that lists all tasks and allows deleting a task or adding a new task (delete requires a page refresh for the change to be seen)



Concept: A Satisfiable: true Complexity: 1 Add Task

- **(∃s.¬C)** -> Satisfiable: **true**, Complexity: **2** Delete
- **(A ⊔ (A ⊔ (∀r.⊤)))** -> Satisfiable: **true**, Complexity: **3** Delete
- **((A ⊓ ⊤) ⊓ (A ⊔ ⊤))** -> Satisfiable: **true**, Complexity: **3** Delete
- **((B ⊓ ⊥) ⊔ (∃r.C))** -> Satisfiable: **true**, Complexity: **3** Delete
- **(¬A ⊔ (∀s.A))** -> Satisfiable: **true**, Complexity: **3** Delete

- GET /stats: html web page that lists the statistics of all tasks that have been attempted by a user so far



```
•          (∃s.¬C)
         Successful/Total attempts: 10/18
         Total attempt duration: 27.200000000000006


•          (A ⊔ (A ⊔ (∀r.⊤)))
         Successful/Total attempts: 4/6
         Total attempt duration: 11.400000000000006
```

- Uses MongoDB for persistence of data

User Management

Initially I had set up user management with Firebase, but then after a discussion in class it came out that Firebase should not be used. So, I removed it again and attempted my own user management implementation. Due to time constraints, I was, however, not able to complete it.

Current state:

- The Backend does support different user profiles
  - Users can be added and updated (PUT) and they can be retrieved (GET) via REST API
  - They are persisted in the database
  - The statistics HTML backend shows the aggregated statistics of all users
- The Frontend does in principle support different users, but it has no user management screens. It has a hardcoded user that is used for sending statistics to the backend
- The admin functionality to delete users or reset passwords is not implemented, nor is the frontend high-score table
- As the backend is intended to support user management, the statistical information is not transferred to the backend anonymously, but instead it is transferred in combination with the user profile

Task Generation

- Is implemented
- All tasks are different and correct
- The program as well as the tasks are included in the repository

Setting up the project

- Frontend
    - Running is the same as you ran our exercises for Flutter. Preferably use Android Emulator, as the frontend uses the hardcoded address to the backend http://10.0.2.2, which is what we need to use for the android client, but it would have to be changed if we would use e.g. Google Chrome for the frontend and want it to connect to the backend
- Backend
    - Install Docker
    - Run the script */serverside/run_mongo_db_container.sh* to start the MongoDB database
    - Run the Go file */serverside/cmd/main.go* to launch the GoLang server
- Task Generation
    - Run */generator/main.go*
    - Optionally change the parameters in the code to get different tasks as output

Technical Documentation

Frontend

- Focus on clear and consistent file structure: widgets, screens, models and handlers have their own respective folders
- Most parts of the Frontend I had already learned how to do during the Lab Course Exercises
- Data is persisted using *shared_preferences* due to simplicity and because it is cross-platform compatible
- Decision that the main screen is the core of the App and contains all options such as toggling between light and dark mode or going to a different screen. Other screens don't have such functionality, they only do their specific job and have a button to go back

Backend

- Files organized in the standardized GoLang way (see https://github.com/golang-standards/project-layout)
- One main.go file that connects to the MongoDB database, loads the tasks (first offline and if available also from the database) and then starts the REST server
- The core functionality can be found in server.go. The *ServeHTTP* function handles all incoming http requests and forwards them to the corresponding sub-functions, such as a function to handle user requests or one to handle task requests. It is hand-written and does not use any third-party libraries
- The data structures are defined primarily in task.go and user_profile.go
- Database-related code is in mongo_task_store.go and mongo_user_store.go. Due to time-reasons, for updating a user or the tasks, instead of finding out how to update data in MongoDB, I decided to simply delete the old data and then insert the new data

Task Generation

- main.go coordinates everything: different parameters are defined there and the sub-programs are called to generate the tasks based on those parameters
    - generate many concepts using generateConcepts()
    - group those concepts by complexity (e.g. operator count), resulting in one list of concepts for every number of complexity
    - pick random tasks from those generated concepts until all desired metrics are reached. To pick a task, the satisfiabilities of the task candidates need to be computed, as well whether a task is functionally equal to already chosen tasks

generateConcepts($n_{basicConcepts}, n_{roles}, n_{levels}$):

1. Concepts $C = \left\{C_1, \dots, C_{n_{basicConcepts}}\right\} \cup \{\top \cup \bot\}$ and roles $R = \{r_1, \dots, r_{n_{Roles}}\}$
2. $lvls = \left[C, lvl_1, \dots, lvl_{n_{levels}}\right]$,with $lvl_x = combineConcepts(lvl_{x-1})$

$combineConcepts(lvl_n)$:

- For every $C \in lvl_n$, add $\neg C$ and C
- For every combination $A, B \in lvl_n, A \neq B$, add $(A \sqcup B)$ and $(A \sqcap B)$
- For every combination $A \in lvl_n, r \in R$, add $(\exists r.A)$ and $(\forall r.A)$

3. Return $lvl_{n_{levels}}$, which contains all generated concepts from all levels, including base concepts

isSatisfiable($concept$):

1. Convert concept to negation normal form
2. Put all data into a tableau and apply satisfiability algorithm. Whenever the union operator is used, the tableau will split into two tableaus
3. Check if any of the resulting tableaus has a solution without having a clash

The tricky part here is that I did not use textbook solutions as they would be much more effort to implement. The tasks I want to generate are restricted, so I could think about a custom and simpler solution for my use case. The base ideas are:

- As defined in alc_logic_model.go, all logic operators are implemented as structs
- Code-wise, every concept has one root element (an operator or a BaseConcept)
- To make things simpler, instead of treating a tableau as a complete ABox, I treat a tableau only as a container for one singular concept, where all unfolded base concepts of that concept are put into
- If the root element is an operator, we can apply its unfolding rule. In case of *A INTERSECTION B*, this would mean we would add both the concept *A* and the concept *B* to our list of concepts within the tableau. This step we repeat until we have applied all unfolding rules possible. Whenever there is a conflict, this means the tableau does not have a solution.
- To handle quantifierExists, I introduced a relationship attribute to the tableaus. A tableau can have a relationship with a given Role to another set of tableaus. For a tableau to be satisfiable, it must not have a conflict, plus if it has a relationship, that relationship must point towards at least one tableau that also is satisfiable
- After all possible unfoldings are performed (excluding quantifierForEach), as a final step the quantifierForEach rule is applied on all relationships with the given Role

isFunctionallyEqual($coneptA, concept$):

To check for functional equivalence, both given concepts are normalized and then compared. Note that normalization in this case is renaming of the Base concepts.
E.g. $D \sqcup C$ and $B \sqcup A$ would both become $A \sqcup B$. This normalization does not change the order of the operators. $C \sqcup (A \sqcap B)$ is seen as different than $(A \sqcap B) \sqcup C$.
We can easily check for equivalence by using satisfiability to check for subsumption and that to check for equivalence. However, this equivalence check assumes different concept names to be different concepts. While it would detect $C \sqcup (A \sqcap B)$ as equivalent to $(A \sqcap B) \sqcup C$, it would see $D \sqcup C$ as different from $B \sqcup A$, since from an ALC Logic point of view, they are different. From a task generation point of view though, they should be treated as identical. Therefore, if we assume concept names to be interchangeable and want to normalize our concepts but also normalize the concept operator order, this quickly becomes a highly complex task. I had spent a lot of time of evaluating solutions, such as assigning different operators different values and then re-ordering the concept based on those operator values, but in the end decided it would be better to leave that part out. My manual inspection has confirmed that the generated tasks don't have issues with duplication and the normalization of base concept names already serves as an efficient filter.

Tests
Testing is limited to manual running of the individual components and trying out different scenarios and visually confirming that the behavior is as expected.

Packages
Frontend

- *flutter_riverpod* for state management
- *go_router* for navigation
- *yaml* for reading the default tasks from the yaml file
- *auto_size_text* for displaying the task within a box of a fixed size, automatically scaling the text to match that size
- *shared_preferences for persistence of user data (statistics) and tasks*
- *intl* for formatting of the time
- *fl_chart* for the plots in the statistics screen
- *http* for sending http requests to the backend
- *flutter_html* for the user guide screen that embeds HTML

Backend

- *yaml* for reading tasks from the yaml file
- *go.mongodb.org/mongo-driver* for the MongoDB client
    - this package has many subdependencies

Task Generation

- *yaml* to export the tasks into a yaml file

Sources
https://stackoverflow.com/questions/54775097/formatting-a-duration-like-hhmmss for time formatting in Flutter

https://github.com/imaNNeoFighT/fl_chart/blob/master/example/lib/bar_chart/samples/bar_chart_sample2.dart example for bar chart in flutter

https://github.com/imaNNeoFighT/fl_chart/blob/master/example/lib/line_chart/samples/line_chart_sample9.dart#L102 example for line chart in flutter

https://logic4free.informatik.uni-kiel.de/llocs/Negation_normal_form algorithm taken over for normalization of logical formulas

Conclusion

Within this module I have learned a lot, starting with Flutter, going to GoLang but also reaching into HTTP templates and MongoDB queries. It consumed a lot of time, but I think it was worth it. At the same time, I am exhausted now and am keeping the conclusion short. I am glad I managed to complete almost all of what I had planned, and I am satisfied with the resulting frontend, server functionality and task generation. It was fun to work with ALC logic and I like the type of tasks that the app now has. The user management somewhat frustrated me, because I had successfully connected Firebase and set up all screens, but now without Firebase I don’t have user management in the frontend. All in all, I am glad I selected the module, but for the next semester I will probably try to choose less time-intense modules, also because this semester I had chosen to strongly limit the time I put into a group project, so I could finish the tasks of this lab course. I had personally overestimated my own capabilities and underestimated the time all of the chosen modules in sum required. I hope my effort will be rewarded with a good (and of course fair) grade ;-)