Step 4

```
- What are the pros/cons associated to the meta-modeling approach?
What is the cost of defining a meta-model?
What is difficult in this activity?

- From the user point of view, what does it change?
Is the approach usable for large apps?

- Consider the LED app and the counter one as two separate models.
Is it possible to automate the creation of the final app
based on these two models?

- What about the readability of the generated code compared
to the previous one "by hand"? Its debugging capabilities ?
Its extensiveness?

- Explain the interest of modeling in terms of genericity,
functional property verification.
```

- The main pros that seem to come with meta-modeling are: readability, modularity and robustness. The readability comes from the fact that all actions are classified (into classes and enumerations), with a clear view of the dependency between them thanks to the UML modeling of our system. The modularity is not a given but is more obtainable in a well-designed meta-model because, when the possibilities of the system (components, signals and actions) are clearly defined and in an independent manner, each part of them acquires a good reusability. For example, the actions which allow to print numbers on the 7seg can be reused in any state of any fsm. Robustness is achieved because the executed code is no longer written by hand but generated. It means that most of the errors become logic errors (for example, by giving the wrong pin to an actuator, or setting up the wrong transition between two states) rather than language or syntaxic errors.
    - One drawback currently is that, after the meta-model is defined, translating it in Java classes is cumbersome as it requires to defined many new objects, sometimes in a way that is a bit repetitive - though this surely can be tackled by more abstraction. For example, the necessary use of Arrays.asList as soon as several elements have to be given is annoying. Maybe it comes from the fact that we are still using a GPU to write our model, with fixed parsing tactics that do not suit our needs. Another con is compatibility with other programs or libraries.

  - Time: it takes some time to define the meta-model, implement it, write it, etc. For the task at hand, it takes more time than it would have to write the code already in C. But the result surely is rewarding. It probably doesn't cost any expressivity, because the translation to code allows the use of any function, behavior etc. (for example, we could reuse the binary operators on registers from the Action class).
  - The main difficulty was to translate the meta-model into C code in a way that would scale to many elements and that would, obviously work. In particular, defining several state machines and then (this was our approach) fusing them to obtain a machine which runs them all the same time in a recursive way is a bit tricky.

- The user can now define its program in terms of components, sensors, interactions between them and planned behaviors. This behavioral description probably allows to focus on the functionality rather than the technicalities of the programming. Nothing seems to prevent it, and it is already reusable-as-is in some sense: if the code generation scales well, if the abstraction classes do not become bloated (though it is always possible to create new ones). Furthermore, the qualities of the generated code make it so that this is in fact surely the best option for large apps.
- It is indeed possible. Our approach was to model each of them as a Fsm (finite-state-automaton) class. The code generation machine then realizes an union of all the defined Fsm (by doing the product of the states into big single states, and unifying the transitions). This works well for two Fsm but have not tested it for more yet, and some more work is need to have full frequencies support.
- The generated code is actually quite readable and has a very regular syntax due to being generated. It is a bit verbose right now (28/09/18) due in part to not enough using abstractions for actions (defining functions action_%s would greatly diminish verbosity for example). Its debugging capabilities are probably identical: there are more lines so it is difficult to read it as a whole, but the code patterns are regular and generic so an error is quickly picked up by the debugger and easily found. However, it is probably still bad in order to debug logic errors (wrongly defined pin, etc.) As said, it can be extensive (500 lines program as of now, but will probably be lessened soon).
- Modeling allows to defined classes and their interactions with other classes. This restrains their possibilities to specific actions and possibilities of realizations (types), which in turn allows to ask that some properties would be enforced on these structures. A first easy one was to specify the type of the variables inside the UML diagram: it ensured that these variables would be initialized to computable values. There are other structural properties that could be verified.