Clinguin Project Document

2022-07-06 Clinguin

1 Ideas

1.1 Project Summary

The *Clinguin* project can be seen as a language extension to a Logic-Program, which adds the functionality to create a User-Interface (UI). Capability wise the UI shall be capable of dynamic updates, i.e. not just a static display of a stable-model(s), but a program where one can interact with the logic program.
An example for this is the Sudoku problem. Using the Clinguin one is able to generate a Sudoku playing-field, where the field is interactive in a sense, that if one selects one of the possible values a single field may have, the program automatically updates the not selected values and fills them in, if only one possibility exists. This is similar to forward checking for Constraint-Satisfactory-Problems (CSPs).

1.2 Prototype

Susana Hahn has written a prototype, which implements the Sudoku example from above. The used programming language is Python, further Clingo libraries (like the Clingo-Object-Relational-Mapper (CLORM)) are and the UI-framework tkinter (https://docs.python.org/3/library/tkinter.html) are used. Syntactically the prototype provides an extension by the following keywords (predicates which are used as keywords/tokens):

- *window(<attribute-name>,<argument-key>,<argument-value>)* - Defines the general structure/-size/etc. of the displayed field
- *widget(<type-name>,<ui-id>,<master-id>)* - Adds a widget (ui-element/item) to the view. The accepted types are *frame* (container) and *menu* (dropdown menu). The *ui-id* should be unique and the *master-id* represents the enclosing item (for top-level frames this is the *window*).
- *geo(<ui-id>,<ui-layout>,<argument-key>,<argument-value>)* - Used for specifying the exact location of a widget. The *ui-id* is of a widget, the *ui-layout* can be chosen from a tuple of *(grid, place, pack)*, the *argument-key* is used for specifying what one wants to set (e.g. row) and the *argument-value* is the corresponding value.
- *config(<ui-id>,<argument-key>,<argument-value>)* - The *ui-id* is of the corresponding widget, the *argument-key* is the property to set (e.g. font width or some color) and the *argument-value* is the corresponding value to the key.
- *opt(<ui-id>,<value>,<original-state>)* - This is used for the dynamic updates of the gui. The *ui-id* is for the corresponding widget. In the prototype the stable matches are computed via the

brave option, and for each possible value the *original-state* can have, also one *opt* is generated. This is used for computing what values a single sudoku field may have.

In general the program works like this: The program scans the logic-program for the syntactic elements above, and then creates the ui with the elements (function *create_layout()*, uses cautious reasoning). Then the dropdown-menus (i.e. the Sudoku cells) are filled, with the function *update_options()* (uses brave reasoning). Here for each such dropdown-menu a callback function is added, which when clicked adds an assumption, that a certain value must be used for the sudoku field. Then the process repeats itself.

1.3 Architecture Proposals

1.3.1 Initial Thoughts

The architecture of Clinguin should be flexible in a sense, that it should be compatible with several UI-Libraries/-frameworks. Further it should be efficient (which is not handled at this point, but e.g. incremental updates can be made instead of full updates).
The need for flexibility leads to the conclusion, that the program may not directly construct the *tkinter* (or some other frameworks) window, but must create an internal representation first (See Figure 1):

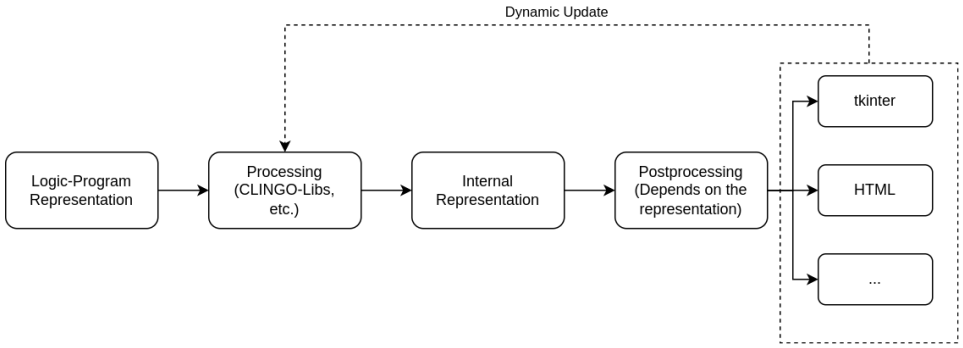

Figure 1: Informal idea of the workings of the program

Therefore for the construction of the program a few things have to be defined/thought about:

1. Logic-Program Representation
   - Syntax of language extension
   - Initial starting point → syntax of prototype
   - In general the syntax should be defined in a way that helps understanding of the program
2. Processing
   - Produces internal representation from the logic program
   - Main thing: Efficiency
   - Other things: Libraries to use (depends on selected internal representation), etc.
3. Internal Representation

- **Very important to think about now**: Many different options exist, like *XML*, *Json*, *Class-Hierarchy*, *HTML*, ...
- Different representations have different pros/cons:
  - XML - Pro: Clearly defined structure, portable format. Cons: Format needs to be parsed again before it can be used (efficieny loss), postprocessing will be pretty complex
  - Json - Pro: For hierarchical GUI tools postprocessing will not be that much, portable format. Cons: Format needs to be parsed again, for non hierarchical GUI tools postprocessing will be quite complex
  - Class-Hierarchy - Pro: Efficiency, for hierarchical GUI tools postprocessing will be relatively easy. Cons: Non portable format (one is limited to Python represetation), for non hierarchical GUI tools postprocessing will be quite complex.
  - HTML - Pro: Direct representation of GUI elements, i.e. one can directly view the html, otherwise similar to XML. Cons: For other views than HTML postprocessing will be pretty complex

4. Postprocessing
   - Depends entirely on the chosen internal representation and chosen GUI-framework
   - Could be implemented as a *1:1* match between internal representation and GUI-framework, which would lead to a modularization of the Clinguin
   - Efficiency will be important here...
5. Initial GUI-Framework to use (tkinter, web/HTML, other)

1.3.2 Monolithic Custom

The idea is to think of an architecture, which is a monolith. The architecture should be designed in a way that fulfills most requirements. One requirement which would be hard to fulfill is the Json/Independability requirement, i.e. that the gui may be switched in the future.
I didn't explore on this approach much further, but it should definitely be possible, if one drops the above mentioned requirements.

1.3.3 Monolithic Model-View-Controller (MVC)

As the MVC pattern is a widely spread Gui-Architectural pattern it makes sense to consider it. In Figure 2 the general idea of MVC is depicted, where normally there exists one such MVC-triad per GUI element, e.g. one for the top level gui element, which contains "children", like menu-bar, grid-controller, etc. (as depicted in Figure 3).

For Clinguin this architecture open two possible tracks of implementation, where it seems to be the case that for both tracks it would be pretty hard to get the callbacks right while using an independent representation like Json.

My main reason of concern in both cases is how would the notification process (callbacks) work

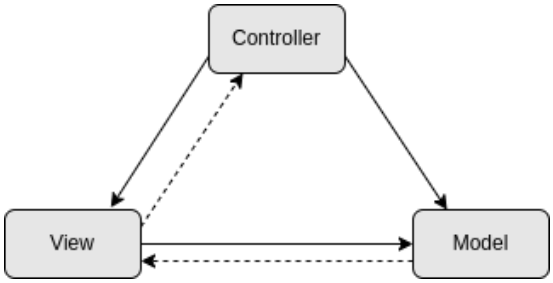

Figure 2: General MVC idea

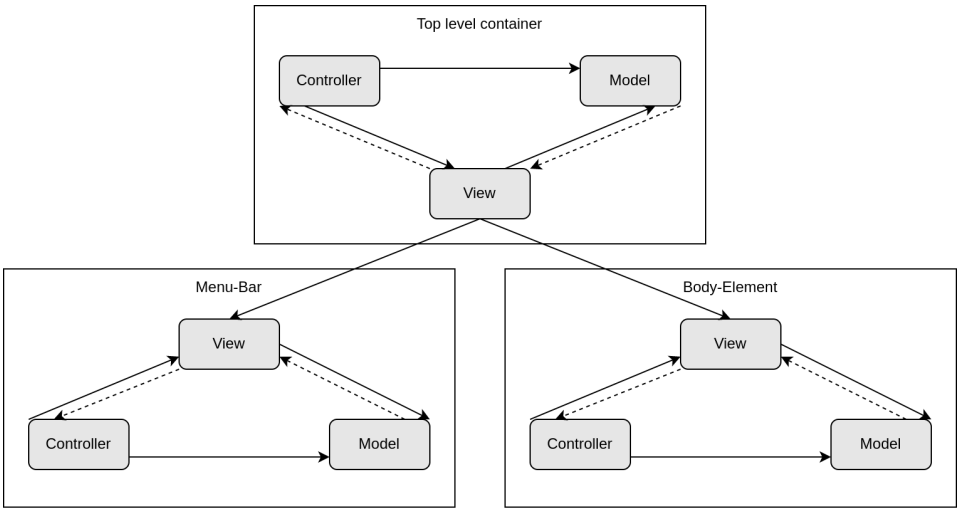

Figure 3: MVC Hierarchy

together with Json?
As far as I know, when using MVC one "hard-codes" the View, the Controller and the Model, each in a seperate file, where then a framework (like Windows-Presentation-Foundation (WPF) for C# - which is actually the MVVM principle, but pretty similar...) connects the files.
On the other hand, when one uses Json as a means for the intermediate representation, one normally uses some kind of Message-Oriented-Architecture (MOA), like a simple Client-Server system.

- Implementation of Clinguin as a single MVC triad
  - Implementing this in a way, that multiple elements are handled in one MVC should violate the MVC design principle
  - Normally in MVC the View should be static in a sense, that the elements of a view doesn't change, just the content - but here this would be violated
  - When using Json for data transfer, I can't imagine how the callbacks should work
- Implementation of Clinguin as a MVC hierarchy
  - Either as a code generator (i.e. a program which generates the python code, should fulfill the MVC design principle, but seems to be hard to implement)
  - Or implement it just via OOP - but which would still be hard and it would even be harder

to get the callbacks right.

- When using Json for data transfer, I can't imagine how the callbacks should work

1.3.4 Simple Message based approach (Client-Server)

The next option to look at is the Client-Server approach. The main point for this architecture is the flexibility, that one can use the Json data as an intermediate representation and use whatever gui one finds suitable. Further the architecture is straightforward (also the callbacks).
The server architecture could be implemented as a normal layered-architecture, where the data layer is responsible for the communication with the underlying logical-data-structrure (i.e. CLORM), the logic layer contains some business logic (might actually not be necessary) and the presentation layer handles incoming requests, processes them (i.e. forwarding to the logic/data-layers) and then formats the response as a Json.

The Client just gets the Json and constructs with this information a gui. If a user clicks on a field a request is sent to the server, which processes it, computes the stable models, etc. . Implementation wise the client can be constructed in a variety of different ways, a straightforward one would be that the Json contains enough information that one may construct a gui with possible actions and if an action occurs, sends it back. The client may be written in any programming language.

Another pro: When one uses a *REST*-API, then One drawback with this approach is, that the development time might be higher than a monolithic approach.

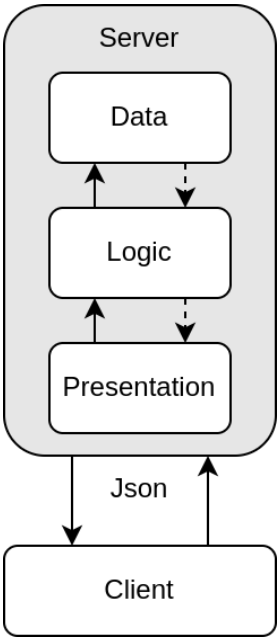

Figure 4: Cient-Server approach