Clinguin: Building User Interfaces in ASP

Susana Hahn

University of Potsdam, Germany

From education to industry, Answer Set Programming (ASP) plays a crucial role in solving complex problems. Its effectiveness is amplified by tools that facilitate the development of interactive applications. However, end users often prefer graphical interfaces, leading to challenges in frontend development as it requires skills in languages typically outside the expertise of ASP developers and researchers. *Clinguin* addresses this gap by enabling ASP developers to create interactive User Interface (UI) prototypes using ASP alone. Following the workflow pattern of our visualization system *clingraph*[1], *clinguin* defines UIs as sets of facts, and facilitates continuous interaction with ASP solvers based on user-triggered events.

When it comes to creating interfaces tailored for specific problems, a PROLOG-based approach was explored [2, 3] to design interfaces declaratively using the XML dialect XUL. Additionally, the area of Automatic User Interface generation [4] explored model-based UIs which was later extended with contextual information and ASP [5]. However, creating problem-specific UIs with ASP remains an open challenge. Nevertheless, the necessary tools to handle interactivity around ASP solvers are already available. Interactivity in ASP was explored in [6], and latter incorporated into *clingo* as multi-shot capablities [7], allowing for the continuous solving of logic programs that undergo frequent changes. *clingo*, facilitated this through its API, enabling the implementation of reactive procedures involving grounding and solving. *Clinguin* leverages these capabilities to create user interfaces that interact with the solver.

Clinguin employs a Client-Server architecture, where communication occurs via an HTTP protocol using JSON. In essence, the server is responsible for executing *clingo* and computing the information required to define the UI. This process unfolds in two distinct steps. Firstly, the *domain-state* is computed using the domain-specific encodings, and it is defined by facts that differentiate between user-selected atoms, potential selections, and inferred atoms. In the next step, the server utilizes the provided UI encoding to generate atoms defining the layout, style, and functionality of the interface, collectively referred to as the *ui-state*.

The workflow, depicted in Figure 1 (Appendix A), can be summarized as follows: The server is started by providing domain files and a UI file as command-line arguments. Using the domain files the server creates a *clingo* control object, namely *domain-control*, which employs multi-shot solving. When the client is launched, it requests the *ui-state* from the server. Upon receiving the *ui-state* in JSON format, the client utilizes a front-end language [1] to render the corresponding

[1] The project started with tkinter, an OS-dependent frontend, and has since transitioned into a web-based frontend developed in Angular to enhance both functionality and style.

UI. Subsequent user interactions with the UI generate new requests to the server, providing details about the selected operations. The operations, defined by the server, allow users to interact with the *domain-control* in different ways, such as adding a selection as an assumption, setting the value of an external atom, or obtaining the next solution. Once the server completes the selected operations, it constructs a hierarchical structure of the updated *ui-state* and returns it to the client for rendering.

Updating the *ui-state* stands as a key process on the server, involving three distinct solve calls on the *domain-control* to determine the brave consequences (atoms considered as "possible"), cautious consequences (atoms considered as "required"), and the first model. These outputs are combined to form the *domain-state*, presented as a set of facts, with the brave and cautious consequences represented in predicates `_b` and `_c`, respectively. The *domain-state* is then expanded with atoms representing contextual information (*clinguin-state*), including data collected by the frontend `_clinguin_context(K,V)`, as well as `_clinguin_assume(A)` for each assumption `A`, and constants `_clinguin_browsing` and `_clinguin_unsat` to determine *clinguin*'s current state. Subsequently, a separate control object is utilized to generate the *ui-state*. Using the *domain-state* as input, the UI encoding produces a single stable model containing the atoms of the *ui-state*. The *ui-state* is defined by predicates `elem/3`, `attr/3` and `when/4`, which are used to specify the UI's layout, style and functionality, respectively. These atoms are mapped into Python classes using *clorm*[2], a Python library that provides an Object Relational Mapping interface to *clingo*. Specifically, an element `X` of type `T` inside element `X'` is defined by atom `elem(X,T,X')`. The attributes of an element, such as position and style, are specified by `attr(X,K,V)`, where `K` and `V` denote the attribute's name and value, respectively. The reactive aspect of the UI is defined by `when(X,E,T,P)`, interpreted as follows: when event `E` is triggered on element `X`, it is followed by an action of type `T`. These actions can be of three types: `update`, where `P` is a triple `(E',K,V)` signifying that the value of attribute `K` on element `E'` is updated to `V`; `context`, where an internal context, defined as a dictionary, is updated using a `P` as the key-value pair; and `call` where `P` represents one or multiple operations on the server that will be executed.

Clinguin was designed in a modular fashion, allowing flexibility to cater to diverse requirements. It offers the option to define and overwrite operations, as well as customize how the UI is updated and the contextual information in the *clinguin-state*. Currently, the system is actively used to create prototype UIs in various domains, including areas such as Study Regulations [8], Product Configuration and Job Shop Scheduling. Figure 2 (Appendix A), shows snapshots of prototype UIs for the sudoku and graph coloring problems. For further reference, the complete codebase and numerous examples are accessible in the Git repository[3].

[2] `https://github.com/potassco/clorm`
[3] `https://github.com/potassco/clinguin`

References

1. S. Hahn, O. Sabuncu, T. Schaub, and T. Stolzmann: clingraph: ASP-based Visualization. In: G. Gottlob, D. Inclezan, and M. Maratea. Proceedings of the Sixteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'22). Lecture Notes in Artificial Intelligence, pp. 401–414. Springer-Verlag (2022). DOI: 10.1007/978-3-031-15707-3_31
2. C Schneiker, M. Khamis, and D Seipel: Prolog Server Faces–A Declarative Framework for Dynamic Web Pages. (2010)
3. C Schneiker and D Seipel: Declarative Web Programming with PROLOG and XUL. In: Proc. 26th Workshop on Logic Programming (WLP) (2012)
4. E Schlungbaum and T Elwert: Automatic User Interface Generation from Declarative Models. In: CADUI, pp. 3–17 (1996)
5. J Zakraoui and W Zagler: A logical approach to web user interface adaptation. In: Information Quality in e-Health: 7th Conference of the Workgroup Human-Computer Interaction and Usability Engineering of the Austrian Computer Society, USAB 2011, Graz, Austria, November 25-26, 2011. Proceedings 7, pp. 645–656 (2011)
6. M. Gebser, P. Obermeier, and T. Schaub: Interactive Answer Set Programming: Preliminary Report. In: S. Ellmauthaler and C. Schulz. Proceedings of the International Workshop on User-Oriented Logic Programming (IULP'15) (2015)
7. R. Kaminski, J. Romero, T. Schaub, and P. Wanko: How to Build Your Own ASP-based System?! Theory and Practice of Logic Programming 23(1), 299–361 (2023). DOI: 10.1017/S1471068421000508
8. S. Hahn, C. Martens, A. Nemes, H. Otunuya, J. Romero, T. Schaub, and S. Schellhorn: Reasoning about Study Regulations in Answer Set Programming (Preliminary Report). In: To appear in ASPOCP'23: 16th Workshop on Answer Set Programming and Other Computing Paradigms (2023)

A Appendix A

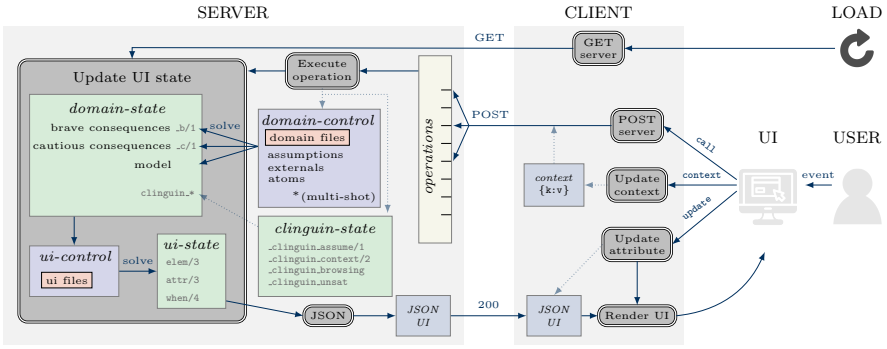
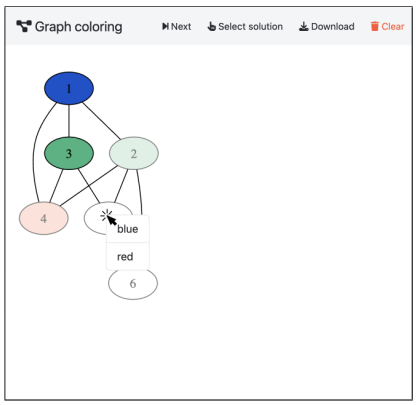
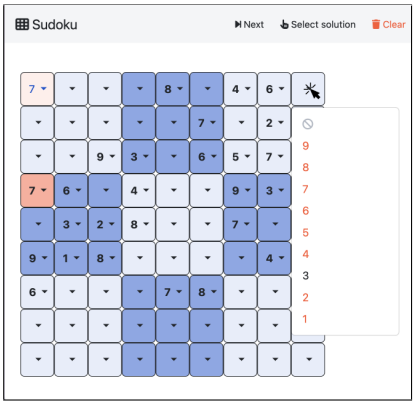


Fig. 1: *clinguin*'s workflows for loading the UI and reacting user-triggered events. Input files are shown in pink. Green sections are composed of ASP facts.



(a) Interactive gaph coloring, with *clingraph* integration.



(b) Iteractive sudoku with values leading to UNSAT in red.

Fig. 2: UIs using *clinguin*.