Auto-Generating Visual Editors for Formal Logics with Blockly

Angelo Ferrando[1][0000−0002−8711−4670], Peng Lu[1], and Vadim Malvone[3][0000−0001−6138−4229]

[1] University of Modena and Reggio Emilia, Modena, Italy
angelo.ferrando@unimore.it
[2] Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France
vadim.malvone@telecom-paris.fr

Abstract. Formal logics are central to the specification and verification of computational systems, yet their adoption outside highly specialised domains is hindered by steep learning curves and error-prone textual notations. Making these notations more approachable is particularly important in education and in settings where non-expert stakeholders need to engage with formal reasoning. We present a framework that automatically generates block-based visual editors for formal logics using the Blockly library. From a structured JSON specification of syntax and composition rules, our tool produces browser-based editors in which formulas are constructed by combining graphical blocks rather than writing code. This approach lowers syntactic barriers for learners and non-experts, while allowing experts to define new logics without manual interface design. Although integration with verification backends is planned, the tool already provides a reusable foundation for accessible and customisable logic editors.

Keywords: Logic Editor Generation · Block-Based Specification · Blockly.

1 Introduction

Formal logics are key to specifying and verifying computational systems, particularly for modelling safety-critical properties and concurrency [19]. In highly regulated domains such as railways, avionics, and automotive, formal methods are already entrenched, supported by training pipelines that prepare domain experts to handle complex notations. However, outside of such contexts, their accessibility remains a barrier: mainstream software engineers, students, and non-expert stakeholders often struggle with precise syntax and abstract semantics [5]. This is particularly evident in educational settings, where empirical studies have shown that students may find formal notations intimidating and error-prone [12]. In these cases, visual representations and block-based metaphors can lower the entry barrier, offering a gentler on-ramp to logical reasoning without compromising formal rigor.

This gap is especially problematic in domains like software engineering and multi-agent systems, where correctness is critical but tooling is often inaccessible.

Despite advances in verification backends, front-end logic specification remains mostly unchanged: users still write formulas in textual syntaxes that are error-prone and hard to learn—especially for non-experts [4].

Visual representations have been explored to improve usability [9], but existing tools are often domain-specific and not generalisable. Beyond logic-specific editors, our work connects to the broader tradition of visual programming languages, such as Scratch and Snap! [10], which demonstrate how block-based metaphors can improve accessibility and learning.

We address this challenge with a logic-agnostic framework for the automatic generation of block-based visual editors for formal logics. Built on the Blockly library [17], our system allows users to construct formulas by composing graphical blocks. Formal methods experts provide a structured JSON [3] specification of the logic's syntax; from this, the framework automatically generates a browser-based visual editor with drag-and-drop blocks, syntactic constraints, and code generation.

This approach offers three main benefits: it decouples logic definition from UI implementation, enables accessible formula construction for non-experts, and supports reuse across different logics, from standard temporal systems to domain-specific languages.

In this paper, we detail the framework's design, architecture, and implementation. We show how logic specifications are expressed in JSON, how the generation pipeline produces Blockly-compatible editors, and how the result enables constraint-aware formula construction. A case study on Alternating-time Temporal Logic (ATL) [1] demonstrates the flexibility of our approach.

Our broader vision is to lower the entry barrier to formal methods by providing customisable, accessible tools for logic specification and analysis. While this work focuses on editor generation, it sets the stage for future integration with verification pipelines—empowering a wider range of users to engage with formal techniques.

2 The approach

Our framework generates visual editors for formal logics from structured specifications authored by experts. While the specification is expressed in JSON—which is itself a form of structured coding—it is lightweight compared to implementing a visual editor by hand, and requires no knowledge of Blockly or front-end programming. The effort resembles writing a formal grammar or BNF description rather than programming a tool, making it accessible to logic experts familiar with syntactic definitions.

Figure 1 shows the architecture: a structured logic specification feeds into a generator that produces Blockly-compatible artifacts, which power a browser-based editor where non-experts can build valid formulas interactively.

To ensure generality and logic-agnosticism, our design follows three core principles: (i) logic specifications are written in structured text, without requiring

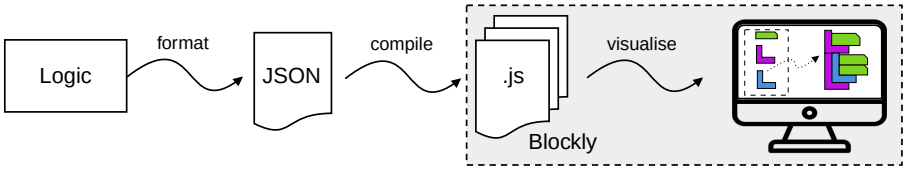

Fig. 1. Overview of the architecture highlighting the automatic translation from logic specifications to visual editors. Automatically generated artifacts are shaded.

visual programming knowledge; (ii) visual editors are generated fully automatically, with no manual or Blockly-specific steps; and (iii) the system is modular, allowing updates or new logics via specification changes alone.

We illustrate our approach using a fragment of Linear Temporal Logic (LTL) [18] as a running example. Our specification language is a structured JSON format that defines logic syntax through *constructs*. Each construct is either a *primitive* (e.g., an atomic proposition) or a *composite* (e.g., a temporal operator with arguments).

Listing 1.1. Primitive construct for atomic propositions.

```
{
"type": "primitive",
"name": "atom",
"format": "%string"
}
```

Listing 1.2. Composite construct for the LTL `F` (*eventually*) operator.

```
{
"type": "composite",
"name": "F",
"format": "F %LTL"
}
```

The `format` field is central to each construct: it defines both the block layout and the code output. Fixed symbols (e.g., `"F"`) determine block labels, while placeholders (e.g., `%string`, `%LTL`) indicate inputs and guide code generation. For example, `%string` creates a free text field, while `"F %LTL"` creates a block with label `"F"` and a subformula slot accepting elements from the `LTL` group.

To simplify constraints and promote reuse, the framework allows defining named *groups* of constructs that can be used interchangeably. This avoids redundancy and improves maintainability. For example:

Listing 1.3. Group definition for valid LTL subformulae.

```
{
"type": "group",
"name": "LTL",
"elements": ["atom", "AND", "OR", "NOT", "X", "U", "F", "G"]
}
```

Placeholders like `%LTL` refer to these groups, enforcing that composite arguments belong to predefined sets of constructs—which may include both *primitives* and *composites*. This structured use of *primitives*, *composites*, and *groups* enables modular, extensible, and constraint-aware specifications. The translation pipeline validates the JSON, generates blocks and toolboxes, and defines code

generators—faithfully encoding the logic's structure while supporting incremental updates.

Because users build formulas via structured blocks, precedence and associativity rules are unnecessary: operation order is enforced visually, simplifying both the specification and the generated editor.

Complete documentation, sample logic definitions, and source code are available in our GitHub repository[3].

3 Case study

To illustrate our framework's flexibility, we present a case study on ATL [1], a modal logic for verifying strategic behaviour in multi-agent systems. ATL's compositional syntax and strategic modalities make it well-suited for visual representation.

The syntax of ATL, following [1], is defined as:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle A \rangle\rangle \mathtt{X}\varphi \mid \langle\langle A \rangle\rangle \mathtt{G}\varphi \mid \langle\langle A \rangle\rangle \varphi \mathtt{U} \varphi$$

where p is an atomic proposition, A represents a coalition of agents, and the remaining constructs are standard Boolean and temporal operators.

3.1 Specification of ATL Constructs

The specification of ATL in our framework begins by identifying its syntactic building blocks. These are organised into three categories: **Primitives**, such as `atom` (representing atomic propositions) and `agent` (denoting agent identifiers), which do not take subcomponents. **Composites**, representing operators such as `NOT`, `AND`, `X`, `G`, and `U`, which define terms composed from one or more arguments. **Groups**, which are auxiliary constructs used to define reusable collections of constructs. They help express argument constraints compactly and consistently across different composite definitions.

A group definition for ATL might appear as follows:

Listing 1.4. Group definitions for ATL constructs.

```
{
    "type": "group",
    "name": "all",
    "elements": ["atom", "NOT", "AND", "X", "G", "U", "agent"]
}
{
    "type": "group",
    "name": "all_but_agents",
    "elements": ["atom", "NOT", "AND", "X", "G", "U"]
}
```

Primitive constructs are specified in a straightforward manner as well.

[3] https://github.com/AngeloFerrando/Logic2Blockly

Listing 1.5. Atomic proposition primitive.

```
{
    "type": "primitive",
    "name": "atom",
    "format": "%string"
}
```

Listing 1.6. Agent primitive with stacking.

```
{
    "type": "primitive",
    "name": "agent",
    "format": "%string",
    "connection": ["agent"]
}
```

By default, primitive blocks are atomic and not directly composable. However, some constructs—such as ATL's strategic modality $\langle\langle A \rangle\rangle$, where A is a set of agents—require collections of primitives. To support this, our specification includes an optional `connection` field, allowing compatible blocks (e.g., `agent`) to be stacked vertically, forming unordered sets as needed.

A typical composite operator in ATL, such as $\langle\langle A \rangle\rangle$ `G` (*globally*), can then be defined with structured placeholders that reference groups:

Listing 1.7. Composite definition for ATL `G` operator with agent context.

```
{
    "type": "composite",
    "name": "G",
    "format": "<<%agent>> G %all_but_agents"
}
```

This format indicates that the $\langle\langle \Box \rangle\rangle$ `G` $\Box$ operator expects two arguments (denoted as $\Box$): a stackable collection of `agent` blocks and a subformula from the `all_but_agents` group. This structure enforces correct composition without manual constraint handling, and can be generalised to other strategic or nested constructs.

For brevity, we have presented only a subset of ATL constructs. Additional operators follow the same specification pattern and can be included with minimal extensions (the full ATL syntax is available on our GitHub repository).

3.2 Automatically Generated Editor

From the ATL specification described above, the framework automatically generates all the required visual artifacts. The resulting editor offers the following features: **Drag-and-drop blocks** for all ATL constructs, each with enforced input constraints derived from the specification. A **structured toolbox** organised according to the defined constructs (e.g., Boolean operators, temporal operators, agents). **Context-aware construction**: blocks accept only syntactically valid children, ensuring correctness by design.

Strategic contexts—such as the agent binding in $\langle\langle A \rangle\rangle$ `G` φ—are handled visually within the block structure. For instance, a user may instantiate a `G` block, which embeds an agent selector and a placeholder for a valid subformula. This structure enforces scoping and composition constraints interactively, preventing the creation of ill-formed expressions. Figure 2 shows a screenshot of an ATL formula constructed using the generated blocks. The expression

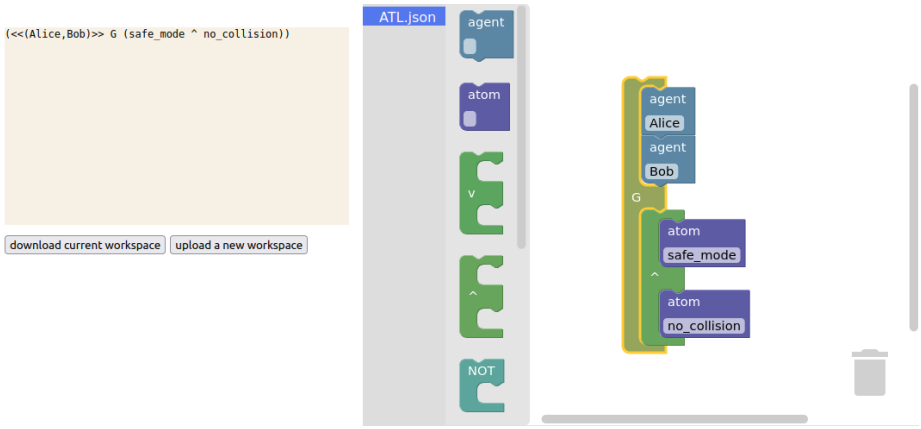

Fig. 2. Generated GUI in use: $\langle\langle Alice, Bob\rangle\rangle$ `G` (`safe_mode`$\wedge$`no_collision`) is composed via blocks.

$\langle\langle Alice, Bob\rangle\rangle$ `G` (`safe_mode` $\wedge$ `no_collision`) states that the coalition of agents Alice and Bob can ensure the system always remains in a safe state with no collisions. This scenario could represent a robotic or safety-critical environment involving movement and spatial coordination. It illustrates how users can visually compose strategic temporal properties using the block-based editor[4].

4 Comparison with Related Tools

Several tools have introduced visual editors to improve the accessibility of formal specifications, especially for educational use or domain-specific applications. These tools typically provide pre-defined constructs tailored to particular logics, with interfaces designed for non-expert users. In contrast, our framework prioritises generality and extensibility: logic syntax is specified declaratively, and new visual editors are generated automatically. Table 1 compares representative systems along design and usage dimensions, highlighting their respective strengths and target contexts.

Table 1. Comparison of visual tools for logic specification.

Reference	Interface	Logic	Users	Extensible	Domain
Ours	Blockly	Structured Spec.	Experts & Non-experts	Declarative	General
Grobelna [9]	Scratch-style GUI	LTL/CTL Templates	Non-experts	Fixed Operators	Control Systems
Nergaard et al. [14]	Scratch-style GUI	XACML Rules	Non-experts	Fixed Templates	Access Control
Omar et al. [16]	Structured Editor	Natural Deduction	Non-experts	Proof-based	Classical Logic

While prior tools clearly demonstrate the benefits of visual metaphors for improving accessibility, they are generally tailored to specific logics or application

[4] Full demonstration video: https://tinyurl.com/Logic2Blockly

domains. For instance, Grobelna [9] provides a Scratch-based editor for expressing temporal logic requirements in control systems, while Nergaard et al. [14] focus on authoring XACML access control policies via a rule-building interface. Omar et al. [16] support step-by-step construction of natural deduction proofs for educational purposes. These tools are well-suited to their respective contexts but rely on fixed sets of constructs and lack general extensibility.

In contrast, our framework is logic-agnostic and auto-generative: experts define syntax declaratively in JSON, from which a full block-based editor is automatically produced. This enables constraint-aware visual editors for both educational and practical use, without manual interface design.

5 Conclusions and Future Work

We presented a general, extensible framework for automatically generating block-based visual editors from structured JSON logic specifications. It lets experts define new logics without coding effort and enables non-experts to construct formulas in syntax-aware visual environments.

The framework has been successfully applied to a range of logics, including CTL [2], ATL [1], NatATL [11], RB-ATL [15], and Strategy Logic [13], demonstrating both its flexibility and reusability.

While our framework enables non-experts to build formulas without worrying about syntax errors, it does not eliminate the need for a basic understanding of logical semantics. Accessibility here should be understood as syntactic support: the editor lowers barriers to entry once a user is introduced to the concepts, but it cannot replace teaching materials or domain training. Moreover, usability for non-experts depends on experts providing the initial JSON specification. This dependency is inherent to our approach, as the framework is designed to separate the tasks of logic definition (by experts) and formula composition (by end users).

Future work includes integrating with verification backends, particularly the VITAMIN framework [6,7,8], which supports accessible modelling for non-experts. We also plan to extend the specification language, explore educational use cases, and improve editor usability.

A limitation of the current work is that JSON specifications have so far been authored only by the tool developers. Our experience suggests that the effort is modest—similar to transcribing a logic's grammar into a structured format—but we acknowledge that non-developers may encounter difficulties or errors when writing specifications. The extent to which this overhead deters use has not yet been systematically evaluated. Addressing this requires empirical studies with external experts and user feedback on the specification process.

References

1. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM **49**(5), 672–713 (2002). https://doi.org/10.1145/585265.585270, https://doi.org/10.1145/585265.585270

2. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981. Lecture Notes in Computer Science, vol. 131, pp. 52–71. Springer (1981). https://doi.org/10.1007/BFB0025774, https://doi.org/10.1007/BFb0025774
3. Crockford, D.: The application/json media type for javascript object notation (json). RFC 4627 (2006), https://www.rfc-editor.org/rfc/rfc4627
4. Czepa, C., Zdun, U.: How understandable are pattern-based behavioral constraints for novice software designers? ACM Trans. Softw. Eng. Methodol. **28**(2) (Feb 2019). https://doi.org/10.1145/3306608, https://doi.org/10.1145/3306608
5. Davis, J.A., Clark, M.A., Cofer, D.D., Fifarek, A., Hinchman, J., Hoffman, J.A., Hulbert, B.W., Miller, S.P., Wagner, L.G.: Study on the barriers to the industrial adoption of formal methods. In: Pecheur, C., Dierkes, M. (eds.) Formal Methods for Industrial Critical Systems - 18th International Workshop, FMICS 2013, Madrid, Spain, September 23-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8187, pp. 63–77. Springer (2013). https://doi.org/10.1007/978-3-642-41010-9_5, https://doi.org/10.1007/978-3-642-41010-9_5
6. Ferrando, A., Malvone, V.: Hands-on VITAMIN: A compositional tool for model checking of multi-agent systems. In: Alderighi, M., Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S. (eds.) Proceedings of the 25th Workshop "From Objects to Agents", Bard (Aosta), Italy, July 8-10, 2024. CEUR Workshop Proceedings, vol. 3735, pp. 148–160. CEUR-WS.org (2024), https://ceur-ws.org/Vol-3735/paper_12.pdf
7. Ferrando, A., Malvone, V.: VITAMIN: A compositional framework for model checking of multi-agent systems. In: Rocha, A.P., Steels, L., van den Herik, H.J. (eds.) Proceedings of the 17th International Conference on Agents and Artificial Intelligence, ICAART 2025 - Volume 1, Porto, Portugal, February 23-25, 2025. pp. 648–655. SCITEPRESS (2025). https://doi.org/10.5220/0013349600003890, https://doi.org/10.5220/0013349600003890
8. Ferrando, A., Malvone, V.: Vitamin: Verification of a multi agent system. In: Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems. p. 3023–3025. AAMAS '25, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2025)
9. Grobelna, I.: Scratch-based user-friendly requirements definition for formal verification of control systems. Informatics Educ. **19**(2), 223–238 (2020). https://doi.org/10.15388/INFEDU.2020.11, https://doi.org/10.15388/infedu.2020.11
10. Harvey, B., Mönig, J.: Snap! a visual, drag-and-drop programming language. https://snap.berkeley.edu/ (2013), accessed September 2025
11. Jamroga, W., Malvone, V., Murano, A.: Natural strategic ability. Artif. Intell. **277** (2019). https://doi.org/10.1016/J.ARTINT.2019.103170, https://doi.org/10.1016/j.artint.2019.103170
12. Mansoor, N., Bagheri, H., Kang, E., Sharif, B.: An empirical study assessing software modeling in alloy. In: 11th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE 2023, Melbourne, Australia, May 14-15, 2023. pp. 44–54. IEEE (2023). https://doi.org/10.1109/FORMALISE58978.2023.00013, https://doi.org/10.1109/FormaliSE58978.2023.00013
13. Mogavero, F., Murano, A., Perelli, G., Vardi, M.Y.: Reasoning about strategies: On the model-checking problem. ACM Trans. Comput. Log. **15**(4), 34:1–34:47 (2014)
14. Nergaard, H., Ulltveit-Moe, N., Gjøsæter, T.: Vispe: A graphical policy editor for XACML. In: Camp, O., Weippl, E.R., Bidan, C., Aïmeur, E. (eds.)

Information Systems Security and Privacy - First International Conference, ICISSP 2015, Angers, France, February 9-11, 2015, Revised Selected Papers. Communications in Computer and Information Science, vol. 576, pp. 107–121. Springer (2015). https://doi.org/10.1007/978-3-319-27668-7_7, https://doi.org/10.1007/978-3-319-27668-7_7

15. Nguyen, H.N., Alechina, N., Logan, B., Rakib, A.: Alternating-time temporal logic with resource bounds. J. Log. Comput. **28**(4), 631–663 (2018). https://doi.org/10.1093/LOGCOM/EXV034, https://doi.org/10.1093/logcom/exv034
16. Omar, C., Voysey, I., Chugh, R., Hammer, M.A.: Live functional programming with typed holes. Proc. ACM Program. Lang. **3**(POPL), 14:1–14:32 (2019). https://doi.org/10.1145/3290327, https://doi.org/10.1145/3290327
17. Pasternak, E., Fenichel, R., Marshall, A.N.: Tips for creating a block language with blockly. In: 2017 IEEE Blocks and Beyond Workshop (B&B). pp. 21–24 (2017). https://doi.org/10.1109/BLOCKS.2017.8120404
18. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
19. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: Practice and experience. ACM Comput. Surv. **41**(4), 19:1–19:36 (2009). https://doi.org/10.1145/1592434.1592436, https://doi.org/10.1145/1592434.1592436