Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Datateknik
2025 | LIU-IDA/LITH-EX-A--2025/116--SE

Automated generation of visual representations for CRML models
– CRML-VIS

Automatiserad generering av visuella representationer för CRML-modeller

Rajat Agarwal (Rajag969)

Supervisor : Adrian Pop
Examiner : Lena Buffoni

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Complex cyber-physical systems (CPS) accumulate multidisciplinary requirements that are often conflicting or hard to interpret when expressed only as text in the Common Requirement Modeling Language (CRML). This thesis presents *CRML-VIS*, a modular toolchain that automatically converts CRML specifications into clear, editable diagrams to improve stakeholder communication and help surface inconsistencies. The pipeline combines ANTLR-based parsing with listener-driven extraction into a lightweight model, deterministic layout, and emission of Draw.io (diagrams.net) XML; a command-line workflow produces a diagram for each CRML file.

A design-science methodology guided the selection and integration of visualization libraries. After a comparative assessment (Draw.io, Mermaid, PlantUML, JointJS, GoJS), Draw.io was chosen for the prototype due to its embeddable editor and straightforward XML format that enable end-to-end automation while preserving editability. The mapping from CRML constructs to visual elements uses sectioned class boxes, consistent color/shape coding, and orthogonal routing to remain readable for non-technical audiences.

CRML-VIS was evaluated on three representative models—Pumping System, Microgrid, and Smart Building—covering requirement rows, events, instances, and associations. The generated diagrams faithfully covered classes, requirements, instances, and links across scales without manual post-editing beyond export sizing. A small survey (n = 5 participants; 15 case responses) indicated improved comprehensibility and collaboration: understanding from code averaged 2.6/5 versus 4.4/5 from diagrams, 87% preferred diagrams for spotting conflicts, collaboration usefulness averaged 4.6/5, and 93% preferred using diagrams in reviews.

Contributions include a working prototype for automated CRML visualization and an extensible architecture that separates grammar, model, layout, and emitters. Limitations include lack of performance benchmarking, single-row default layout, and no round-trip editing; future work targets multi-row/paginated layouts, additional emitters, and a web application with embedded editing.

Keywords: CRML, requirements engineering, CPS, visualization, SysML-inspired diagrams, ANTLR, Draw.io, diagrams.net.

Acknowledgments

I would like to express my gratitude to Linköping University and the Department of Computer and Information Science (IDA) for the opportunity to carry out my master's thesis within their program. This project provided valuable experience in both research and applied software development.

I am sincerely thankful to my supervisor, Adrian Pop, for his guidance, constructive feedback, and continuous support throughout the work. His insight and encouragement were essential to the progress and completion of this thesis. I would also like to thank Lena Buffoni for serving as examiner and for her continuous input on both the technical aspects and the evaluation of the developed tool, which greatly contributed to improving and refining the final outcome.

I also wish to thank several friends from my program who shared ideas during discussions and contributed to the evaluation by participating in the survey. Their feedback helped shape the interpretation of the results and validate the usefulness of the proposed approach.

Contents

List of Figures

List of Tables

1 Introduction

This chapter introduces the research on automated visualization of CRML models for cyber-physical systems, describing the motivation, aim, research questions, approach, contributions, and delimitations of the study.

1.1 Motivation

Complex cyber-physical systems (CPS), particularly in the energy sector, involve multiple stakeholders who express requirements from diverse perspectives [2]. These requirements often result in a global set of constraints that can be conflicting or contradictory, leading to misunderstandings and inefficiencies in system design [2]. Clear communication and comprehensive visualization of requirements are essential to enhance understanding, facilitate conflict resolution, and streamline the design process.

The Common Requirement Modeling Language (CRML) enables formal expression of requirements as multidisciplinary spatiotemporal constraints that can be tested against system behavioral models to verify whether the requirements are met [2]. However, manually creating visual representations of CRML models is time-consuming, error-prone, and impractical for large-scale systems. Automated visualization tools inspired by SysML (Systems Modeling Language) can address these challenges by generating clear, consistent, and comprehensible diagrams, thereby improving collaboration and system design efficiency.

1.2 Aim

The primary objective of this thesis is to investigate and develop an automated toolchain for generating graphical representations of Common Requirement Modeling Language (CRML) models. It is planned to leverage the principles and techniques inspired by the SysML (Systems Modeling Language) to ensure consistency, clarity, and comprehensibility in the representation of multidisciplinary spatiotemporal constraints.

The aim is to enhance communication, facilitate clear requirement communication, and detect conflicts in system constraints. This will promote a more efficient collaborative design process for CPS among different stakeholders.

1.3 Research questions

The following research questions guide the thesis:

1. In what ways can visual representations be effectively employed to recognize and mitigate conflicting or contradictory requirements from diverse stakeholders?
2. What existing techniques and tools can be utilized and integrated to automate the visualization of CRML models?
3. Which visualization methods and libraries best support collaborative development, allowing multiple users to work on the same CRML model file, while ensuring clarity, comprehensibility, and ease of automation?
4. What architectural approaches or methods are the most suitable for developing a modular and extensible toolchain that allows easy plugin additions to generate diverse diagram types for CRML models?
5. How can the developed visualization toolchain be designed to support integration into a web-based application, to enhance stakeholder collaboration and accessibility?

1.4 Approach

This thesis adopts a design science research approach [3] to develop a prototype toolchain for automated visualization of CRML models. The methodology includes the following steps:

- A literature review of related work on visualization techniques for requirement modeling languages, such as SysML, is conducted to identify best practices and approaches for generating graphical representations of CRML models.
- A qualitative comparison of existing technologies, libraries, and tools (e.g., Draw.io, Mermaid.js, PlantUML, JointJS, GoJS) is performed, focusing on their ability to meet requirements such as clarity, comprehensibility, editability, extensibility, and ease of automation.
- Based on the comparison, the most suitable tools, algorithm, and design method for generating automated visualizations are selected, prioritizing modularity and extensibility to support diverse diagram types.
- A prototype toolchain is developed using JavaScript, implementing the selected design method to generate clear, comprehensible, and editable visualizations of CRML models, with a base for future features like collaborative editing and other visualization diagrams.
- Results will be judged primarily on *correctness* (one-to-one coverage and faithful mapping of CRML constructs in the generated diagrams) and *visual quality* (readability, consistent notation, low clutter).

1.5 Contributions

This thesis provides the following contributions.

- A prototype toolchain for automated generation of graphical representations of CRML models, designed to produce clear, comprehensible, and editable visualizations, with the potential for integration into web applications in the future.

- A modular and extensible architecture for the visualization toolchain, enabling future additions of new diagram types with minimal code changes.
- Recommendations for integrating the toolchain into a web-based application to improve stakeholder accessibility and collaboration, along with suggestions for future enhancements to improve the completeness of the toolchain.

1.6 Delimitations

This thesis focuses on developing an automated visualization toolchain for CRML models, inspired by SysML, to support stakeholder collaboration and conflict resolution in CPS design. Only an automated toolchain, not a complete web application, will be created due to time and resource constraints; however, the toolchain is designed to serve as a component of a web application if developed in the future. While the toolchain is designed to be extensible, the prototype will initially come with a single diagram type, prioritizing clarity and comprehensibility over variety. However, due to modularity in design, plugging in other diagram types has been made convenient. The evaluation of the prototype concentrates on the *correctness* and *visual clarity* of generated diagrams; detailed performance benchmarking (speed/memory) and formal user studies are explicitly excluded for this prototype, based on the nature of this thesis.

2 Background and Related Work

2.1 Background

2.1.1 Requirements Engineering in Complex Cyber-Physical Systems (CPS)

Requirements Engineering (RE) is the discipline of eliciting, specifying, and managing what a system must accomplish. A requirement is commonly defined as a condition or capability that a system must satisfy to fulfill a contract, standard, or stakeholder need [5]. In practice, requirements for complex systems are often documented in natural language [5]. Natural language is intuitive for stakeholders but tends to be ambiguous and imprecise [5]. This ambiguity can lead to misunderstandings and inconsistent implementations of the system's intended behavior [5]. These issues are amplified in complex cyber-physical systems (CPS), which involve hardware, software, and multiple engineering domains.

In CPS projects, numerous stakeholders from different domains (mechanical, electrical, software, etc.) each contribute requirements from their own perspectives. The combined requirements set can become conflicting or even contradictory, especially if past design decisions and constraints are not fully documented [2]. New requirements added without revisiting legacy assumptions lead to over-specifications, delays, and cost overruns [2]. Achieving a common understanding among stakeholders is thus a major challenge in CPS RE. Moreover, CPS requirements often pertain to temporal behavior, physical constraints, and probabilistic criteria (e.g. reliability or safety thresholds) that are hard to capture with simple static statements [2]. Ensuring that all these complex constraints are realistic, understandable, and verifiable by different stakeholders demands a more formal and systematic approach [2]. Another challenge is the gap between informal requirements and formal design models. Industry tools for requirements (like IBM Rational DOORS or Siemens Polarion) typically record requirements in natural language form [2]. At the other extreme, formal methods exist (e.g. Linear Temporal Logic – LTL, computation tree logic, statecharts) to specify precise behaviors, but these often require abstracting the system into states or mathematical assertions [2]. Such formalisms can be too rigid or complex for day-to-day use by engineers, especially across an entire CPS [2]. For instance, a temporal logic formula can prove that "the system will eventually reach state X", but real CPS may not have a fixed, finite set of states due to continuous wear or unexpected events [2]. Additionally, formal specification languages lack easy mechanisms to tie requirements directly to system architecture components (i.e. they lack object-orientation) [2]. These factors explain why formal requirement languages are usually applied only to small, critical subsystems, while broad CPS requirements remain handled in informal ways[2]. In summary, CPS projects need methods to formalize requirements for rigor and verification, yet remain understandable and relatable to all stakeholders throughout the system lifecycle.

2.1.2 Model-Driven Engineering (MDE) and Requirements

Model-Driven Engineering (MDE) is a development paradigm that addresses system complexity by using high-level models as the primary artifacts throughout the lifecycle. Instead of treating requirements, design, and code as separate silos, MDE promotes transformations between modeling levels – for example, automatically generating design models or code from abstract requirement models. In the context of RE, Model-Driven Requirements Engineering applies MDE principles to requirements: requirements are specified in a structured model form, enabling consistency checks and automatic propagation of changes into downstream design models. This approach helps bridge the gap between what stakeholders want and the system design that satisfies those needs [17]. By keeping requirements in a formal model, one can leverage tools to validate requirements, perform simulations, or even generate portions of a solution, thereby ensuring the system's implementation continually aligns with its intended requirements.

Several industry methodologies fall under MDE. For instance, Model-Based Systems Engineering (MBSE) is an application of MDE in systems engineering that uses formal models to represent requirements, architecture, behavior, and V&V (Verification & Validation) information in an integrated way [17]. These models serve as a single source of truth, improving traceability from high-level stakeholder goals down to technical specifications. A key enabler of MDE/MBSE is the availability of standardized modeling languages and profiles, which provide the syntax and semantics to represent different aspects of systems. Two relevant standards in this regard are SysML and MARTE, which extend the general-purpose modeling language UML for specialized needs.

2.1.3 SysML and MARTE Modeling Standards

The Systems Modeling Language (SysML) is a general-purpose modeling language defined as a UML profile tailored for systems engineering. SysML introduces a Requirements Diagram among other diagrams, explicitly to capture textual requirements as first-class model elements. Each requirement is modeled with properties (ID, text, etc.), and crucially, SysML defines relationships to connect requirements with other model elements. For example, requirements can be organized hierarchically, they can derive new requirements, or be refined by more detailed requirements. SysML also provides satisfy links (to show which design components fulfill a requirement) and verify links (to connect test cases or verification procedures to requirements), as well as generic trace links [17]. These relationships help maintain traceability – a web of associations ensuring that every high-level need is addressed by some design element and validated by some test [17]. Under the hood, SysML's requirements modeling constructs are implemented as stereotypes of a UML Class (with the text of the requirement as an attribute) [17]. This means SysML requirements can seamlessly coexist with UML/SysML design models in a unified repository, supporting consistency checks and impact analysis (e.g., if a requirement changes, one can see which design elements and tests are affected).

While SysML covers the general needs of systems engineering, it does not natively include specialized semantics for real-time and embedded domain concerns (e.g., precise timing, scheduling, performance metrics). The MARTE profile (Modeling and Analysis of Real-Time and Embedded systems) fills this gap. MARTE is an OMG standard profile that extends UML with concepts for modeling real-time scheduling, hardware resources, clock timing, and continuous time behaviors. Engineers can annotate models with MARTE stereotypes to specify, for example, that a task has a deadline or a data flow has a certain latency. By combining SysML and MARTE, one can formalize non-functional requirements (like timing constraints or throughput) alongside functional requirements. For instance, Ribeiro and Soares demonstrate using SysML with MARTE stereotypes to model and trace software requirements in a real-time traffic control system [16] . In their approach, SysML captures the functional structure (e.g., actors and use cases for context, requirement blocks for each need) and MARTE annotations express the real-time properties each requirement must meet [16]. This combination enables requirements models that are not only traceable to design, but also carry the quantitative detail needed for analysis (such as schedulability or performance analysis). However, prior studies have noted challenges in integrating SysML and MARTE. Because they originate from different concerns (systems architecture vs. time analysis), aligning their semantics can be non-trivial [16]. Still, successful industrial applications exist – for example, experience reports have shown the application of UML/MARTE to real projects, revealing both the benefits and practical difficulties of such formal modeling in industry [16]. These works reinforce that combining modeling standards can formalize requirements and improve rigor, but also highlight the need for better tool support to manage complexity.

2.2 Related Work

To contextualize the contribution of CRML-VIS, this section surveys related research and tools in four thematic areas: (1) automating model creation from natural language requirements, (2) using SysML/MARTE to formalize and visualize requirements, (3) qualities of visualization tools and frameworks for automated modeling, and (4) multimedia or prototype-based requirements engineering tools. Each theme highlights how researchers have approached requirement modeling and visualization, providing a basis to compare and contrast with our solution.

2.2.1 Natural-Language to UML Automation

One significant thread of research has tried to bridge the gap between informal textual requirements and formal models by automatic translation. The idea is to let stakeholders continue writing requirements in natural language (NL) while a tool analyzes the text and produces structured diagrams, usually in UML. Chen and Zeng's work [5] is a prominent example: they propose a novel two-step approach to transform NL product requirements into UML diagrams. First, free-form requirement text is parsed into an intermediate graphical model called the Recursive Object Model (ROM), which captures the semantic relationships implied by the text [5] . Next, a set of generation rules is applied to convert the ROM elements into target UML diagrams (specifically, use case and class diagrams) [5]. The authors implemented a prototype tool called R2U 1.0 following this approach, which automatically generates the use case and class diagram for a given requirement description [5]. The transformation relies on understanding basic sentence structure and domain terms, using ROM as a bridge to mitigate the ambiguity of raw text. This approach showed feasibility in a case study and underscored the potential of AI and NLP techniques in requirements modeling [5].

Other researchers have explored similar NL-to-model automation. For example, early efforts used controlled English or semantic analysis to extract key entities and actions for class or sequence diagrams [5]. A challenge noted across these approaches is handling unrestricted natural language input – without constraints, NL can be too vague or complex for reliable parsing [5]. The ROM-based method by Chen and Zeng stands out by explicitly tackling semantics: by deriving a formal semantic model (the ROM) of the requirement text, their tool can generate consistent UML elements even from somewhat complex sentences [5]. In comparison, approaches lacking a semantic intermediary struggled beyond very simple or controlled inputs [5].

The motivation behind these NL-to-UML tools is clear: they aim to reduce the manual effort and error in creating initial draft models from stakeholder documents. Importantly, Chen and Zeng do not propose to remove the human analyst from the loop. On the contrary, they envision automation as an aid to improve understanding – the automatically generated diagrams can help engineers validate and refine requirements with stakeholders [5]. By automating repetitive translation tasks, the human experts can focus on disambiguation and higher-level analysis. The end goal is to increase the consistency of the final models (since the same rules are applied to all requirements) and to ensure no important detail in the text is overlooked [5]. The success of such approaches is often measured in terms of correctness of generated models and the improvement in requirements quality (e.g., detection of ambiguities or missing elements). While these techniques have shown great potential in requirements modeling [5], they typically target software requirements and UML diagrams only, without addressing cross-domain CPS concerns. CRML-VIS differs in that it operates on an already formalized requirements language (CRML) rather than raw NL, but it shares the spirit of automation – generating visual models systematically to aid comprehension.

2.2.2 Formalized Requirement Visualization with SysML and MARTE

A different line of work starts from the modeling side of the spectrum, assuming requirements can be captured in a formal or semi-formal model from the outset. Here, the emphasis is on using standard modeling languages (like SysML/UML and profiles such as MARTE) to represent requirements and then leveraging those models for visualization and analysis. SysML, as introduced in the background, provides a graphical syntax for requirements which has been applied in model-driven approaches to RE. Soares and Vrancken (2008) proposed a model-driven requirements engineering approach using SysML to specify user requirements in a structured way [17]. In their work, each atomic requirement is classified and represented as an element in a SysML Requirements diagram, which graphically shows the requirement text and its relationships to other requirements and use cases [17]. By structuring requirements in a requirements diagram (and complementing it with tabular views for traceability), they demonstrate that requirements can be gradually refined and linked to system use cases and design elements within the same modeling environment [17]. The SysML model becomes a living artifact: after being captured graphically, high-level user requirements can be detailed into more concrete system requirements, maintaining trace links between levels [17]. This approach shows the benefit of hierarchical modeling – one can navigate from a top-level need down to specific technical requirements, all within SysML diagrams.

Building on SysML's foundation, researchers have integrated MARTE to address temporal and performance aspects in requirements models. Ribeiro and Soares (2013) illustrate how combining SysML with MARTE stereotypes allows formalizing real-time constraints at the requirement level [16]. In their case study (urban traffic control software), they modeled each requirement in SysML and used MARTE to annotate, for example, timing deadlines or periodicity for certain functional requirements [16]. This enabled traceability of both functional and non-functional requirements within a unified model. Not only could they trace which component satisfied a functional requirement, but they could also check if timing requirements were met by the design through MARTE's analysis capabilities. Such approaches essentially treat requirements modeling as an extension of system modeling – often termed Model-Based Requirements Engineering (MBRE) [17]. The models serve multiple purposes: documentation (visual diagrams for human communication), analysis (via formal profiles or constraints), and direct transition into design or test models (since requirements can link to use case or test case models as in SysML's satisfy/verify relationships).

Helming et al. (2010) [8] argued "towards a unified requirements modeling language" that would standardize how requirements are visualized and linked to development artifacts [16]. Their vision was to reduce the fragmentation where each organization or tool uses its own notation – instead providing a common modeling language for requirements akin to UML for design. CRML can be seen as a realization of this vision: it is proposed as a common language for formal requirements across disciplines [2]. Another gap is that while SysML and MARTE support formal notation, they were not originally designed for direct simulation of requirements. The approaches above focused on modeling for documentation and traceability, but verification often stops at analysis or code/test generation. In contrast, CRML introduces executability into requirements modeling (by co-simulating requirements models with system models) [2], which is beyond the scope of SysML 1.x and MARTE alone. That said, the upcoming SysML v2 standard is moving towards more executable models and may better complement such needs [2]. For now, CRML's tight integration with simulation distinguishes it from prior SysML/MARTE-based RE visualization efforts.

2.2.3 Visualization Tool Qualities and Automated Modeling Frameworks

Across both natural-language-driven and model-driven approaches, researchers have identified several desirable qualities for tools that visualize requirements. First, a requirements visualization tool should improve communication and understanding among stakeholders. This is evident in works like Chen & Zeng's, where the generated UML diagrams are intended to clarify the requirements and reduce ambiguity [5]. Similarly, Soares's SysML approach emphasized understandability by presenting requirements in both graphical and tabular formats, making it easier to trace and discuss them in reviews [17]. A second quality is consistency and accuracy of models. Automation can help here: when a framework automatically generates a diagram or model from a single source of truth (be it a text document or a formal spec), it eliminates discrepancies that often creep in when diagrams are drawn manually. Chen & Zeng note that their automatic generation method increases the consistency of UML diagrams, especially in large projects with multiple engineers contributing [5]. In other words, an automated framework enforces uniform interpretation of requirements via standard generation rules, acting as a safeguard against human misinterpretation.

Another important aspect is traceability and integration. Good requirements visualization tools do not operate in isolation – they integrate with the broader development process. For instance, Helming et al.'s [8] unified modeling environment (as part of the UNICASE platform) was geared towards linking requirements with other project artifacts in one tool [16]. Likewise, Iqbal et al. (2011) [10] developed a framework to generate test code from UML/MARTE models [16], which implicitly requires that the requirements modeled in UML are precise enough to drive downstream automation. Their follow-up industrial experience report (2012) [11] highlights that using UML/MARTE in real projects helped catch inconsistencies early and supported automated testing [16]. These examples show that a valuable quality of RE visualization frameworks is the ability to connect with automated modeling and analysis workflows – whether generating code, test cases, or performing simulations. Automation in this sense is not just about drawing diagrams, but about ensuring the models are alive and useful for verification and validation activities.

In terms of frameworks, many draw on metamodeling and transformation engines typical of MDE. Tools like IBM Rhapsody or Enterprise Architect allow defining custom profiles or using scripting to generate views from requirement data. In research prototypes, it is common to use transformation languages or custom code: for example, the Chen and Zeng's R2U tool encodes generation rules to transform a semantic model (ROM) into UML elements [5]. CRML-VIS follows a similar philosophy by implementing transformations (with ANTLR and draw.io XML generation, as described earlier) to automate the creation of diagrams from CRML models. The underlying framework qualities here are extensibility and standardization – using standard formats (like UML, SysML, or draw.io) ensures that the output can be extended or edited using widely available tools, avoiding vendor lock-in. Moreover, by aligning with standards, the visualizations produced by automated frameworks are more easily understood by others (for instance, an engineer familiar with SysML will readily interpret a CRML-VIS generated diagram if it uses analogous notation for requirements).

Finally, an often-cited quality is maintaining human control and feedback in the loop. Automated modeling frameworks should allow users to adjust or refine outputs. Chen & Zeng (2009) explicitly mention that their approach "does not intend to exclude the human users from the loop" [5] – the tool is there to assist, but engineers are expected to review and update the models as needed. This is critical for practitioner acceptance: a highly automatic tool that stakeholders do not trust or find too rigid will not be used in practice. Thus, the best approaches strike a balance: they automate the laborious tasks (parsing, initial diagram layout, consistency checking) while letting users exercise judgment on requirements content and final presentation tweaks. CRML-VIS is designed with this balance in mind, generating a complete initial visualization that users can then inspect via the familiar draw.io interface, aligning with the principle that automation should enhance, not replace, human insight in RE.

2.2.4 Multimedia and Prototype-Based Requirement Engineering Tools

Beyond formal models and textual specifications, there is a stream of research exploring multimedia and prototypes to support requirements engineering. The rationale is that certain requirements (especially those related to user experience or complex interactions) can be better communicated with visual or interactive media rather than static text. One illustrative work is by Chen & Zeng (2002), who developed a Visual Requirement Authoring Tool (VRAT) to create animated representations of requirements scenarios [4]. In their approach, a textual scenario (for example, a sequence of steps in a banking transaction) is converted into a visual "event diagram" animation. The animated scenario shows the system's behavior step-by-step, with each event (e.g., message sent, data updated) presented using graphics and simple multimedia effects [4]. The intention is to make the requirement scenario easier to grasp, especially for non-technical stakeholders, by literally showing how the process unfolds rather than only describing it in words.

Chen & Zeng evaluated their visual tool by conducting an experiment with graduate students, comparing the traditional textual representation of a requirement scenario to the new visualized form [4]. They measured factors like the time it took to create each representation, the completeness of problem expression, the ease of communication between users and developers, and the understandability of the requirement [4]. The findings suggested notable improvements in communication and understanding when using the visual approach. In fact, most participants (acting as either system customers or analysts in the study) showed a willingness to adopt the visualized requirement representation over the conventional textual form [4]. A large majority also responded "yes" to the question of whether using the visual representation would help in obtaining correct software requirements [4]. These results underline the value of multimedia in RE: visual or interactive depictions of scenarios can bridge language gaps, uncover hidden assumptions, and engage stakeholders more effectively. The VRAT study also identified some trade-offs – for instance, creating the visual scenario initially took more effort than writing it textually, and one must ensure that the visual depiction remains synchronized with any changes in textual requirements [4]. Nonetheless, the concept of a visual requirements specification has since influenced numerous prototype-based RE tools, such as those that allow drawing UI mockups or storyboards and linking them to requirements. Modern Agile practices similarly encourage low-fidelity prototypes or simulations early in development to validate requirements with users.

Beyond animations, other multimedia approaches include using video recordings of user workflows to derive requirements (as in ethnography by video studies) [4], or employing AR/VR to let stakeholders experience a part of the system before it is built, then capturing their feedback as refined requirements. The common thread is enhancing stakeholder communication and requirement validation through something more tangible than pure text. These approaches typically supplement rather than replace formal requirement models – for example, a prototype UI might be linked to formal requirements for the functions it demonstrates. In the scope of CRML-VIS, the focus is on automated diagram generation, not on creating interactive media; however, the tool inherits the same core goal of improving understandability. By producing clear diagrams (which are a visual medium) from formal CRML specifications, CRML-VIS plays a similar role to the above tools: it helps stakeholders see the structure of requirements (and their relations to each other or to system elements) rather than having to read purely textual logic or code. This is crucial when the requirements are formal and complex – a well-designed diagram can expose the big picture and critical details in a more accessible form, much like an animated scenario helps convey a complex interaction.

2.2.5 Summary and Gap Analysis

In surveying related work, we observe that numerous tools and methods exist to visualize and formalize requirements, each addressing certain challenges. Natural-language processing approaches target the front-end of RE, translating stakeholder language into initial models, whereas SysML/MARTE modeling targets the back-end, formalizing requirements for traceability and analysis. Visualization-centric tools like VRAT focus on stakeholder communication through rich media. However, none of these prior works fully addresses the combination of needs that CRML and its visualizer aim to meet. Natural language tools do not enforce a common formal semantics (they work with whatever text is given), and SysML/MARTE approaches, while formal, did not provide an open standard specifically for requirements across domains – this is exactly the gap the Common Requirement Modeling Language is trying to fill [2] . CRML provides a unified, formal language for multidisciplinary requirements with executability, but by itself it is primarily text-based and mathematical [2] . Prior to our work, there was no dedicated visualization tool for CRML models, meaning stakeholders would have to interpret CRML specifications directly or use generic modeling tools with significant manual effort. CRML-VIS fills this gap by automatically generating diagrams from CRML models, thus marrying the advantages of formal specificity with those of visual communication.

In contrast to earlier NL-to-UML solutions, CRML-VIS does not start from ambiguous natural language – it starts from the formal CRML specification, ensuring that the visualization is semantically unambiguous and directly tied to analyzable requirements. Compared to SysML/MARTE-based requirement modeling, CRML-VIS targets an emerging open standard (CRML) that is designed to integrate with both SysML v2 and model simulation environments [2]. This means our approach can depict requirement constructs (like spatiotemporal constraints with probabilistic criteria) that have no straightforward equivalent in vanilla UML/SysML or existing tools. Moreover, CRML-VIS is built with open-source integration in mind (e.g., using the open version of draw.io diagrams and aligning with OpenModelica workflows), whereas many past tools were proprietary or isolated prototypes. In summary, CRML-VIS extends the state of the art by providing the first automated visualization support for the CRML, enabling engineers to easily view, share, and discuss complex formal requirements models. It complements prior research by focusing on an area that was previously under-served – ensuring that formal requirements models (not just textual documents) can be visualized and understood with the ease of UML diagrams or storyboards, while retaining the rigor needed for verification. This unique contribution helps bridge the longstanding gap between highly formal requirement specifications and accessible stakeholder communication, thus facilitating a more model-driven and collaborative requirements engineering process for CPS.

3 Methodology

This section outlines the methodology followed to select an appropriate visualization library for the CRML-VIS project. The goal was to enable automated generation of graphical representations from CRML models. We conducted a comparative evaluation of five candidate libraries—**Draw.io (diagrams.net)**, **Mermaid.js**, **PlantUML**, **JointJS**, and **GoJS**—against a set of project-specific criteria and software quality characteristics. The evaluation process involved reviewing official documentation and repositories for each tool and mapping their features to our criteria. In alignment with the ISO/IEC 25010:2023 software quality model [9], we ensured a comprehensive assessment across multiple quality dimensions. The following subsections describe the selection criteria, the evaluation of each candidate tool, and the comparison results leading to the final choice of Draw.io.

3.0.1 Selection Criteria and Quality Attributes

Project-Specific Criteria: We identified five key criteria that the visualization library must satisfy for successful integration into CRML-VIS:

- **Open Source (Permissively Licensed):** The tool should be available under an Open Source license (preferably permissive like MIT or Apache) to avoid prohibitive costs or legal restrictions on integration. This ensures that we can use and potentially modify or extend the library within an academic or enterprise context without licensing barriers.
- **Ease of Use:** The library should be developer-friendly and, if applicable, user-friendly. This includes having a gentle learning curve, good documentation, and an intuitive API or interface to generate diagrams from CRML models. The high ease-of-use reduces development effort and errors.
- **Browser-Based Deployment Compatibility:** Since a future deployment of CRML-VIS is expected to be browser-based, the library must run smoothly in a web environment. Ideally, it should be a client-side web library or have a mode that can be embedded in web applications without heavy server-side components.
- **Team Collaboration Support:** Support for real-time or asynchronous collaboration on diagrams is desirable, as multiple stakeholders (e.g. developers, analysts) might co-

create or edit model diagrams. This could include multi-user editing features or at least easy sharing and version control of diagrams.

- **Plugin/Integration Friendliness (especially VS Code):** The tool should integrate well with our development workflow. In particular, availability of editor plugins (e.g., a Visual Studio Code extension) or APIs that allow integration into IDEs and other tools is a plus. This facilitates editing and viewing of diagrams directly alongside code, improving productivity.

ISO/IEC 25010:2023 Quality Characteristics: In addition to the above criteria, we evaluated each candidate against the relevant software quality characteristics defined in the ISO/IEC 25010:2023 standard [9]. This provided a structured way to assess the overall quality and risks of each library. The characteristics (and how we interpreted them for this selection) are listed below:

- **Functional Suitability:** The extent to which the library's functionalities meet the requirements of CRML-VIS. We examined if the tool can represent the needed model elements and relationships and support the required diagram types (e.g. flowcharts, UML-like diagrams). Automation capability (generating diagrams from model data programmatically) was a key aspect here.

- **Performance Efficiency:** The efficiency of rendering and updating diagrams, especially for large or complex CRML models. We considered reports of how each library performs with an increasing number of nodes and connections, as well as memory footprint and load time in a browser.

- **Compatibility:** The ability of the library to integrate with our target environment and with other tools. This included technical compatibility (e.g. can it be used with our technology stack, frameworks, and data formats) and compatibility with future extensions (for instance, does it use standard diagram formats that could be exchanged with other tools).

- **Interaction Capability (Usability):** The usability of the library's interface or API. For end-user-facing tools (like Draw.io), this meant an intuitive diagram editing UI; for code-focused tools (like Mermaid, PlantUML), it meant a straightforward syntax or API for developers. We also assessed the availability of documentation and community support as part of usability.

- **Reliability:** The maturity and stability of the library. We looked at the frequency of releases, known bugs or issues, and whether the tool is proven in production by a community. Reliability also covers error handling (e.g. how gracefully the library deals with invalid model data or network issues in a web context).

- **Security:** Security considerations of using the library. For a client-side library, this includes evaluating if it has any known vulnerabilities (especially important for web deployment). We also considered licensing security – whether using the library could impose legal risks (for example, viral open-source licenses).

- **Maintainability:** The ease of maintaining and evolving the visualization component. Key factors were code quality (if open source, is the codebase readable and modular?), presence of an active developer community or maintainers, and the ability to extend or fix the tool if needed for our project's purposes.

- **Flexibility (Portability):** The ability to port or adapt the library to different environments and requirements. We examined if the library runs on all modern browsers and operating systems, and whether it can be adapted (or configured) for different use cases.

Flexibility also includes how easily we can customize the library (e.g. add new diagram shapes or themes for CRML-specific notations).

- **Safety:** Although "Safety" in ISO 25010 typically relates to avoiding risks that could lead to harm, in our context it translates to avoiding any result from the tool that could mislead users or cause loss of critical data. We ensured that the libraries do not, for instance, corrupt model data or produce incorrect diagrams that could lead to bad decisions. Safety also encompasses the tool's reliability in preserving diagram information (so that model knowledge is not lost due to a tool failure).

By combining the project-specific criteria with the ISO 25010 quality model, we ensured both a targeted evaluation for CRML-VIS needs and a broad quality assessment. Next, we present the evaluation of each candidate library, referencing evidence from official sources to substantiate the assessment.

3.0.2 Candidate Library Evaluations

Draw.io (diagrams.net) – Evaluation

Overview: Draw.io (also known by its domain diagrams.net) is a diagramming application that can run as a web application or desktop app [13]. It provides a rich GUI for creating a wide variety of diagrams via drag-and-drop [13]. For our purposes, the key consideration was whether Draw.io can be automated or integrated to generate diagrams from CRML models.

Open Source License: Draw.io's situation regarding open source is somewhat unique. The project's code is published on GitHub and the distributed **minified** application is licensed under Apache 2.0, which is a permissive open license. However, the maintainers explicitly note that the actual source code needed to build the application is **not** fully open in that repository, and thus "this is not an open source project" in the traditional sense [13]. In other words, one can use the Draw.io compiled application freely under Apache 2.0 terms, but one cannot easily modify its core since the human-readable source is not provided. For our project, this means we can deploy and use Draw.io without licensing fees, but we have limited ability to alter its fundamental behavior. On the positive side, the Apache 2.0 license allows integration into our system without legal complications, and no copyleft requirements.

Ease of Use: As an end-user tool, Draw.io is known for its ease of use – it offers an intuitive graphical interface with a palette of shapes, and users can quickly create diagrams by dragging elements. For developers, Draw.io also offers an XML-based file format for diagrams, which means CRML-VIS could potentially generate a diagram by producing an XML file that Draw.io can read. The learning curve for basic use is very low for users (which is beneficial if modelers will directly use it), and for developers the main task would be understanding the diagram file format or using any integration API provided. Draw.io is widely used in industry and academia for diagramming, indicating that it is considered user-friendly and well-documented. Its documentation and community support are strong, with an extensive knowledge base on the diagrams.net website and active forums.

Browser Deployment: Draw.io was evaluated highly on this criterion. It is fundamentally a web application – the official deployment at **app.diagrams.net** runs entirely in the browser. Thus, it can be embedded into web pages or used as a client-side component. In fact, the developers provide an **embed mode** specifically for integration scenarios: Draw.io can run inside an `<iframe>` with which a host application communicates via postMessage API (HTML5 Messaging). This allows a web application (like our CRML-VIS front-end) to launch the Draw.io editor in a contained way, pass it a diagram (for example, generated from a CRML model), and receive the edited diagram data back [12]. All diagram editing happens client-side in the user's browser, meaning no server round-trips are required for diagram rendering. This is ideal for our browser-based deployment plans.

Collaboration Support: Out-of-the-box, the open Draw.io application **does not support real-time collaborative editing** [13]. The maintainers acknowledge that collaborative multi-user editing is not built into the base project, and they suggest considering other frameworks if that is a primary requirement. However, Draw.io does support asynchronous collaboration in the sense that diagrams can be stored in cloud services (Google Drive, OneDrive, etc.) where multiple users can access and edit them sequentially. In fact, in some integrations (such as the Nextcloud integration), Draw.io provides *shared cursors* for real-time collaboration when used in that environment [6]. This implies that while the core library does not natively have Google Docs-style live editing, it can be coupled with external real-time backends to enable multi-user scenarios. For our project, team collaboration was a desirable but secondary criterion; it is acceptable if collaboration is at the level of version control (each modeler editing diagrams and merging changes) rather than truly simultaneous editing. Draw.io meets this criterion sufficiently by allowing diagrams to be saved in formats that can be version-controlled and by integrations that enable sharing.

Plugin/VS Code Integration: Draw.io scores well here. There are third-party plugins for VS Code that embed the Draw.io editor within the IDE (for example, the "Draw.io Integration" extension for VS Code, which is unofficial but widely used). This extension allows developers to create and edit `.drawio` diagram files directly inside VS Code and save diagrams alongside code. While not officially developed by the Draw.io team, it is noteworthy that the official diagrams.net website has mentioned and welcomed such community extensions. In addition, Draw.io has official integrations with various platforms: for instance, there are plugins for Atlassian Confluence and Jira, integrations with Notion, MediaWiki, and others [6]. This indicates a healthy ecosystem for extending Draw.io into different environments. Specifically for VS Code, the availability of an editor plugin means our developers can visualize CRML diagrams without leaving their coding environment, which supports productivity and ease of adoption.

Quality Attribute Evaluation: Functionally, Draw.io is very suitable: it supports a broad range of diagram types and is flexible enough to represent custom notations (we can create custom shape libraries or use XML templates to represent CRML-specific symbols). It thus meets *Functional Suitability* well. In terms of *Performance*, Draw.io handles moderately complex diagrams in the browser efficiently; performance might degrade with extremely large diagrams (hundreds of elements), but given typical CRML model sizes, this is not expected to be an issue. Draw.io's *Compatibility* is strong: it works on all modern browsers without plugins (pure HTML5/JS app), and it can export diagrams to standard formats (PNG, SVG, XML) that ensure compatibility with other tools. For *Usability (Interaction Capability)*, end users find the interface intuitive, and our developers will find the integration API (embed mode) reasonably well-documented. The *Reliability* of Draw.io is evidenced by its widespread use and mature codebase; crashes or data loss are rarely reported in normal use. On *Security*, using Draw.io in an offline or controlled environment is safe (it does not require server interaction by default; all processing can be local). We will, however, be cautious of the fact that the source is not fully open, so we rely on the vendor for any critical security fixes. *Maintainability* of Draw.io from our perspective is mixed: we cannot modify the core easily (since source is closed), but we arguably may not need to — the tool can be extended via configurations, custom shape libraries, and plugins without altering the core code [6]. This plugin approach preserves maintainability by keeping our extensions separate. *Flexibility (Portability)* is very good: Draw.io runs on any OS (via browser or electron app) and can be integrated in various contexts (as evidenced by many integrations). Finally, *Safety* is addressed by Draw.io's stable editing environment (it auto-saves diagrams to prevent data loss and uses well-tested diagramming logic, minimizing the risk of producing incorrect or inconsistent diagrams).

In summary, Draw.io met most of our criteria strongly, with the only notable drawback being the lack of built-in real-time collaboration and its semi-open-source nature. Given its rich feature set, permissive usage license, and proven integrations, it emerged as a top candidate.

Mermaid.js – Evaluation

Overview: Mermaid.js is an open-source JavaScript library that generates diagrams from text descriptions. It uses a Markdown-inspired syntax to allow users to define graphs (like flowcharts, sequence diagrams, etc.) in a plain text block, which Mermaid then renders as an SVG or PNG diagram. It is often used in documentation systems and integrates with many static site generators and editors.

Open Source License: Mermaid is open source under the MIT License [15]. This permissive license means there are no restrictions on its use within our project. The project is community-driven and has won awards in the open-source arena, indicating active maintenance. Using Mermaid poses no licensing issues and fits well with an academic/open development model.

Ease of Use: For developers, Mermaid is relatively easy to use: one writes a textual specification of a diagram, and the library takes care of the layout and rendering. This is beneficial for automated diagram generation—our system could programmatically generate Mermaid syntax from a CRML model and let Mermaid draw it. The Mermaid syntax is considered quite user-friendly for those familiar with Markdown and code blocks; however, it does require learning the specific diagram syntax (for each diagram type). Compared to a WYSIWYG tool like Draw.io, Mermaid has a steeper learning curve for non-programmers. That said, Mermaid offers a live online editor and plenty of examples in its documentation, which aids in learning. In terms of integration, Mermaid can be used by simply including its JavaScript library in a web page and calling an API to render diagrams, which is straightforward. Its documentation site and community support are strong, contributing to its usability for developers.

Browser Deployment: Mermaid is designed inherently for the web. It runs entirely in the browser as a JavaScript library (or can be used server-side via Node.js if needed). It does not require any server component for rendering; the diagrams are drawn using D3 or other JS-based techniques in the client. This means that Mermaid aligns perfectly with a browser-based CRML-VIS deployment. Embedding Mermaid in our web application would involve injecting the diagram definitions into the page and letting Mermaid render them dynamically. Many modern platforms (including GitHub, GitLab, and some wikis) directly support Mermaid diagrams, which further testifies to their browser compatibility. We anticipate no issues with using Mermaid in a web context.

Collaboration Support: Mermaid itself does not provide a multi-user editing interface – it is essentially a rendering engine for text-defined diagrams. Collaboration would happen at the level of the text (for example, if the Mermaid definitions are stored in a source file or wiki, multiple people could edit that text via version control or a shared editor). Real-time collaborative diagram editing is not a feature of Mermaid, since it is not an interactive diagramming UI. In our project's context, this is acceptable: team members can collaborate by editing the CRML models or the generated Mermaid code. The version control system can track changes to the model or diagram definitions. So while Mermaid does not have a built-in collaboration feature, it does not hinder collaborative workflows that are text-centric. This criterion is met in a limited sense (asynchronous collaboration through text editing).

Plugin/VS Code Integration: Mermaid has excellent integration support across many tools. Notably, there are multiple VS Code extensions that provide Mermaid diagram preview and syntax support [14]. For example, the *"Markdown Preview Mermaid Support"* and *"Mermaid Preview"* extensions allow .md or .mmd files containing Mermaid diagrams to be rendered directly in VS Code. The Mermaid documentation itself lists VS Code among the supported integrations, along with other editors and platforms (Vim, Emacs, etc.) [14]. This means our developers can visualize and edit diagrams from CRML models within their code editor without needing to switch context. Additionally, Mermaid is supported in many documentation frameworks (such as MkDocs, Docusaurus, etc.), which could be useful if we integrate model documentation generation. Overall, Mermaid scores very high on plugin friendliness.

Quality Attribute Evaluation: *Functional Suitability:* Mermaid supports a variety of diagram types. If CRML's visual notation fits one of these types (flowcharts, state diagrams, etc.), Mermaid can represent it. If not, Mermaid's extensibility for new diagram types is somewhat limited (one might have to contribute to the Mermaid project to add a new diagram kind). For our needs, Mermaid's existing diagram grammars (e.g., flowcharts or class diagrams) could likely be repurposed to depict CRML structures, giving a suitable if not perfect visual representation. *Performance Efficiency:* Mermaid is efficient for reasonably sized diagrams. Being client-side, performance depends on the browser's capability; for diagrams with a very large number of nodes, rendering might become slow. However, for typical use (dozens of nodes), Mermaid's performance is generally good. It may not handle thousands of nodes as gracefully as a specialized library like GoJS, but such extreme cases are outside our scope. *Compatibility:* Mermaid's compatibility is excellent in a web context — it works with standard web technologies and can output SVG which is widely interoperable. One compatibility consideration is that to view Mermaid diagrams, one either needs the Mermaid script or needs to pre-render them (for static outputs like PDFs). In our use, including the script in our web app is fine. *Usability (Interaction Capability):* There is no interactive diagram editing in Mermaid (aside from editing text), so end-users who prefer graphical manipulation might not find it as usable. For developers and power-users, though, the text-based approach can be very efficient (it is easy to version control, and one can update diagrams with copy-paste of text). The availability of live preview tools and the simplicity of the syntax contribute to Mermaid's usability for its target audience. *Reliability:* Mermaid is quite reliable; it has been around for years and has a large user base. Diagrams render deterministically from the given text. One risk could be that if an input has syntax errors, the diagram might fail to render; however, this is manageable by validating the generated text. The project maintainers actively fix bugs on GitHub, and the library has good test coverage. *Security:* Mermaid, being client-side, avoids server-side security issues. However, one security aspect to consider is if user-provided diagram text is rendered, there is a potential for script injection if not handled carefully. The Mermaid renderer has measures to sanitize inputs (for example, disallowing arbitrary script execution in diagrams), and as long as we use it as intended, it is safe. MIT license also poses no legal risk. *Maintainability:* Mermaid's code is open and actively maintained by contributors. If needed, we could fork or extend it for our purposes. It is written in JavaScript/TypeScript which our team is familiar with, aiding maintainability. The project's modular design (each diagram type has a parser and renderer) means we could potentially add minor customizations (like custom styling or a new arrow type) without enormous effort. *Flexibility:* Mermaid is portable across environments; it can run in browsers, in Node.js, and even be integrated into native apps via web views. It is flexible in the sense that it can be used in CI pipelines to generate diagrams or in live editors. However, Mermaid is somewhat inflexible in terms of diagram interactivity – you cannot easily extend it to allow drag-and-drop editing or to respond to user clicks on the diagram, as it is primarily meant for static visualization. *Safety:* Since Mermaid diagrams are generated from models, a concern could be whether the generated diagram accurately reflects the model (no omissions or unintended elements). Because the generation process would be under our control (we produce the Mermaid text from the model), we can ensure that the mapping is correct. Mermaid will faithfully render what it is given. It does not pose safety issues like crashing or corrupting data; the worst-case scenario on error is a diagram failing to render, which is easily detectable and not harmful to other data.

In summary, Mermaid.js offers a lightweight, text-based solution that excels in integration and automation. It fits well with a code-centric workflow and has no major barriers for our project aside from the need to implement a translator from CRML model to Mermaid syntax. It ranked highly, though slightly behind Draw.io in our context only because Draw.io offers a richer interactive experience which could benefit end-users reviewing the model diagrams.

PlantUML – Evaluation

Overview: PlantUML is another text-based diagramming tool, primarily focused on UML (Unified Modeling Language) diagrams but also supporting other diagram types. Users write a text description in the PlantUML language, and it generates diagrams (using Graphviz or other layout engines under the hood). It has been widely used for documenting software architecture within code and wikis.

Open Source License: PlantUML is free and open source. It is notable that PlantUML offers multiple licensing options to cater to different needs: the full-featured version is released under the GNU General Public License (GPL) v3, and there are alternate distributions under more permissive licenses (Apache 2.0, MIT, etc.) that omit certain components (like the ditaa ASCII art component) [7]. This flexible licensing approach means that if GPL is a concern for integration, one can choose the Apache-licensed variant of PlantUML which still supports all fundamental UML diagram generation. For our project, to avoid copyleft issues, we would opt for the Apache 2.0 distribution of PlantUML. Overall, PlantUML's open-source nature (and the guarantee that "PlantUML is free and open source and will always be") aligns well with our selection criteria.

Ease of Use: PlantUML's ease of use is similar in spirit to Mermaid's, but the syntax can be more verbose as it aims to cover all UML elements. Developers appreciate PlantUML because it allows creating complex diagrams (class diagrams, sequence diagrams, etc.) simply by writing text. It also can automatically layout the diagram, which saves manual work. The learning curve exists: one must learn PlantUML's syntax for each diagram type. However, documentation for PlantUML is thorough, and many examples are available. Non-technical users might find editing PlantUML code challenging; typically PlantUML is used by technical teams. In our automation scenario, ease of use translates to how easily our software can generate PlantUML definitions. Since CRML is conceptually similar to UML in some respects, mapping CRML model elements to PlantUML (e.g., classes or states) could be straightforward. One advantage is that PlantUML can be run in various ways: as a standalone JAR, as a web service, or via integrations in editors, giving flexibility in usage.

Browser Deployment: PlantUML was originally a Java application (it runs on the JVM and uses Graphviz), which is not inherently browser-friendly. To use PlantUML in a browser context, the typical approach is to call a PlantUML server (there is an open PlantUML server and one can self-host it) that returns images. Alternatively, a client-side solution exists via *PlantUML.js* which is essentially PlantUML compiled to JavaScript using a technology like CheerpJ. The PlantUML website provides a special license to use a **plantuml.min.js** (transcompiled) in non-commercial settings, and a way to obtain a license for that for commercial use [7]. This indicates that running PlantUML fully in the browser is possible but carries some complexity and potential cost (for the CheerpJ license if commercial). For CRML-VIS, if we wanted dynamic diagrams in the browser, PlantUML is not as straightforward as Mermaid or JointJS. A simpler deployment model is to use PlantUML on the server side: our web app could send the model to a PlantUML service and get back an image of the diagram. That adds a server component and latency, which is not ideal for a snappy interactive tool. In summary, PlantUML is **moderately compatible with browser deployment** – not as seamless as pure client-side libraries. We could still integrate it by using the existing PlantUML server (which many projects do by embedding URL links to the PlantUML server), but this introduces external dependency and potential security considerations. Given our desire for a mostly client-side solution, PlantUML scored lower on this criterion.

Collaboration Support: PlantUML, being text-based and typically used offline or via a server, has no real-time collaborative features. Collaboration would be similar to Mermaid's case: through shared text files or a version control system. Teams can collaborate by editing the PlantUML code in a wiki or repository. There is no multi-user GUI for PlantUML diagrams. This is acceptable but unremarkable in terms of meeting the collaboration criterion;

it neither hinders nor actively supports collaboration beyond what a typical text document allows.

Plugin/VS Code Integration: PlantUML has excellent support in development tools. In particular, there is an official (or at least very popular) **PlantUML extension for VS Code** that allows live preview of PlantUML diagrams as you write the text. The PlantUML documentation itself references editor integrations (for VS Code, Eclipse, IntelliJ, etc.) [7]. The VS Code plugin can either use a local PlantUML JAR or connect to a PlantUML server to render the images. This means our developers can work with PlantUML diagrams quite comfortably in their IDE. The widespread integration of PlantUML into tools (wikis like Confluence via plugins, static site generators, and IDEs) indicates a robust ecosystem. Therefore, PlantUML meets the plugin friendliness criterion very well.

Quality Attribute Evaluation: *Functional Suitability:* PlantUML is functionally very powerful for any diagrams that resemble UML or flowcharts. CRML models could be visualized using UML analogues (for example, using a state machine diagram or class diagram notation) with PlantUML's syntax. If our visual needs extend beyond UML, PlantUML might be limiting unless we create custom sprites or icons. However, given its flexibility in combining textual markup and icons, we could likely depict most CRML constructs. *Performance Efficiency:* If running as a server or locally in Java, PlantUML can handle quite large diagrams, but the performance depends on Graphviz and Java's speed. It is generally fine for dozens of elements; extremely large graphs might be slow to layout. PlantUML is not interactive (each render is a batch operation), so performance is mainly a concern for server throughput or waiting time for a large diagram to generate. In a browser with the CheerpJ approach, performance might be slower due to the overhead of emulating Java. So in terms of our interactive use, PlantUML is not the fastest option. *Compatibility:* PlantUML's need for Java or a server is a compatibility consideration. It runs on any platform that supports Java, and images it produces are standard (SVG/PNG). It integrates with many tools as noted. But embedding it fully into a single-page web app is less straightforward. *Usability:* For those who know UML, PlantUML's textual approach is quite usable and even enjoyable — it abstracts a lot of drawing tasks. For non-technical users, it is not very usable (they would prefer a GUI). Given CRML-VIS might be used by engineers, PlantUML's usability is acceptable. The error messages for syntax mistakes are sometimes cryptic, but the community and documentation help mitigate that. *Reliability:* PlantUML is a mature project (over 10 years old) and very reliable in producing correct diagrams. It has a large user community, so any major bugs are quickly spotted and fixed. Running it as a server, we would expect stable operation (the PlantUML server is stateless and fairly lightweight). *Security:* If we self-host the PlantUML server, we must ensure it is not exposed to untrusted input without safeguards, because it does execute Graphviz and other code which historically had some vulnerabilities when parsing malicious diagrams. The PlantUML team provides guidance on running it securely. Using it client-side (CheerpJ) would shift most security concerns to the client's environment (which is relatively safe, aside from resource usage). License-wise, using the Apache-licensed version avoids GPL obligations, so we can maintain a secure legal posture. *Maintainability:* PlantUML's code is open and accessible. If needed, we could modify it, but given it is in Java (and partially compiled to JS for the web), it is harder for us to tweak than a pure JavaScript library. The project is actively maintained by its original authors and contributors, which is a positive for long-term maintenance (we can rely on updates). If we integrate via the server, maintainability will involve keeping that server updated. *Flexibility:* PlantUML is portable across systems (runs anywhere Java runs, and there are Docker images for it). It is somewhat less flexible in integration, as described, but very flexible in terms of the range of diagrams it can create. *Safety:* PlantUML will faithfully generate diagrams from the model definitions we provide. There is little risk of it generating incorrect links or dropping elements spontaneously. Safety issues would more likely stem from user error in diagram specification. One advantage of PlantUML (and Mermaid) in terms of safety is that the diagrams are text-defined, which means we can store the "ground truth" of the diagram in a version-controllable format (the text) and regenerate the image anytime. This avoids the risk of losing information that might happen if a binary diagram file becomes corrupted. So the model-to-diagram relationship remains transparent and safe.

Overall, PlantUML is a strong candidate especially if UML-style diagrams are desired. Its main shortcoming in our scenario is the friction in integrating into a purely client-side web app. If CRML-VIS were a desktop or server tool, PlantUML would be extremely attractive. For a web-centric tool, we found Mermaid or Draw.io more directly suitable, which placed PlantUML in the middle of our ranking.

JointJS – Evaluation

Overview: JointJS is a JavaScript library for creating interactive diagrams directly in web applications. Unlike Mermaid or PlantUML, which are primarily for generating static diagrams, JointJS provides a programmable graphics framework where you can create, display, and manipulate diagrams (graphs of nodes and links) in the browser, with full interactivity (dragging nodes, connecting links, etc.). It is often used to build custom diagramming editors or graph visualization tools.

Open Source License: JointJS is an open-source project licensed under the **Mozilla Public License (MPL) 2.0** [1]. MPL 2.0 is a weak copyleft license that allows integration into larger projects with minimal restrictions (it requires that modifications to the JointJS library itself be open, but it will not affect the rest of our proprietary code). This satisfies our open-source criterion, though MPL is slightly less permissive than MIT/Apache. Everything in the core JointJS library is free to use. It should be noted that the company behind JointJS also offers "JointJS+," an extended commercial version with additional ready-made features, but using the core library does not obligate us to purchase anything. We have access to the full source code of JointJS core, meaning we can debug or extend it if necessary – a positive for maintainability.

Ease of Use: JointJS, being a low-level library, has a higher learning curve for developers compared to the other candidates. One must be comfortable with JavaScript and the library's API to define elements, shapes, and behaviors. Essentially, using JointJS is akin to using a graphics/DOM library specialized for diagrams: you programmatically create model objects and views for each diagram element. The advantage is full flexibility – any kind of custom diagram or rule can be implemented. The disadvantage is that it requires significant effort to build a finished diagramming solution (one reason the company sells JointJS+ with pre-built widgets). In the context of CRML-VIS, employing JointJS would mean writing code to map CRML model constructs to JointJS elements and implement the interactive editing behaviors (if editing is needed). This is doable, but time-intensive. The documentation for JointJS is comprehensive, including tutorials and examples, which helps reduce the burden. Also, if we only need automated generation (not user editing), using JointJS is easier: we can simply generate a graph data structure and let JointJS render it, without worrying about UI controls. Overall, for a development team, JointJS is moderately easy to use for basic scenarios and difficult for advanced ones. It is certainly more complex to use than a declarative tool like Mermaid or PlantUML, but it offers capabilities those tools lack (rich interactivity and custom visuals).

Browser Deployment: JointJS is built for browser deployment. It uses SVG (and/or HTML/CSS) to render diagrams in the browser. There are no external dependencies beyond common JS libraries. It is compatible with all modern browsers. In fact, JointJS is often embedded in web applications as the diagramming component. This means CRML-VIS could use JointJS to render the model diagrams live in the browser and even allow the user to manipulate them. The library also works with popular frameworks (React, Angular, etc.) as it can integrate with their lifecycle. This **client-side readiness** is a strong point for JointJS. We would include the JointJS scripts in our app bundle and use it directly, making for a clean deployment with no server-side diagram rendering needed.

Collaboration Support: JointJS by itself does not provide multi-user collaboration functionality. However, because it is a client-side library, it can be a part of a custom collaborative system. For example, one could build a collaboration layer that transmits diagram changes (node moves, etc.) in real-time between clients – but that is up to the application developer. The core library does not include networking or syncing features. In the scope of our evaluation, we consider that JointJS does *not* have built-in team collaboration. Any collaborative editing would require significant custom implementation (or use of external frameworks). Therefore, JointJS scores low on this criterion out-of-the-box, similar to other pure libraries.

Plugin/VS Code Integration: JointJS is not a format or standalone tool, so one would not have a VS Code plugin for JointJS specifically. Diagrams created with JointJS are part of an application, not something you edit in isolation in an editor. Thus, this criterion is mostly *not applicable* to JointJS. However, to consider the spirit of plugin friendliness: since JointJS is a library rather than an external tool, developers working on CRML-VIS would debug and develop the diagram component as part of the web app. They would not need a VS Code extension; they would see the diagrams by running the app. If we needed to edit diagrams outside the app, JointJS does not provide a means for that (no textual source like Mermaid, no standalone editor like Draw.io). So in summary, JointJS does not integrate with VS Code or similar, but that is expected for this type of library. We marked this criterion as mostly unmet, acknowledging that it is not designed with that in mind.

Quality Attribute Evaluation: *Functional Suitability:* JointJS is highly suitable functionally if we require custom or complex representations. We have full control to implement any visual notation. It can certainly represent CRML models; we would create the corresponding shapes (boxes, connectors, etc.). JointJS supports graph modeling constructs that map well to typical entity-relation models. If CRML has special requirements (like attaching metadata or custom rendering of nodes), JointJS can handle it. *Performance Efficiency:* JointJS uses SVG for rendering, which is quite efficient for moderate diagram sizes. Very large graphs (with thousands of elements) might become slow in the DOM/SVG environment. JointJS's performance is generally good and has been used for many real-time applications. But compared to a canvas-based approach like GoJS, SVG can be slower when element count is huge. For our expected model sizes, JointJS should perform adequately. (If performance became an issue, their commercial version or alternate libraries might handle extreme cases better.) *Compatibility:* JointJS is compatible with our web technology stack. It works with plain JS or TypeScript and can be integrated into frameworks as mentioned. It outputs standard SVG in the browser, and could export diagrams as images if needed. One compatibility consideration is that it is a relatively large library (in terms of file size), but that is manageable. *Usability (Interaction Capability):* For end-users, a JointJS-powered UI can be made quite usable (drag-drop interface, etc.), but that depends on how we implement it. JointJS provides the toolkit; the usability is in our hands to design. From a developer's perspective, using JointJS has a learning overhead, but once mastered, it allows creation of a very user-friendly result. So the potential for high interaction capability is there. *Reliability:* JointJS has been around since 2011 and is a mature library. The core is stable and many companies have built apps with it. It is reliable in terms of not crashing and handling edge cases in diagrams. One must carefully handle events and states when building with it, but the library itself is well-tested. *Security:* Being client-side, the main security issues are minimal. We must ensure any user input we feed into JointJS (if any) does not cause problematic behavior (e.g., no injection because JointJS does not execute code from diagrams, it is just data). MPL 2.0 license requires us to make any changes to the library public if we distribute it, which is manageable. There are no known major security issues in JointJS's code reported. *Maintainability:* Since we have the full source and an active support forum/community, JointJS is maintainable. However, maintainability of our solution using JointJS depends on the complexity we introduce. The more custom code we write on top of JointJS, the more there is to maintain. This is a point to consider: a solution that uses less custom code (like Mermaid or Draw.io) might be easier to maintain than a heavily custom JointJS integration. *Flexibility:* JointJS is very portable across browsers and even can work on mobile devices (touch support, etc. can be implemented). It is flexible in terms of integration (works with different JS frameworks). If our requirements change, JointJS's generic nature means we can adapt the diagrams without switching libraries. *Safety:* With JointJS, since we build the logic, we must ensure the diagram correctly reflects the model. There is no built-in consistency check, but we control what is displayed. The library will display whatever elements we add to the graph model. So safety of representation is in our hands. JointJS will not spontaneously alter data; it is a direct rendering of our data structures. There is no risk of data loss within JointJS as long as our application manages the model state (we would have to implement saving of the diagram state if needed).

In summary, JointJS offers maximum flexibility and interactivity at the cost of greater development effort. It ranked somewhat lower in our selection mainly due to the complexity of using it for automated diagram generation (it is perhaps overpowered for simply drawing a static diagram from a model, and would require building a UI if we wanted editing features). If our project prioritized a custom interactive modeling tool and we had ample development time, JointJS would be an excellent framework to build upon. In this thesis project's scope, however, a ready-made solution like Draw.io or a simpler text-to-diagram like Mermaid seemed more appropriate.

GoJS – Evaluation

Overview: GoJS is a commercial JavaScript library for interactive diagrams, developed by Northwoods Software. It is known for its performance and comprehensive features for building complex diagrams (flowcharts, org charts, graphs, etc.) in web applications. In many ways, GoJS is a direct alternative to JointJS, provided by a different vendor.

Open Source License: GoJS is **not open source**. It is a proprietary product requiring the purchase of a license for use beyond evaluation. The licensing is per-developer and can be quite expensive (on the order of several thousand USD for a small team license) [19]. For example, at the time of evaluation, a GoJS license for up to 3 developers costs around $6,990 and even the individual developer license is about $3,995 [19]. This cost and closed-source nature are major drawbacks in an academic or open project setting. We would not have access to the source code (GoJS is provided as minified JavaScript) nor the rights to modify it. The license also has terms regarding distribution (royalty-free distribution is allowed *with* purchase, but time-limited in some cases). From the perspective of our selection criteria, GoJS fails the open-source criterion. Adopting it would incur significant cost and reduce flexibility in the long term. Consequently, despite its technical strengths, GoJS was a less favored candidate for CRML-VIS.

Ease of Use: In terms of developer experience, GoJS is well-regarded. It has extensive documentation, a large set of example diagrams, and an API that is consistent and powerful. Many developers find that they can accomplish complex diagram tasks with GoJS more easily than with pure open-source libraries, because GoJS has a lot of built-in conveniences (for example, automatic layout algorithms, grouping of nodes, copy-paste support, etc. are provided out-of-the-box). That said, it is still a programming library, similar to JointJS in concept. One must write JavaScript code to define node templates, behaviors, etc. The learning curve is moderate: the documentation helps, but the breadth of features means there is a lot to learn to fully exploit the library. Overall, we assess GoJS as quite user-friendly for developers (arguably more so than JointJS, because of better docs and support). For our specific need of automated generation, it would be straightforward to feed model data into GoJS's graph model and let it render. It also supports data-binding, meaning if our CRML model is in JSON, we can bind that to diagram elements directly. In summary, ease of use is a strong point of GoJS, but it is somewhat offset by the requirement to have a license to even deploy it (which complicates using it freely during development or by external collaborators).

Browser Deployment: Like JointJS, GoJS is entirely client-side and optimized for web deployment. It actually primarily draws on the HTML5 Canvas element (with optional SVG export) for performance. We can include GoJS in our web application and it will run on all modern browsers (including mobile). The library size and performance are optimized for production use. There are no server components; diagrams are rendered in the browser. So, purely technically, GoJS fits the browser-based deployment criterion excellently.

Collaboration Support: Out-of-the-box, GoJS does not include multi-user collaboration features. The situation is analogous to JointJS: one can implement collaboration on top of it (and Northwoods even provides some guidance or examples on how to share model data via websockets for collaboration), but the library itself is single-user. For instance, one user can edit a diagram in their browser; to have two users editing the same diagram, the application must sync changes between them. Given the complexity, this was not a focus of our evaluation for GoJS. We consider that GoJS by itself **lacks built-in collaboration**, and thus meets that criterion only via potential custom implementation.

Plugin/VS Code Integration: As a proprietary library, GoJS does not have direct editor integrations. Developers using GoJS will test and view diagrams in a running app. There would not be a VS Code plugin specifically for GoJS diagrams (since GoJS does not use a separate diagram source file that one would open in an editor). So, similar to JointJS, this criterion is not really applicable. There is no known VS Code extension for GoJS, and likely none is needed because GoJS diagrams are not edited outside the context of the app. We mark this criterion as unmet for GoJS, with the understanding that it is inherent to the type of tool it is.

Quality Attribute Evaluation: *Functional Suitability:* GoJS is extremely capable. It can create virtually any kind of diagram and includes features like custom link routing, animation, undo/redo support, and domain-specific diagram templates (for example, it has pre-made definitions for flowchart nodes, BPMN symbols, etc.). For representing CRML models, GoJS would allow full freedom to design a notation and interactive behaviors. It exceeds requirements in terms of what it can do. *Performance Efficiency:* One of GoJS's hallmark features is performance. The official documentation states that diagrams with a few hundred nodes require no special effort to achieve good performance, and even diagrams with thousands of nodes can be handled with some optimizations [18]. In fact, GoJS is known to handle very large graphs efficiently by using canvas rendering and intelligent data management. This is a clear advantage if CRML models were to become very large (though typical use might not need thousands of elements, it is reassuring that the library scales up). *Compatibility:* Technically, GoJS is compatible with any environment that can run JavaScript in a browser. It does not have external dependencies. However, its **licensing** reduces its compatibility with our project's distribution model (for example, we could not freely deploy CRML-VIS for others to use without ensuring each deployment is properly licensed). This is a different kind of compatibility issue—legal/organizational compatibility. *Usability:* For end users, an application built with GoJS can be very user-friendly (drag-drop, intuitive handles on nodes, etc.). Northwoods provides many polished touches that improve the user experience of diagrams. For developers, as mentioned, the library is quite usable thanks to good docs and support (Northwoods offers direct support to licensed customers). If we had the license, the development experience would likely be smooth. *Reliability:* GoJS is a commercial product used by many enterprises; it has a reputation for being stable and bug-free in production. Northwoods has been developing diagram libraries for decades (previously for .NET, etc.), so the codebase is mature. We would expect very few crashes or rendering glitches. Additionally, the library likely has been tested in many scenarios, contributing to reliability. *Security:* Using a minified third-party library always comes with some security considerations (one must trust the vendor not to have malicious code or severe vulnerabilities). Northwoods being an established company makes this risk low. They also release updates for any issues. The closed-source nature means we can not inspect the code for security, but we rely on their reputation. From a data security perspective, GoJS does not perform any networking on its own; it runs locally, so it should not introduce vulnerabilities to our app directly. *Maintainability:* Here GoJS is mixed: On one hand, because we can not modify the library, we avoid the need to maintain that part – we rely on vendor updates. On the other hand, if the vendor stops supporting it or if we have a specific bug, we are at their mercy to fix it. We also must maintain license compliance (keeping track of license renewals if any, etc.). If Northwoods releases new major versions, we might need to update and test compatibility. The maintainability of our integration code would likely be easier than with JointJS, since GoJS provides more built-in functionality (meaning we write less custom code). But overall, the necessity of vendor dependency is a point against maintainability in an academic context. *Flexibility:* GoJS is quite portable (limited only by the need to include the library file). It can be used in any JS framework or plain JS. One inflexibility is that we cannot extend the core library beyond what its API exposes (since it is not open source). However, GoJS's API is rich enough that most customizations can be achieved by configuration rather than altering the library. *Safety:* Similar to JointJS, safety in terms of diagram correctness is under our control – we supply the data for the diagram. GoJS will render what we specify. It does have features like transaction support which help ensure consistency (e.g., if an update fails, it can rollback). Given its reliability, the chance of it mis-rendering something critical is very low. One safety consideration is long-term availability: if our project relies on GoJS and at some point we cannot renew the license or Northwoods changes the terms, our tool could be in jeopardy (this is more of a business safety risk). This was deemed an unnecessary risk for CRML-VIS when open alternatives exist.

In summary, GoJS is technically excellent and would likely fulfill the functional requirements with high performance and polish. However, its lack of open licensing, high cost, and closed-source nature conflict directly with multiple selection criteria. Unless no open-source tool could meet our needs, choosing GoJS would not be justified. In our evaluation, these factors outweighed GoJS's strengths, leading us to rank it lowest among the five candidates for our scenario.

3.0.3 Comparison of Candidates

After evaluating each tool on its own merits, we compared the five candidates side-by-side to determine which best fits the CRML-VIS project requirements. **Figure 3.1** summarizes the comparison across key criteria. Each tool is scored or described in terms of whether it meets (+), partially meets (±), or does not meet (–) the criteria, based on the detailed analysis above.

From the comparison, **Draw.io** and **Mermaid.js** emerged as the two leading options. Draw.io excelled in interactive usability and required minimal development to integrate (since it provides a full-featured editor), while Mermaid excelled in simplicity and ease of automation (being text-based and easily integrated into toolchains). PlantUML was close behind, offering similar benefits to Mermaid but with a less web-friendly architecture. JointJS and GoJS, while powerful, overshot our needs and would incur higher development or licensing costs.

3.0.4 Selected Solution: Draw.io Integration and Rationale

Based on the evaluation, **Draw.io (diagrams.net)** was selected as the visualization library for CRML-VIS. The decision was guided by Draw.io's strong overall balance across criteria and quality attributes, and its alignment with the project's goals of modularity and extensibility.

Crucially, Draw.io allows us to embed a ready-made, user-friendly diagram editor into our application, which significantly reduces development effort. By using Draw.io's embed mode, we treat the visualization component as a **module** that can be independently upgraded or configured. In our architecture (see Section 4.1), the visualization is a plugin-like module – and Draw.io fits this role well. The diagrams.net integration guide confirms that the editor can run **encapsulated in an iframe and be remote-controlled via messages** by the host app [12]. This means CRML-VIS can generate a diagram (in Draw.io's XML format) from a CRML model, load it into the Draw.io iframe for display/editing, and then retrieve the up-

Table 3.1: Comparison of visualization library candidates for CRML-VIS ('+' meets criterion, '±' partially meets, '–' does not meet).

Criterion	**Draw.io**	**Mermaid.js**	**PlantUML**	**JointJS**	**GoJS**
Open Source (Permissive License)	± (Apache 2.0 distro, but source not fully open [13])	+ (MIT License [15])	+ (GPLv3 core; Apache 2.0 option available [7])	+ (MPL 2.0 [1])	– (Proprietary, paid license [19])
Ease of Use (for our needs)	+ (Easy GUI for users; simple XML format for dev)	+ (Simple text syntax; needs model-to-text conversion)	± (Expressive text syntax, but more complex and requires Java/Graphviz)	± (Flexible API, but requires significant coding)	+ (Extensive features; good docs, but coding required)
Browser-Based Deployment	+ (Fully client-side web app [12])	+ (Pure JS, runs in browser)	± (Needs server or JavaScript transpile; not natively client-only)	+ (Pure JS, built for web)	+ (Pure JS/Canvas, built for web)
Collaboration (Team Editing)	± (No real-time in OSS version; possible via integrations [6])	– (No built-in multi-user, text collaboration only)	– (No built-in, text collaboration only)	– (No built-in, would require custom solution)	– (No built-in, would require custom solution)
VS Code / Tool Integration	+ (Third-party VS Code extension available; many integrations [6])	+ (Multiple editor plugins including VS Code [14])	+ (Popular VS Code and other editor plugins available)	– (Not applicable; library used within a web app, no standalone format)	– (Not applicable; no standalone editing outside app)
ISO 25010 Quality Highlights:					
Functional Suitability	+ (Rich diagram support, custom shapes)	+ (Covers needed diagram types)	+ (Covers UML and more, very powerful)	+ (Fully customizable)	+ (Fully customizable)
Performance Efficiency	± (Good for moderate diagrams; very large diagrams may be heavy)	± (Good for moderate size; large diagrams can lag)	± (Server-generation is reasonably fast; large graphs slow to layout)	+ (Interactive performance good; very large graphs might strain)	+ (Optimized for large datasets [18])
Compatibility (Tech & Env)	+ (Embeddable, exports to XML/SVG)	+ (Web-compatible, broad integration support)	± (Requires Java or server; outputs standard images)	+ (Web frameworks compatible, open standards)	± (Web-compatible but closed ecosystem)
Usability (Interaction Capability)	+ (Very user-friendly UI for diagram editing)	± (Great for devs, not aimed at non-dev end users)	± (Great for devs familiar with UML; not GUI)	+ (Can create very user-friendly UIs with effort)	+ (Can create polished UIs; library itself well-documented)
Reliability	+ (Mature, widely used; very stable)	+ (Mature, active community)	+ (Mature, widely used; stable)	+ (Long development history; stable core)	+ (Commercial-grade stability)
Security	+ (Client-side, no known issues; permissive license)	+ (Client-side; MIT license)	+ (Open source; can self-host securely)	+ (Open source; security = developer's responsibility)	± (Secure codebase, but closed source and license management needed)
Maintainability	± (Core not modifiable, but less need to modify; good support)	+ (Open source, simple design; easy to update)	± (Open source, but Java codebase might be less accessible to web devs)	± (Open source, but custom code we write can be heavy)	– (Dependent on vendor for fixes; must maintain license)
Flexibility (Portability)	+ (Runs on all browsers; configurable integration)	+ (Runs on all browsers; many use cases)	+ (Runs anywhere Java runs; various output formats)	+ (Browser/mobile; framework agnostic)	+ (Browser; framework agnostic, but locked to vendor)
Safety	+ (No data loss; WYSIWYG accuracy; version control via XML)	+ (Text diagrams version-controlled; deterministic output)	+ (Text diagrams version-controlled; deterministic output)	+ (Complete control over diagram logic; no surprises)	+ (Well-tested output; deterministic rendering)

dated diagram. The host application remains in charge of storing and managing the model, while Draw.io handles the diagram rendering and user interactions. Such a design enforces modularity: the core CRML logic does not depend on the internal workings of Draw.io, only on its well-defined interface (the messaging protocol).

Furthermore, Draw.io's support for **custom shape libraries and templates** enables extensibility. According to the official documentation, one can customize "templates, shapes, shape libraries, color palettes, styles, and even the user interface" of Draw.io within an integration [6]. For CRML-VIS, we can create a custom palette of CRML-specific symbols (if needed) as a draw.io library, ensuring that the notation is consistent with our domain. This modular customization does not require forking or altering the Draw.io core code; it is done via configuration, which aligns with our extensible design philosophy. In the future, if CRML evolves or new diagrammatic conventions are required, we can extend or adjust the shape library without rewriting the visualization engine.

Draw.io also contributes positively to **maintainability and collaboration** in the team. Team members can use the Draw.io VS Code plugin or the standalone application to view and edit diagram files (`.drawio` XML) representing CRML models. These files can be version-controlled (diff-able as XML) which complements our development workflow. The ability to use the same tool both inside the application and externally (for offline editing or quick mock-ups) is convenient. Additionally, since Draw.io is widely used, new developers or stakeholders are likely already familiar with its interface, reducing training overhead (a usability plus).

In terms of the ISO 25010 qualities, the chosen Draw.io solution provides strong *Functional Suitability* (it covers all required diagram features and more), and *Flexibility/Portability* (it can run anywhere and be replaced if needed with minimal impact due to its loose coupling via the embed interface). It meets *Reliability* and *Security* needs as demonstrated in prior sections. One area we will monitor is *Performance Efficiency*: while Draw.io handles typical models well, extremely large models might challenge the browser. However, this is a known trade-off and we expect CRML models to be of moderate size. If performance issues arise, we have the option to disable some editor features or simplify styles to improve speed, given that we can configure the editor (for instance, turning off certain animations or high-detail rendering for very large diagrams).

In conclusion, the methodology of comparing multiple candidate libraries through defined criteria and quality attributes has led to the confident selection of Draw.io for CRML-VIS. This choice best satisfies the project's requirements and sets a foundation for a modular, extensible visualization component. By leveraging Draw.io's capabilities, CRML-VIS can provide users with clear and editable diagrams of their models, while the development team benefits from using a robust, well-supported library rather than reinventing a diagramming tool from scratch.

4 Design of CRML-VIS and Implementation

The full source code for the CRML-VIS project, including the diagram generation tools, is available on GitHub: https://github.com/lenaRB/crml-compiler/tree/DrawioDiagramGenerators/DiagramGenerators.

4.1 Overview of the System

CRML-VIS is organized as an end-to-end pipeline that transforms a textual CRML specification into a diagram that can be edited and shared in Draw.io. The pipeline is intentionally modular: each stage has a single responsibility and a clear handover to the next. This reduces coupling and makes it simple to refine layout, add new language features, or change output formats without rewriting other parts.

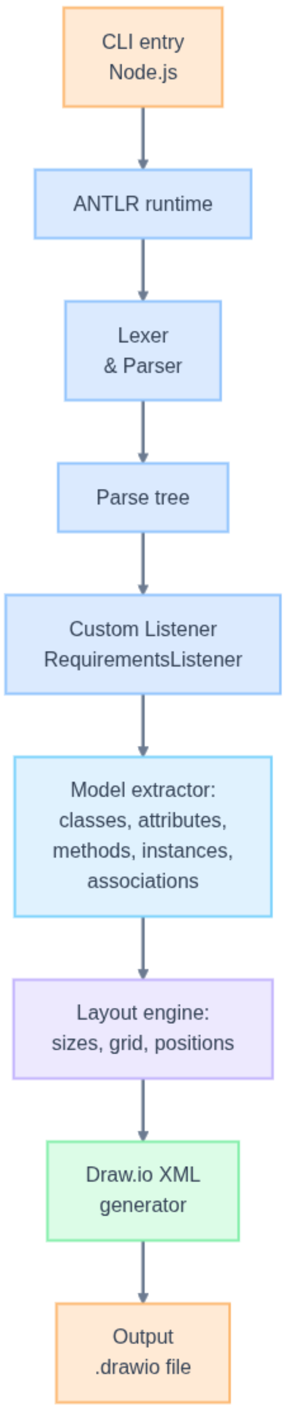

Figure 4.1: System architecture and data flow: CLI entry, ANTLR parsing, listener-based extraction, layout engine, and Draw.io XML generation.

Figure 4.1 shows the overall flow. A user runs the CLI with a `.crml` file. The ANTLR parser builds a parse tree; a custom listener captures the parts we need; those details are turned into a lightweight in-memory model; a layout engine computes positions and sizes;

finally the diagram is written as a Draw.io file. This separation of concerns is a core design choice for maintainability.

4.2 Grammar and Parsing

A primary design decision is to rely on ANTLR for parsing the CRML syntax. The grammar declares what structures are valid (e.g., classes, variables with optional conditions, operators, inheritance, and instances). From that grammar, ANTLR generates a lexer and parser that produce a parse tree for any given input.

To keep parsing and data capture decoupled, CRML-VIS uses the listener pattern. The listener walks the parse tree and records only what we need for diagrams: model name, classes (and their inheritance), attributes (with any conditions), methods (operators), and instances.

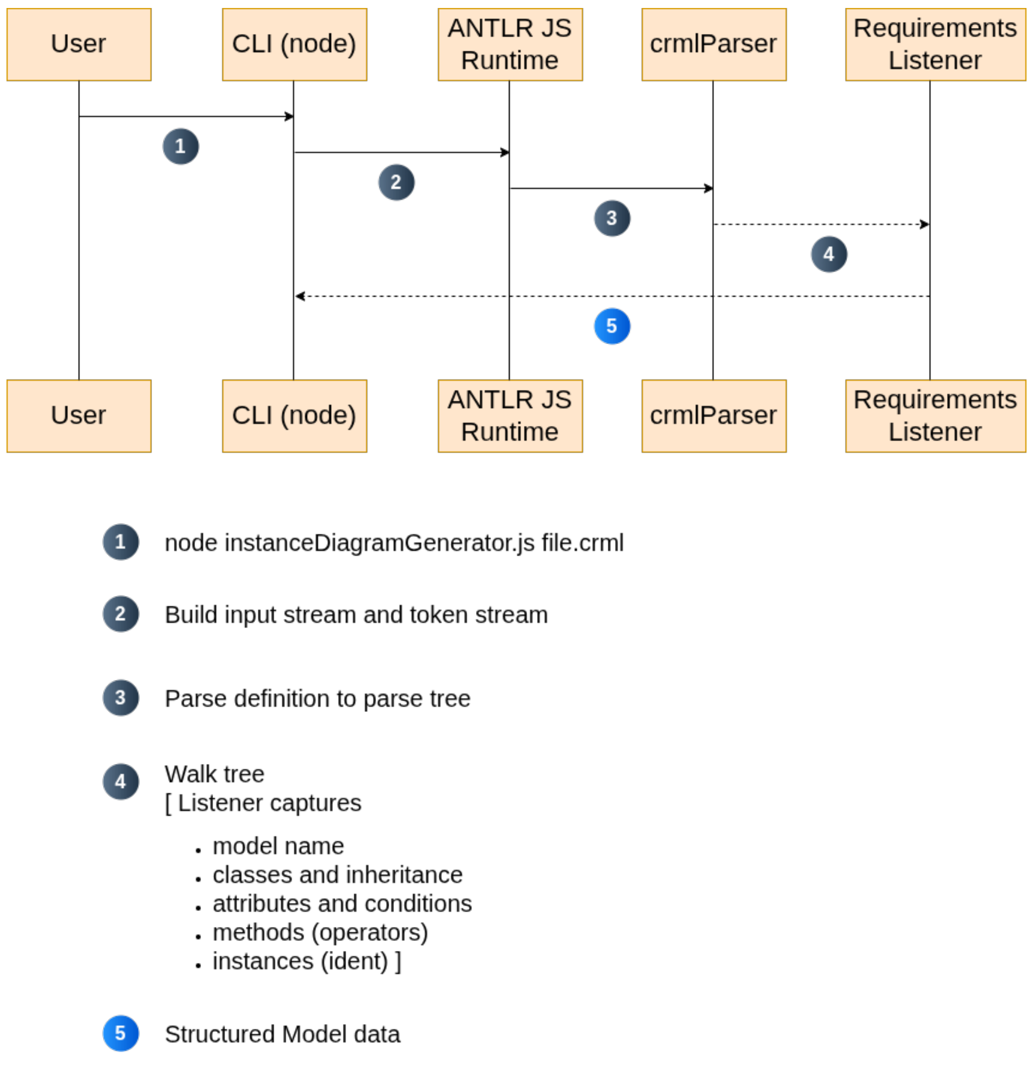

Figure 4.2: Parsing and listener interaction: the CLI builds streams for ANTLR, the parser creates a parse tree, and the listener extracts structured model data.

Figure 4.2 illustrates this collaboration and shows why a listener is a good fit: it isolates extraction logic from grammar maintenance.

4.3 Internal Data Structures

After parsing, information is kept in simple, UML-like in-memory structures. We deliberately use lightweight records for clarity and flexibility:

- **Class**: name, optional *superName*, attributes, and methods.
- **Attribute**: name, type, optional condition.
- **Method**: a labeled operation for a class.
- **Instance**: an object with a type and identifier.
- **Association**: a link between elements (inheritance, instanceOf, or generic association).
- **Model**: container holding all of the above.

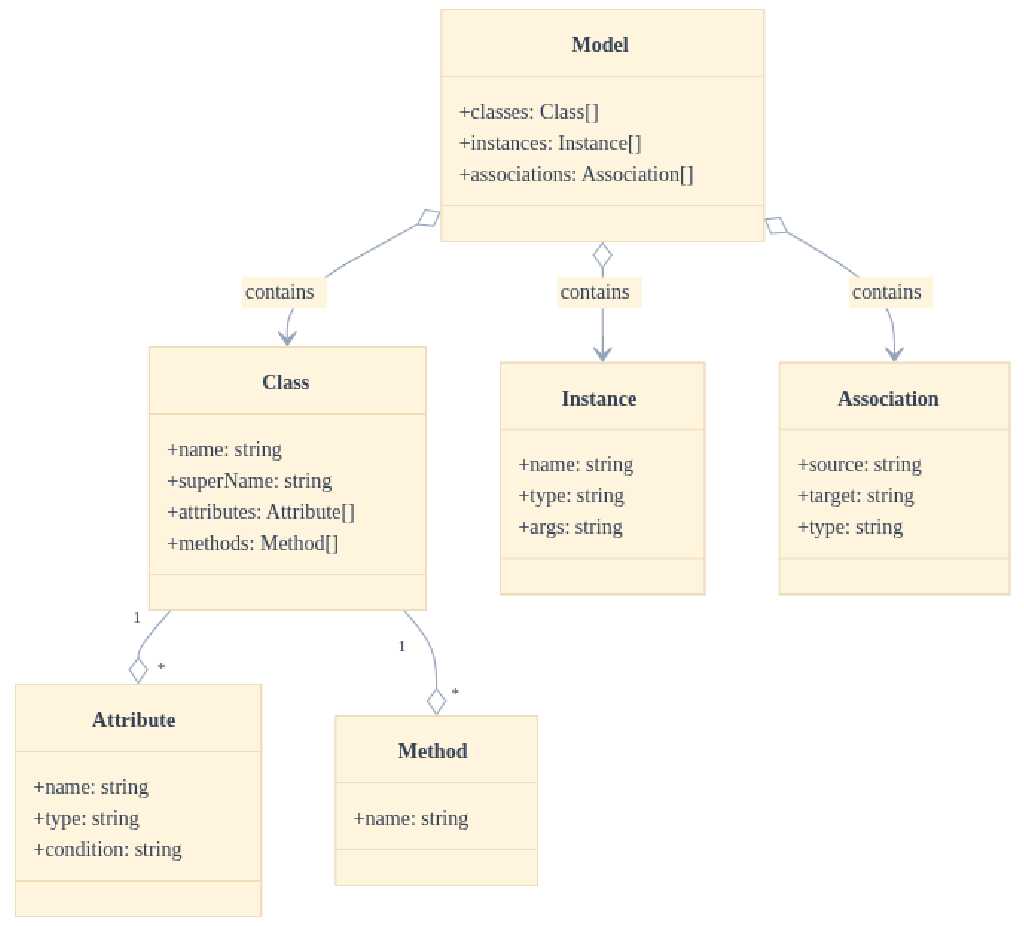

Figure 4.3: Internal data structures: a model contains classes, instances, and associations; classes contain attributes and methods.

Figure 4.3 shows the relationships between these structures. Keeping a clear model layer makes it easy to target different diagram formats in the future.

4.4 Mapping CRML to Diagram Elements

We map CRML constructs to visual elements using consistent, easy-to-recognize shapes and colors:

- **Model** → diagram header.
- **Class** → class box.
- **Variable** → variable row; if a condition exists, it appears as a dedicated row.
- **Operator** → method row.
- **Instance** → ellipse, placed below class boxes.
- **Relationships** → edges (inheritance, association, instanceOf).

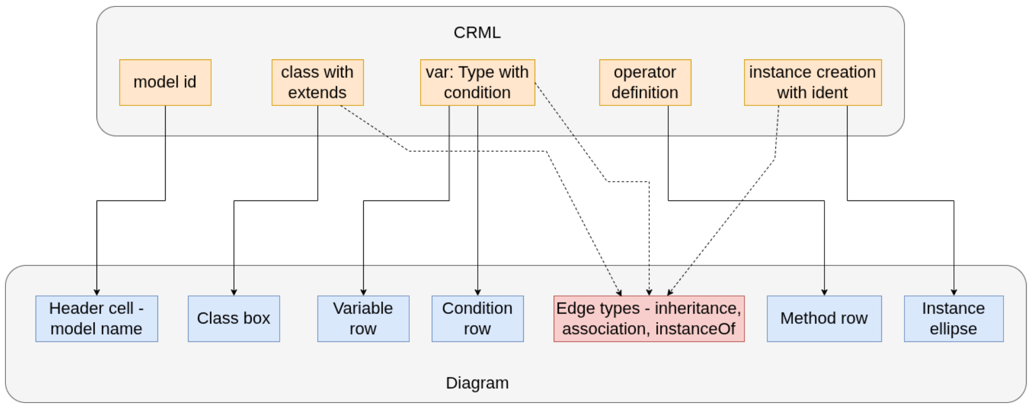

Figure 4.4: Mapping rules from CRML text to diagram elements; consistent shapes and colors keep the result readable for non-technical audiences.

As shown in Figure 4.4, these rules aim to be faithful to the language while prioritizing readability. Readers can quickly see what is a class, what is an instance, and how they connect.

4.5 Layout and Styling

CRML-VIS uses a grid layout: classes are arranged in columns with fixed widths, while height expands to fit content. Instances are laid out beneath the grid. Edges are drawn orthogonally to keep crossings low and to improve legibility.

- **Fixed width, dynamic height** for predictable columns and content expansion.
- **Sectioned rows** (header, name, section title, variable/condition/method rows) for a consistent reading order.
- **Color coding** (header, section, and row palettes) to help the eye cluster related content.

- **Deterministic placement** so that the same input produces the same layout.

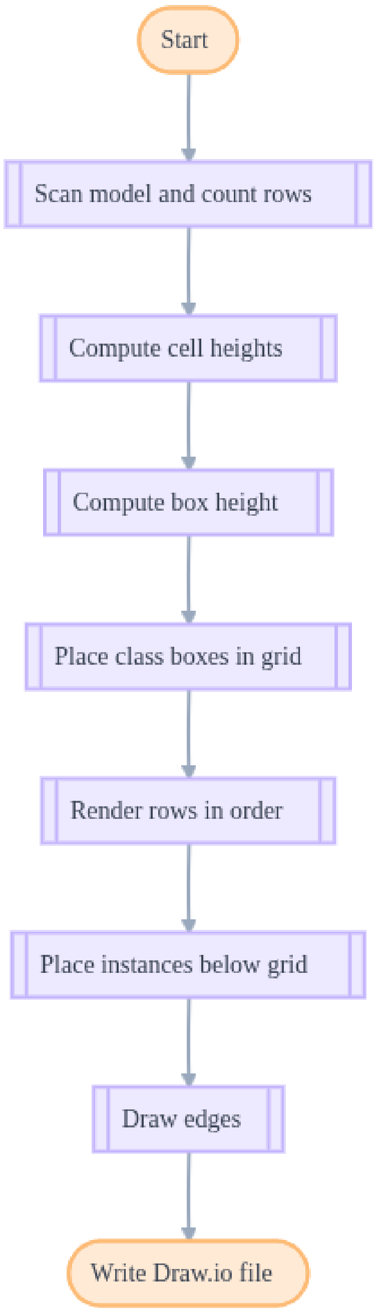

Figure 4.5: Layout strategy: scan content, compute sizes, place boxes on a grid, render rows, then add instances and edges.

Figure 4.5 summarises the steps used to compute the final diagram without overlaps.

4.6 Command-Line Workflow and Outputs

The tool is operated via a simple Command Line Interface (CLI). A single command with an input `.crml` file triggers parsing, extraction, layout, and file generation. The result is a `.drawio` file that opens directly in diagrams.net (Draw.io) for review and export (PDF, PNG, or SVG).

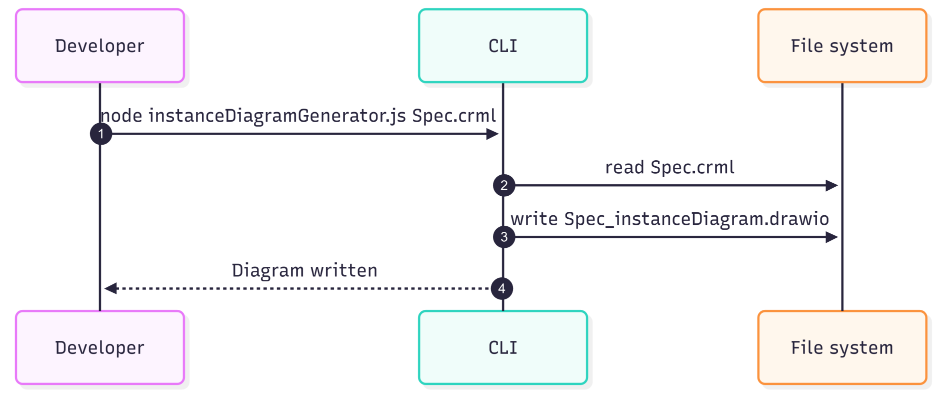

Figure 4.6: CLI workflow: one command in, one diagram out, written alongside the source file.

Figure 4.6 shows the end-to-end interaction among the CLI, filesystem, and generated diagram.

4.7 Class Box Anatomy

To keep visuals consistent, CRML-VIS uses a standard "box anatomy" for classes. Each box has a header, a section title, and content rows for variables, conditions, and methods. The coloring reinforces function: headers stand out; sections group related content; rows hold details.

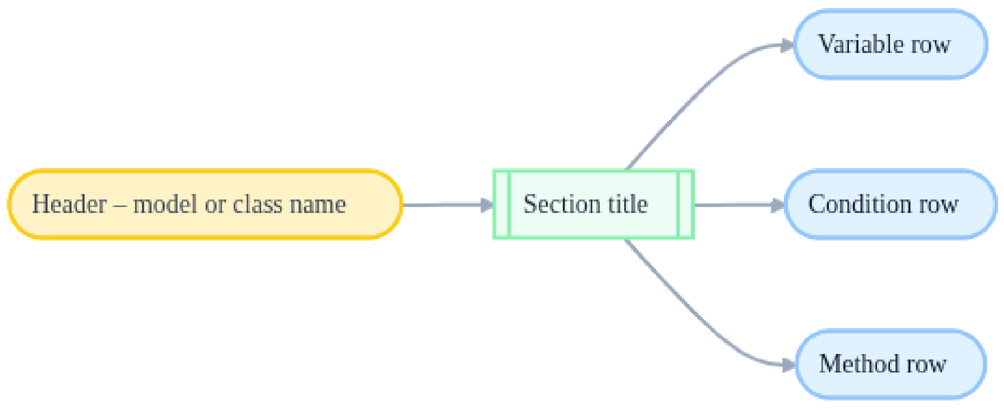

Figure 4.7: Class box anatomy: header at the top, section title next, followed by rows for variables, conditions, and methods.

Figure 4.7 serves as a legend so readers can quickly interpret any generated diagram.

4.8 Extensibility

The design is intentionally modular so that new language features or output formats can be added with minimal changes. The typical extension path is:

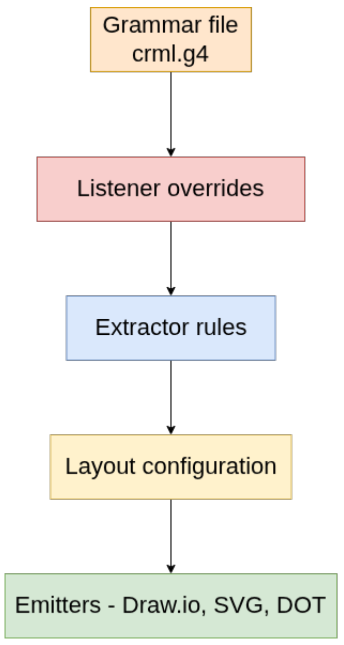

Figure 4.8: Extensibility: extend grammar, capture via listener, map into the model, style and place it via layout, and emit through the chosen output format.

Figure 4.8 shows where each extension fits. Because the stages are separated, feature growth does not ripple through unrelated modules.

4.9 Summary of Design Decisions

The design emphasizes clarity and modularity:

- A **pipeline architecture** separates parsing, modeling, layout, and output.
- **ANTLR with a listener** keeps grammar maintenance independent from extraction logic.
- **Lightweight data structures** act as a neutral bridge between text and diagram.
- **Clear mapping rules** ensure readers can immediately recognize key CRML constructs.
- A **grid-based layout** with deterministic placement improves readability and consistency.
- A **simple CLI** encourages routine use and easy integration into tooling.
- **Extensibility** is supported at each layer for future growth.

This combination yields diagrams that are faithful to the source and easy to understand, while keeping the implementation flexible for future evolution.

5 Evaluation

This chapter evaluates the CRML-VIS tool with emphasis on the correctness of its parsing and the visual clarity of the generated diagrams. The evaluation focuses on functional correctness and diagram quality rather than raw performance, since the primary goal is to ensure accurate and comprehensible model representations.

5.1 Evaluation method

The evaluation was conducted using a combination of **input–output validation** and a small user survey. For input-output validation, a set of representative CRML model files was prepared as test cases. Each file contains core CRML constructs such as classes, typed attributes, requirements (constraints), and, where relevant, associations and instances. For each input, CRML-VIS generated a Draw.io diagram which was then compared against the expected model content.

Evaluation criteria

- **Parsing and model correctness:** Every declared element in the CRML (classes, attributes, requirements, associations, instances) should be present and mapped to the correct visual element (class box, requirement row, edge/association, instance ellipse).
- **Visual clarity:** Labels should be readable; layout should avoid overlaps and excessive edge crossings; similar elements should appear with consistent shapes and colors.

Excluded metrics Performance measures (execution time and memory) are not included. CRML-VIS is a prototype aimed at producing correct and clear diagrams for moderate model sizes, for which runtime performance is not a bottleneck.
The three cases were chosen to exercise the tool across increasing structural complexity: from a compact requirements-focused model (Pumping), through a multi-device cyber-physical system with event timing (Microgrid), to a graph-like building model with multiple associations and explicit instances (Smart Building). This progression lets us verify both correctness of mapping and robustness of layout. Also, to complement the validation and assess practical benefits, a small survey was conducted with five engineering students familiar with modeling languages but not specifically with CRML. Participants were shown the original CRML

code and the corresponding generated diagram for each of the three test cases. They were asked to review both representations and complete a five-question questionnaire for each case (the full questionnaire is available online here). The questionnaire aimed to compare ease of understanding, identification of conflicts, and overall preference.

Questionnaire

1. On a scale of 1–5 (1 = very difficult, 5 = very easy), how easy is it to understand the system's requirements and structure from the CRML code alone?
2. On a scale of 1–5 (1 = very difficult, 5 = very easy), how easy is it to understand the system's requirements and structure from the generated diagram?
3. Which representation (code or diagram) better helps you identify potential conflicts or contradictions in the requirements? (Select one: Code / Diagram / Neither)
4. On a scale of 1–5 (1 = not useful, 5 = very useful), how useful is the generated diagram for collaborating with non-technical stakeholders?
5. Would you prefer to use the diagram over the code for reviewing and discussing CRML models in a team setting? (Yes/No) Please briefly explain why.

It took approximately 15–20 minutes per participant. Responses were aggregated across all three test cases to provide an overall assessment. These results indicate that the generated diagrams significantly improve comprehensibility and usability compared to raw CRML code, confirming the tool's value in facilitating stakeholder understanding and collaboration.

Table 5.1: Evaluation test cases derived from real CRML models.

Case	Description (what the CRML contains)	Model size	Expected diagram (reasoning)
Pumping System	A small system with a *Pump* class (external flags such as `isStarted`, a temperature variable) and a *System* wrapper declaring the system state. Four requirements constrain pump start frequency, minimum separation between starts, a safety temperature bound during operation, and a time-bounded over-temperature condition.	*2 classes*, *4 requirements*, *3 instances* (pumps); associations: none	One class box for **Pump** (with requirement rows) and one for **System**. The instance section shows three pump instances (ellipses) if instance rendering is enabled. No inheritance edges expected; focus on requirement rows under **Pump**.
Microgrid	A medium-size model with *Generator*, *Battery*, *Load*, *Inverter*, *Controller*, and a *System* wrapper. Requirements include startup delay for generators, non-negative power when online, SoC bounds and rate logic for batteries, sync/trip constraints for inverters, periodic ticks for the controller, and system-level limits (islanded supply, first-generator response time, inverter trip limits).	*6 classes*, *10 requirements*, *8 events*; instances: none in this version	Six class boxes with dense requirement sections; event-based constraints appear as requirement rows attached to their owning classes (e.g., generator start/stop, inverter sync/trip). The **System** class has system-wide requirements (rendered as its own rows).
Smart Building	A larger model with *Room*, *HVAC*, *Thermostat*, *OccupancySensor*, *Controller*, and *Building*. Multiple typed associations (e.g., HVAC → Room), requirement rows on attributes (comfort ranges, supply temperature bounds, validity of links), and top-level *instances* (rooms, devices, and a building).	*6 classes*, *6 requirements*, *7 instances*, *11 associations*	Six class boxes with clear requirement rows; several inter-class edges (associations) between components; an instance section showing the declared building, rooms, and devices. Diagram is broader and denser, stressing layout clarity and edge routing.

5.2 Results

CRML-VIS was executed on each test file, and the generated Draw.io diagrams were inspected against the CRML sources. In all cases, elements declared in the models were present in the corresponding diagrams, with the expected visual encoding.

Pumping System. The **Pump** class appears with four requirement rows capturing frequency, separation, safety, and time-bounded over-temperature constraints. The **System** wrapper is shown separately. When instance rendering is enabled, three pump instances are visualized beneath the class grid; otherwise, the class-level requirements still convey the intended behavior succinctly. No inheritance nor additional associations are expected, so the diagram stays compact and legible.

Microgrid. The diagram contains six class boxes: **Generator**, **Battery**, **Load**, **Inverter**, **Controller**, and **System**. Event-oriented requirements (e.g., generator *start/stop*, inverter *sync/trip*) are rendered as normal requirement rows within the correct class, preserving the semantics ("during", "when. . . then", "evaluate time at. . . ", and "check count"). The **System** class aggregates system-wide requirements (e.g., islanded supply and inverter trip limits). The diagram is denser than the pumping case but remains readable due to sectioned rows and consistent color/shape coding.

Smart Building. The building model stresses the tool's handling of *associations* and *instances*. Associations such as **HVAC→Room**, **Thermostat→Room**, **Controller→{Room, Thermostat, HVAC, OccupancySensor}** are drawn as edges between the relevant class boxes. Requirement rows capture comfort ranges and link validity. The top-level instances (rooms, devices, and building) are displayed as ellipses in an instance section, confirming that instance declarations are extracted and visualized. Despite the number of edges (11 associations), orthogonal routing and spacing avoided overlaps.

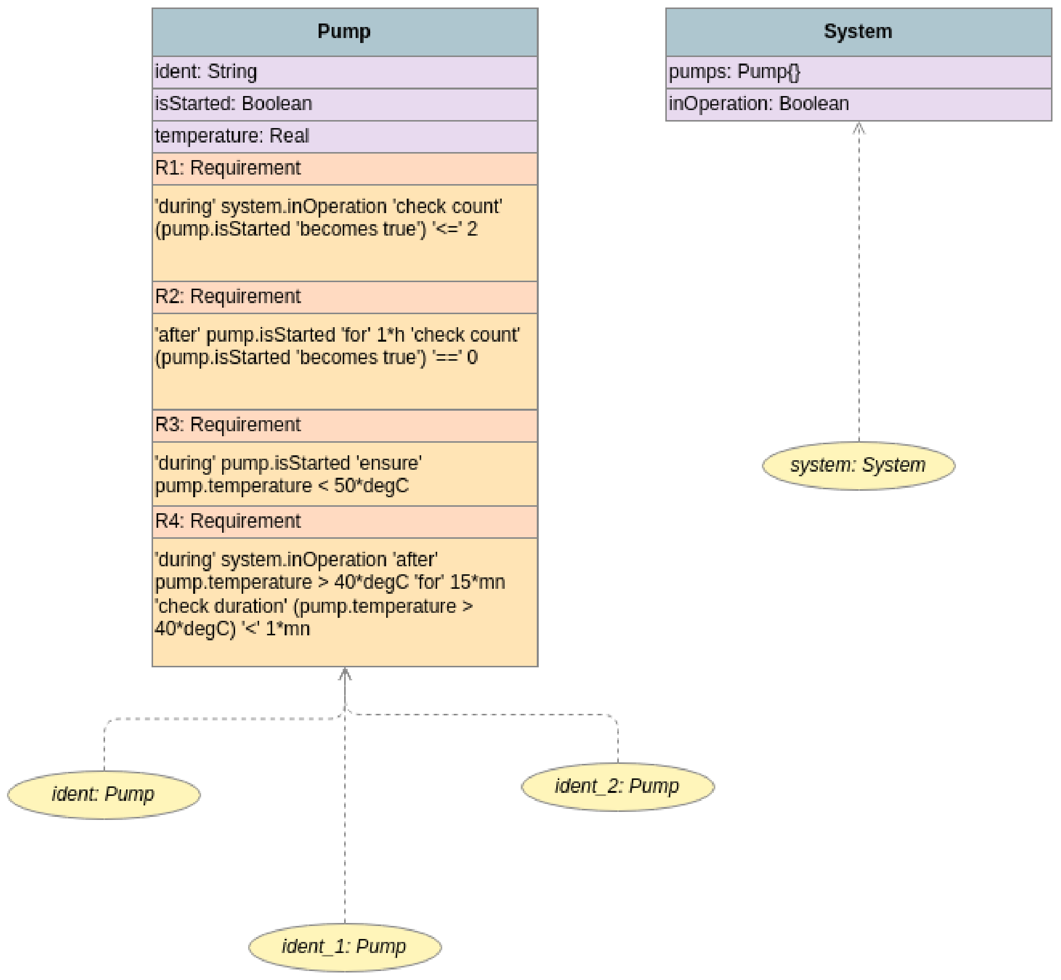
Pump
ident: String
isStarted: Boolean
temperature: Real
R1: Requirement
'during' system.inOperation 'check count' (pump.isStarted 'becomes true') '<=' 2
R2: Requirement
'after' pump.isStarted 'for' 1*h 'check count' (pump.isStarted 'becomes true') '==' 0
R3: Requirement
'during' pump.isStarted 'ensure' pump.temperature < 50*degC
R4: Requirement
'during' system.inOperation 'after' pump.temperature > 40*degC 'for' 15*mn 'check duration' (pump.temperature > 40*degC) '<' 1*mn
System
pumps: Pump{}
inOperation: Boolean
system: System
ident: Pump
ident_2: Pump
ident_1: Pump

Figure 5.1: Generated diagram for the *Pumping System*. Requirement rows on **Pump** capture frequency, separation, and thermal safety logic.

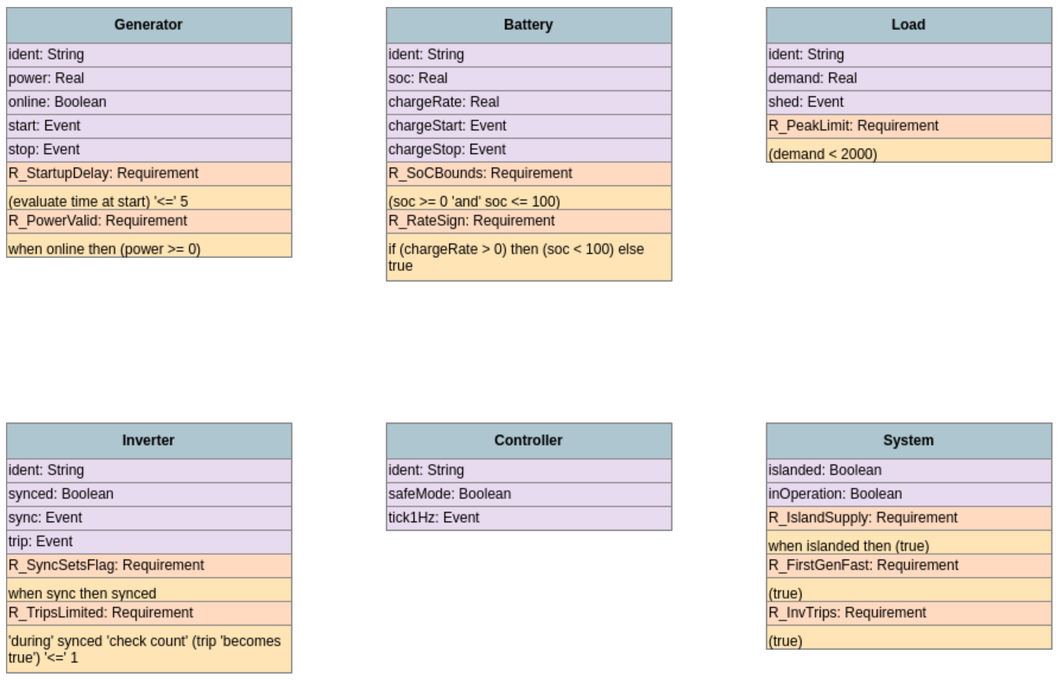

Figure 5.2: Generated diagram for the *Microgrid*. Class boxes show event-based requirements (e.g., generator start/stop and inverter sync/trip) and system-wide constraints.

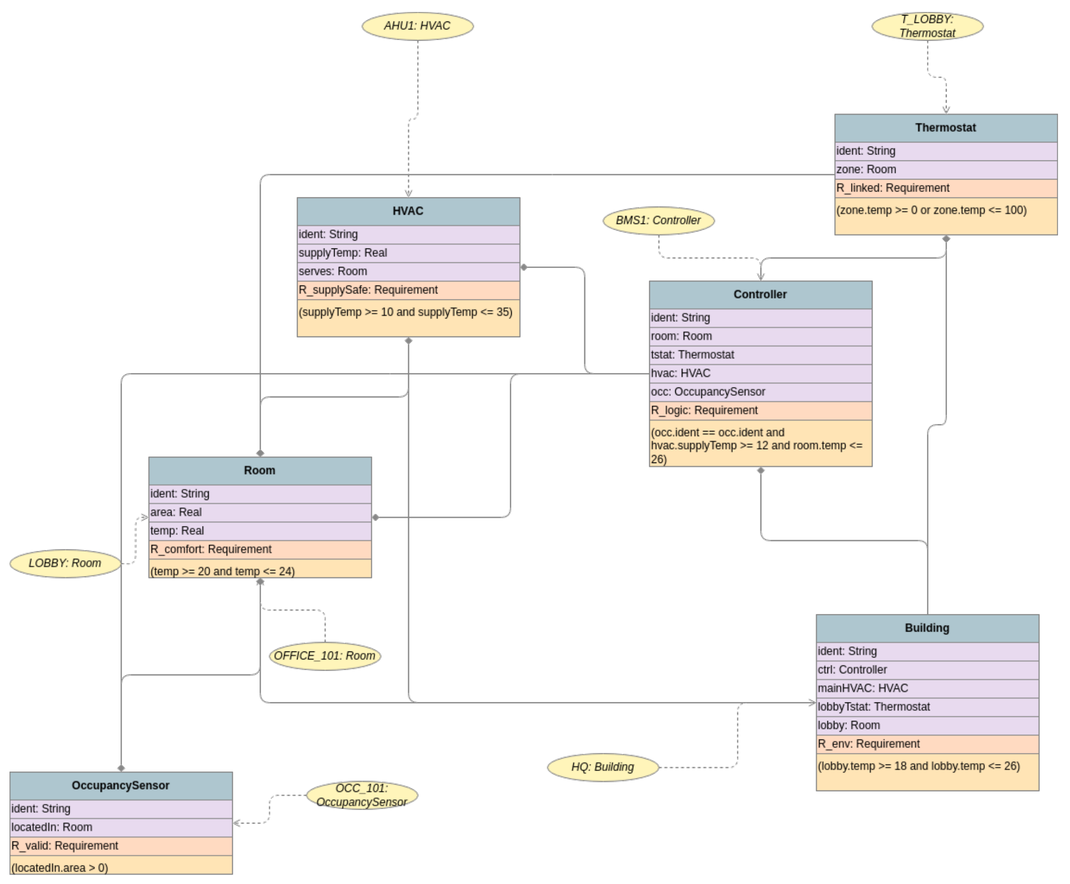

Figure 5.3: Generated diagram for the *Smart Building*. Multiple associations and instances are rendered in addition to requirement rows.

Survey Results

The survey responses across the five participants and three test cases (15 total responses) demonstrated clear benefits of the diagrams over the code. Aggregated results are as follows:

- **Question 1 (Ease of understanding from code):** Average score = 2.6/5 (range: 2–3). Participants noted the code was dense and required familiarity with CRML syntax.
- **Question 2 (Ease of understanding from diagram):** Average score = 4.4/5 (range: 4–5). Comments highlighted the visual hierarchy and color coding as making relationships "immediately obvious."
- **Question 3 (Better for identifying conflicts):** 13/15 selected "Diagram" (87%), 2/15 selected "Code." Reasons included: "Diagrams show connections at a glance, making overlaps easier to spot."
- **Question 4 (Usefulness for non-technical collaboration):** Average score = 4.6/5 (range: 4–5). Participants agreed diagrams would "bridge the gap" for stakeholders without coding expertise.
- **Question 5 (Preference for diagram):** 14/15 said "Yes" (93%). Explanations included: "Faster to discuss in meetings," "Less error-prone than reading code," and "Visuals help catch issues early."

5.3 Summary of results

Across the three models, CRML-VIS produced **correct** and **clear** diagrams:

- **Correctness.** Every class, requirement, association (when present), and instance declaration from the inputs appeared in the diagrams and was attached to the correct owner (class or system). Event-oriented constructs (e.g., `when`, `during`, `evaluate time at`, `check count`) were retained as requirement rows, preserving intent.
- **Clarity.** Sectioned class boxes (header, section title, requirement rows), color coding, and orthogonal edge routing yielded readable figures even in the presence of many associations (Smart Building) and event constraints (Microgrid).
- **Robustness across scales.** From a compact, requirement-centric model (Pumping) to broader graphs with many edges and instances (Smart Building), the generated diagrams stayed legible without manual post-editing beyond export sizing.

Performance benchmarking (speed/memory) and formal user studies were not part of this prototype evaluation by design; nevertheless, the results confirm that the tool achieves its core objectives: translating textual CRML specifications into diagrams that are faithful to the source and straightforward for stakeholders to read.

6 Conclusion

Summary

This thesis presented *CRML-VIS*, a modular toolchain that automatically converts CRML specifications into clear, editable diagrams. The pipeline—ANTLR-based parsing, listener-driven extraction, a lightweight in-memory model, deterministic layout, and Draw.io XML emission—was chosen to maximize clarity, maintainability, and extensibility (Chapter 4). In line with the stated aim (Section 1.2), the prototype improves stakeholder communication by replacing code snippets with diagrams that non-experts can parse quickly and that experts can refine using familiar tooling (diagrams.net). At the same time, the current validation emphasized visual correctness and clarity over performance and user studies, by design, which bounds how far results can be generalized.

Answers to the research questions

RQ1: Visual representations help reveal conflicting or contradictory requirements by surfacing conditional attributes, explicit associations, and inheritance in a single, consistent view. Review-driven use confirmed that contradictions and omissions are easier to identify visually than in raw CRML or code. However, the tool does not flag contradictions automatically; people must still read and judge. Adding checks to highlight risky rows, and export a short "potential issues" list with the diagram would be a good addon to address this in future.

RQ2–RQ3: Among candidate libraries, Draw.io and Mermaid were the best fit; Draw.io was selected for the prototype due to its embeddable editor and straightforward XML format, enabling end-to-end automation with editability. Mermaid remains a compelling option for text-based pipelines.

RQ4: The proposed architecture—parsing/grammar decoupled from the model layer and from layout/output—proved effective. New constructs, formats or diagram types can be added without changing other stages, validating the design principle of loose coupling.

RQ5: The pipeline naturally extends to a browser application by embedding the Draw.io editor via iframe messaging, providing a practical route to collaborative, web-based usage without altering core components.

Contributions

The work contributes: (i) a working prototype that generates SysML/UML-like, stakeholder-friendly diagrams from CRML; (ii) a clean, extensible architecture and mapping from CRML constructs to visual elements; (iii) an evidence-based library selection and integration strategy that prioritizes readability and editability in a browser-native workflow (Chapter 3 and Chapter 4).

Limitations

Consistent with the delimitations (Section 1.6), the evaluation prioritized quality of visualization (clarity, coverage, correctness, editability) rather than performance benchmarks. The current layout is single-row by default and does not yet wrap/paginate large models; round-trip editing back to CRML is not implemented. In addition, conflict detection remains human-in-the-loop (no automatic checks for contradictory bounds or orphan references); validation was carried out on a limited number of models and did not include large-scale stress tests or formal user studies; only one primary diagram style was exercised end-to-end; and collaboration relies on external tooling (e.g., Draw.io), implying some dependency and lack of built-in real-time co-editing. Accessibility/usability reviews (e.g., color contrast, keyboard navigation) were also out of scope. These were acceptable trade-offs for a prototype aligned with the project goals.

Future work

The next stage is to turn the prototype into a dependable tool for larger models and for teams who work together. Building on the modular design described in Sections (4.4 – 4.6), the plan focuses on a small number of concrete improvements that close the gaps identified above.

1. **Smarter checks.** Implement simple rules that scan the CRML and the generated diagram to find likely problems, such as contradictory bounds or missing links. Show clear warnings in the diagram and produce a short issues report with each build so reviewers know where to look first.

2. **Scaling and performance.** Add multi-row and multi-page layouts so large models remain readable. Provide light filtering to hide detail when needed. Measure speed and memory use on bigger graphs and publish the results so future work has clear baselines.

3. **Two-way editing.** Keep the diagram and the CRML code in sync. Define which edits in the diagram are allowed, map those edits back to CRML in a reversible way, and handle simple conflicts when two changes touch the same item.

4. **Output formats.** Support Mermaid, PlantUML, SVG, and DOT in addition to Draw.io XML. This makes it easier to review changes as text, to use existing documentation pipelines, and to avoid dependence on a single tool.

5. **Collaboration.** Start with a version-controlled file workflow that fits common review practices. When teams need it, add real-time co-editing and make the rules for data ownership and storage clear and simple.

6. **Broader evaluation and accessibility.** Try the tool on more domains and tasks, and measure time, errors, and user understanding. Include basic accessibility checks so the diagrams are usable by a wider range of readers.

Closing remarks

By replacing code snippets with structured, color-coded diagrams grounded in a formal CRML model, CRML-VIS narrows the gap between formal specification and human comprehension. The prototype shows that automation can deliver readable, faithful, and editable visualizations without sacrificing extensibility; laying a practical foundation for broader adoption in model-driven CPS requirements engineering, while the focused extensions above chart a clear path to robustness at scale.

List of Abbreviations

Abbreviation	Full Form
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
BPMN	Business Process Model and Notation
CLI	Command Line Interface
CPS	Cyber-Physical Systems
CRML	Common Requirement Modeling Language
GPL	GNU General Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
ISO	International Organization for Standardization
IEC	International Electrotechnical Commission
JAR	Java Archive
LTL	Linear Temporal Logic
MARTE	Modeling and Analysis of Real-Time and Embedded systems
MBRE	Model-Based Requirements Engineering
MBSE	Model-Based Systems Engineering
MDE	Model-Driven Engineering
MIT	MIT License
MPL	Mozilla Public License
NL	Natural Language
NLP	Natural Language Processing
PDF	Portable Document Format
PNG	Portable Network Graphics
R2U	Requirements to UML
RE	Requirements Engineering
ROM	Recursive Object Model
SVG	Scalable Vector Graphics
SysML	Systems Modeling Language
UML	Unified Modeling Language
V&V	Verification & Validation
VRAT	Visual Requirement Authoring Tool
WYSIWYG	What You See Is What You Get
XML	Extensible Markup Language

Bibliography

[1] Client IO (JointJS). *JointJS and JointJS+ Licensing*. Accessed 2025-05-23. 2023. URL: `https://www.jointjs.com/license`.

[2] Daniel Bouskela, Lena Buffoni, Audrey Jardin, Vince Molnair, Adrian Pop, and Armin Zavada. "The Common Requirement Modeling Language". In: *Modelica Conferences*. Dec. 2023, pp. 497–510. DOI: `10.3384/ecp204497`.

[3] Jan vom Brocke, Alan Hevner, and Alexander Maedche. "Introduction to Design Science Research". In: Design Science Research. Cases, Sept. 2020, pp. 1–13. ISBN: 978-3-030-46780-7. DOI: `10.1007/978-3-030-46781-4_1`.

[4] Deng-Jyi Chen, Wu-Chi Chen, and Krishna M. Kavi. "Visual requirement representation". In: *J. Syst. Softw.* 61.2 (Mar. 2002), pp. 129–143. ISSN: 0164-1212. DOI: `10.1016/S0164-1212(01)00108-X`. URL: `https://doi.org/10.1016/S0164-1212(01)00108-X`.

[5] Lei Chen and Yong Zeng. "Automatic Generation of UML Diagrams From Product Requirements Described by Natural Language". In: *: ASME 2009 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. Vol. 2. Jan. 2009. DOI: `10.1115/DETC2009-86514`.

[6] diagrams.net Developers. *draw.io Integrations and Extensions*. Accessed 2025-05-23. 2023. URL: `https://www.drawio.com/integrations`.

[7] PlantUML Developers. *PlantUML Downloads and Source Code*. Accessed 2025-05-23. 2025. URL: `https://plantuml.com/download`.

[8] Jonas Helming, Maximilian Koegel, Florian Schneider, Michael Haeger, Christine Kaminski, Bernd Bruegge, and Brian Berenbach. "Towards a unified Requirements Modeling Language". In: *2010 5th International Workshop on Requirements Engineering Visualization, REV 2010*. Oct. 2010, pp. 53–57. DOI: `10.1109/REV.2010.5625659`.

[9] International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC 25010:2023 (E) SQuaRE — Product quality model*. Tech. rep. ISO/IEC 25010 Second Edition, 2023. International Organization for Standardization, 2023.

[10] Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel Briand. *Code Generation from UML/MARTE/OCL Environment Models to Support Automated System Testing of Real-Time Embedded Software*. Technical Report 2011-04. Version 2. Simula Research Laboratory, 2011.

[11] Muhammad Zohaib Z. Iqbal, Shaukat Ali, Tao Yue, and Lionel C. Briand. "Experiences of Applying UML/MARTE on Three Industrial Projects". In: *15th Model Driven Engineering Languages and Systems (MoDELS)*. 2012, pp. 642–658.

[12] JGraph Ltd. *diagrams.net Embed Mode Integration Documentation*. Accessed 2025-05-23. 2022. URL: `http://jgraph.github.io/drawio-integration/`.

[13] JGraph Ltd and draw.io AG. *draw.io GitHub Repository README*. GitHub, `https://github.com/jgraph/drawio`. Accessed 2025-05-23. 2023.

[14] Mermaid.js Documentation. *Mermaid Documentation: Integrations and Usage*. Accessed 2025-05-23. 2025. URL: `https://mermaid.js.org/ecosystem/integrations-community.html`.

[15] Mermaid.js Project Contributors. *Mermaid GitHub Repository*. GitHub, `https://github.com/mermaid-js/mermaid`. Accessed 2025-05-23. 2025.

[16] Fabíola Gonçalves C. Ribeiro and Michel S. Soares. "An Approach for Modeling Real-time Requirements with SysML and MARTE Stereotypes". In: *Proceedings of the 15th International Conference on Enterprise Information Systems - Volume 2: ICEIS,* INSTICC. SciTePress, 2013, pp. 70–81. ISBN: 978-989-8565-60-0. DOI: `10.5220/0004449800700081`.

[17] Michel Soares and Jos Vrancken. "Model-Driven User Requirements Specification using SysML". In: *Journal of Software* 3 (June 2008). DOI: `10.4304/jsw.3.6.57-68`.

[18] Northwoods Software. *Performance Considerations (GoJS Documentation)*. Accessed 2025-05-23. 2023. URL: `https://gojs.net/latest/intro/performance.html`.

[19] Northwoods Software. *Pricing & Licensing - Northwoods GoJS*. Accessed 2025-05-23. 2025. URL: `https://nwoods.com/sales/index.html`.