TU WIEN Informatics

Feature Based Analysis of Extensible Modeling Frameworks and Libraries

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Business Informatics

by

Benjamin Weber
Registration Number 12025956

to the Faculty of Informatics
at the TU Wien

Advisor: Associate Prof. Dipl.-Wirtsch.Inf.Univ. Dominik Bork, Dr.rer.pol.

Assistance: Dipl.-Ing. Philip Langer, Dr.techn.

Vienna, 18th February, 2025

Benjamin Weber	Dominik Bork

Erklärung zur Verfassung der Arbeit

Benjamin Weber

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. Februar 2025

Benjamin Weber

Acknowledgements

I would like to express my deepest gratitude to my advisor, Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork, for his invaluable guidance, support, and encouragement throughout the duration of my research.
I am also profoundly grateful to Dipl.-Ing. Dr.techn. Philip Langer for his assistance and insightful feedback, which greatly contributed to the development of this thesis.
Lastly, I would like to extend my appreciation to TU Wien for providing an excellent academic environment and the necessary resources to complete this work.

Abstract

This thesis conducts a comprehensive feature-based analysis of extensible web-based modeling frameworks and libraries, focusing on their capabilities to support the creation and customization of a Domain-specific modeling language (DSML). These tools are indispensable in modern software development, providing structured environments for designing, visualizing, and managing complex systems. To evaluate their suitability for different use cases, a hierarchical feature model is constructed, breaking down the essential functionalities into categories such as extensibility, editing capabilities, model management, and user experience enhancements.
Seven frameworks: A Tool for Multi-paradigm modeling (AToMPM), JointJS, React Diagrams, Sirius Web, Sprotty, Graphical Language Server Platform (GLSP), and Web-based Generic Modeling Environment (WebGME) are examined in detail. Each is analyzed based on its implementation of features like constraint definition, meta-modeling, integration with external tools and advanced editing workflows. The research points out significant shortcomings, including insufficient attention to accessibility, alongside standout feature support such as intuitive interfaces and robust extensibility mechanisms. By comparing these frameworks, this thesis provides insights into their respective strengths, limitations, and applicability for practitioners and researchers. The analysis highlights areas for future development, emphasizing the need for more cohesive integration, enhanced user experience, and better support for emerging modeling paradigms. Ultimately, this work seeks to advance the adoption and evolution of modeling frameworks, contributing to more efficient and domain-tailored software development practices.

Contents

Abstract **vii**

Contents **ix**

1 Introduction **1**

2 Constructing the Feature Model **3**
2.1 Extensibility 5
2.2 Editing Capabilities 7
2.3 Model Management 11
2.4 User Experience 14

3 Modeling Framework Classifications **21**
3.1 AToMPM 21
3.2 JointJS 23
3.3 React Diagrams 24
3.4 Sirius Web 25
3.5 Sprotty 26
3.6 GLSP 28
3.7 WebGME 30

4 Evaluation **35**
4.1 Critical Feature Gaps 35
4.2 Standout Feature Support 36
4.3 Framework Deficiencies 37
4.4 Areas for Future Development 37

5 Recommendation **39**

6 Conclusion **43**

List of Figures **45**

List of Tables **47**

Acronyms **49**

Bibliography **51**

CHAPTER 1

Introduction

Modeling frameworks and libraries provide a set of tools, methodologies, and interfaces to create and manipulate models [38] [51], which themselves are abstractions of systems and processes. Models represent essential artifacts in software development by enabling the structured analysis of system interactions and components [31].

Extensible modeling frameworks are designed to be flexible and customizable, allowing them to be adapted to various domains. Often this is achieved through their modular architecture, for example enabling the development of plugins that extend the core functionalities, allowing for the creation of customized modeling environments. This modularity is crucial for adapting tools to meet specific domain requirements and integrating them with existing technologies [31]. Customizing these environments based on domain requirements eventually leads to the creation of a domain-specific modeling language DSML, which provides constructs that are tailored to specific domains, enhancing communication with domain experts and improving the accuracy of models [29]. By using DSMLs, domain experts can more naturally express concepts and ideas without needing deep programming knowledge, bridging the gap between system implementation and domain-specific requirements [29] [51]. Compared to more general purpose modeling languages, DSMLs also offer more expressive power when designing systems within a specific domain [38]. For this reason, only frameworks with some degree of extensibility will be highlighted. Furthermore, given the recent trend towards web-based platforms, which offer platform independence and ease of collaboration [51], only such tools will be discussed in this paper.

In terms of research and scientific applications, extensible modeling frameworks are significant for several reasons. Firstly, they facilitate the systematic development of DSMLs, which can lead to improved productivity and better quality of software systems through domain-specific component reuse. Secondly, these frameworks support rigorous analysis and verification of models, contributing to more reliable software development [29]. Hence, almost exclusively open source frameworks and libraries will be analyzed

in this thesis because compared to closed source and commercial tools, they are more relevant in a scientific context.
The primary goal of this paper is to classify various web-based extensible modeling frameworks by constructing and applying a feature model. Generally, a feature model is a hierarchical tree structure used to classify a specific domain by breaking it down into a subset of features [32]. The resulting features will subsequently be used to classify the frameworks based on their level of support for each feature.
Through detailed analysis and subsequent evaluation, we aim to uncover several critical insights about these frameworks. This paper will try to identify features that might be missing or lacking from certain frameworks, helping to pinpoint areas where these tools could be improved. Additionally, we will determine which frameworks are the most well-rounded in terms of their feature support and which are the most lacking, providing a clear picture of the current landscape of modeling frameworks. The selected frameworks and libraries do not represent extensible modeling environments as a whole. The analysis in this paper is not meant to be exhaustive, but merely aims to highlight tools that match different requirements.
Moreover, the analysis will reveal features that are either lacking or unexplored within the examined frameworks, highlighting potential areas for future development.
By presenting these findings, the paper aims to assist readers in identifying the tools that best fit their individual requirements, whether they are looking for comprehensive feature support or specific functionalities tailored to their domain.
Ultimately, this work seeks to contribute to the ongoing development and refinement of modeling frameworks, fostering an environment where both practitioners and researchers can effectively utilize these tools for improved software development.
The remainder of this paper is structured as follows. First, a feature model is going to be constructed in Section 2. This section will include an explanation of what a feature model is, how it functions, and how to interpret it. The research topic of modeling frameworks will be broken down into a subset of its features, each of which will be described in detail.
Next, the paper classifies the modeling frameworks in Section 3, starting with an introduction to all the frameworks analyzed. Each framework is described in detail, covering aspects such as platform type, technologies used, intended use, and distinctive features. This section will also include the current relevance of each framework, notable connections or relationships with other frameworks, and detailed results of the feature analysis.
A table presenting the results of the feature analysis follows these descriptions in Section 4. The subsequent discussion highlights vital features missing from certain frameworks, identifies exceptional feature implementations, significant deficiencies in frameworks, and addresses any features that are in need of further development.
Based on the results, recommendations are provided in Section 5, grouping frameworks that meet similar requirements.
Lastly, the paper will be concluded in Section 6 by summarizing the key findings, highlighting their implications for practitioners and researchers, and suggesting areas for future research based on identified gaps.

CHAPTER 2

Constructing the Feature Model

As stated previously a feature model is a hierarchical tree structure that specifies features of a certain domain. Each model has a root representing the domain, with every subsequent node representing a new feature that can potentially be broken down into sub-features. A feature consisting of sub-features is called a compound feature. A standalone feature without any child notes is referred to as a primitive feature [53].
The model imposes certain constraints that result in a variety of valid feature combinations. Constraints are generally defined between a parent feature and one or all of its child features [32]. Three specific constraints are relevant for the proposed feature model:

1. Mandatory Relationship: If a child feature is in a mandatory relationship with its parent and the parent feature is supported, the child feature must also be supported.

2. Optional Relationship: In an optional relationship, if the parent feature is supported, the child feature may be supported.

3. Or Group: An "or group" specifies a constraint between a parent and all its children. If the parent feature is supported, at least one child feature must be supported [32].

How this notation is realized graphically is shown in Figure 2.1.

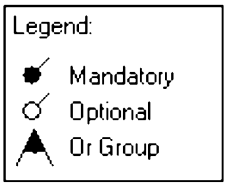

Figure 2.1: Feature model notation

In the following section, we will discuss the proposed feature model, with a particular focus on its primitive features. We will classify the modeling frameworks and libraries based on these features, indicating whether each framework provides full, partial, or no support. To ensure clarity, we will define these levels of support precisely, as they are essential for understanding the decisions made in the subsequent analysis.
Additionally one important distinction has to be made. In the upcoming sections we will differentiate between two types of "personas". First, an adopter is a person that uses a framework or a library to adapt it to their respective domain-specific needs by extending and customizing it or using it as a foundation to build custom modeling editors. Second, a user is a person that is entirely uninvolved in the implementation process and uses these customized tools to create models. In some cases these personas can overlap and one person can fulfill both roles but for the sake of unambiguous definitions we separate these terms.

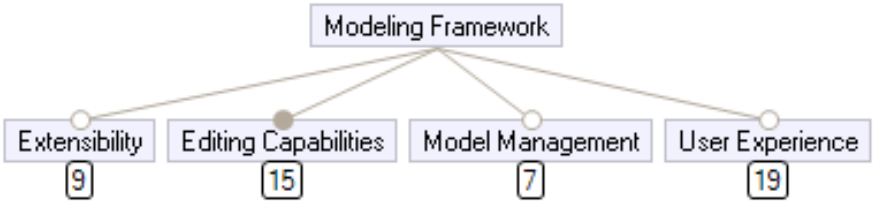

Figure 2.2: Top level feature model

Figure 2.2 shows the first level of features that can be supported by modeling frameworks. Relatively broad features are depicted here, which will be broken down further in the following sections.
Extensibility features are features that enable adopters to extend the framework both logically and visually. Also, depending on the level of support, the functionality of the framework can be extended by providing tools and possibilities not previously included [10]. We will only examine tools supporting this feature, but since not all modeling frameworks need to provide extension capabilities this node is marked as optional.
Editing capabilities are fundamental features of every modeling framework. They allow users to interact with or modify models. Using such features constitutes changing the state of a model in one way or another. Every framework needs to provide at least basic tools that fulfill this role. Hence, it is marked as mandatory.
Model management is about managing the lifecycle of models, including creating, storing, versioning, and easily switching between different models. It is strongly linked to model persistence, since many model management features imply some level of data persistence. The ability of the application to save models within its own environment ensures that they are retained and accessible within the applications context, as opposed to just being downloadable. Some frameworks do not include model management within their scope, making it optional.
Lastly, **user experience features** are add-ons that are generally not required for basic application of a framework. Instead they enhance the usage of a tool by providing essential

information and functionality that increase comprehension and decrease redundancy [41, 34].

2.1 Extensibility

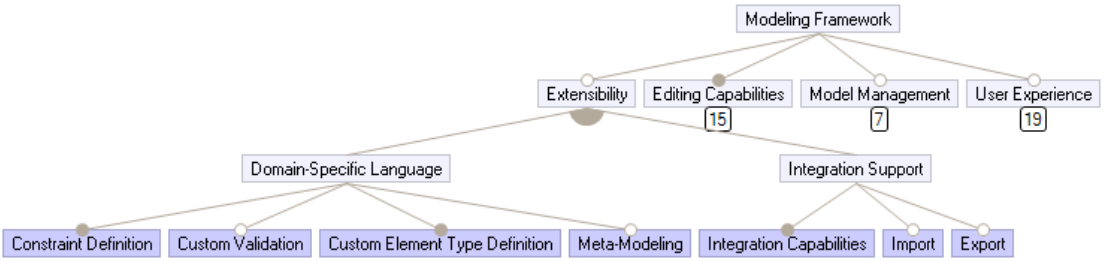

Figure 2.3: Extensibility

The extension of a framework is expressed mostly in two ways: the definition of a DSML [10] and the ability to integrate other technologies and tools into the framework [22, 51, 38]. At least one of the two need to be realized for a framework to be considered extensible. First, we discuss the creation of a DSML in order to extend built-in logic. Unlike general-purpose modeling languages such as UML, DSMLs are designed with constructs and semantics that directly reflect the concepts, rules, and logic inherent to a particular domain. Being more focused simplifies the modeling process for people in this domain. [20, 10]

As seen in Figure 2.3 DSMLs can be expressed in many different ways, one of which is the ability to define custom constraints. A framework supporting this feature allows the establishment of specific rules that govern relationships and attributes within the model, enabling adopters to define and enforce structural and semantic restrictions on model elements [10]. The descriptions of the levels of support are as follows.
Full Support: Adopters can define and enforce structural and semantic restrictions on model elements through a well-designed constraint system. Said constraints can be applied to relationships, attributes, or other elements, ensuring the model adheres to the specified rules.
Partial Support: Adopters may have access to basic constraint definition features, but the range of supported customization options or complexity of constraints could be limited. There might not be an easy way to create or manage constraints like a built in tool or interface but instead there only exists a bothersome solution.
No Support: The framework does not provide any built-in support for adopters to define or enforce constraints on model elements. The modeling environment relies solely on adherence to predefined rules, without allowing adopters to introduce custom constraints.

Although not mandatory, often a constraint definition feature is implemented with the ability to define custom validation rules in mind. At times both features may even

overlap. Custom validations empower users to validate models against previously defined rules, ensuring adherence to specified standards and constraints to ensure constant model correctness. [10, 22]
Full Support: Validation rules can cover a wide range of aspects, including syntax, semantics, and domain-specific criteria. Visual feedback and real-time validation mechanisms provide immediate insight into model correctness, highlighting areas of non-compliance with specified standards and constraints.
Partial Support: The framework provides some level of support for validation mechanisms, but with limitations. For example, only having fixed validation rules leaves custom constraint violations unresolved. Visual feedback or real-time validation mechanisms may be present but might be less informative or comprehensive.
No Support: The framework does not have built-in validation mechanisms. Lack of support for validation mechanisms may impact the ability to ensure model adherence to predefined criteria.

One particularly important element for defining DSMLs is the ability to define elements aligned with domain ideas and concepts. Among other things, this can include defining behavior, properties and the appearance of a new element type. Having refined element types available adds a layer of abstraction for users, eliminating the need to combine standard elements in complex ways to model domain concepts [52].
Full Support: The framework allows adopters to define new modeling elements aligned with domain requirements. Adopters can specify things such as properties, attributes, behavior and appearance of custom element types. Said element types can have relationships with standard or other custom element types. Tools or interfaces are provided to easily create, modify, and use custom element types within the modeling environment.
Partial Support: Adopters might be able to define only certain aspects of new element types, such as being able to specify behavioral traits but not its appearance. Custom element types may also have limited or predefined relationships with other element types. Creating and managing custom element types might require considerable effort or a special workaround.
No Support: Adopters and users are limited to using elements offered by the framework.

A prominent approach of defining rules and behavior is meta-modeling. Many tools rely on logic defined in code to enforce syntactical rules. An abstraction of this approach is to define syntax and semantics within a model that in itself describes models. This in essence is meta-modeling, making every model adhering to these rules an instance of this so-called metamodel [10, 33].
Full Support: The framework allows the explicit definition of meta-models. Framework adopters can define the structure and semantics of modeling elements within a separate meta-model.
Partial Support: Framework adopters can define basic structures and semantics within a meta-model, but there might be constraints on the complexity or depth of the meta-model.
No Support: Although the framework might provide other means of specifying syntax

and semantics of models, meta-modeling is not supported.

As mentioned previously, apart from DSMLs frameworks can also be extended by means of integration. This involves extending the core functionality of a framework by incorporating other tools into the framework [22]. In most cases this is achieved by making use of a modular architecture [51]. The performance of a framework in this aspect is largely determined by its level of integration support.
Full Support: The framework posses fully fleshed out integration capabilities, providing extensive support for seamless integration with various tools, systems, libraries, and frameworks. Adopters and users can easily connect the modeling environment with external tools and systems through well-defined interfaces or an application programming interface API. Comprehensive documentation and support are available to guide adopters and users through the integration process.
Partial Support: The integration process is supported but may require additional configuration or manual steps.
No Support: Adopters and users are unable to integrate the modeling environment with external tools, systems, libraries, or frameworks.

Support for integration often requires data interoperability. An import feature ensures the framework can ingest data from various external sources, ensuring that it can work seamlessly with data produced by different systems. An export feature facilitates data sharing with other systems, allowing for a smooth data flow [10].
Full Support: Users can import and export models using at least one widely adopted data interchange format such as JSON, XML, Avro, YAML, etc. Full Support implies interoperability with other tools and systems that also adhere to these standard data interchange formats. Import and export options include the ability to retain the structure, semantics, and relationships of models.
Partial Support: Users may be limited to a single data format specific to certain tools or platforms.
No Support: Users are unable to import or export models directly within the modeling environment.

2.2 Editing Capabilities

Editing capabilities are characterized by one common trait. All features allow users to change the state of a model. Be it a structural change or a completely new representation of the model, each editing capability is a tool that enables the user to interact with a model and alter it in some way.

Structural modifications are fundamental and among the most essential features a modeling framework has to provide when it comes to actually constructing a model. In this group, we include features to create both nodes and connections, as these element types are often the base building blocks used for constructing models [20].

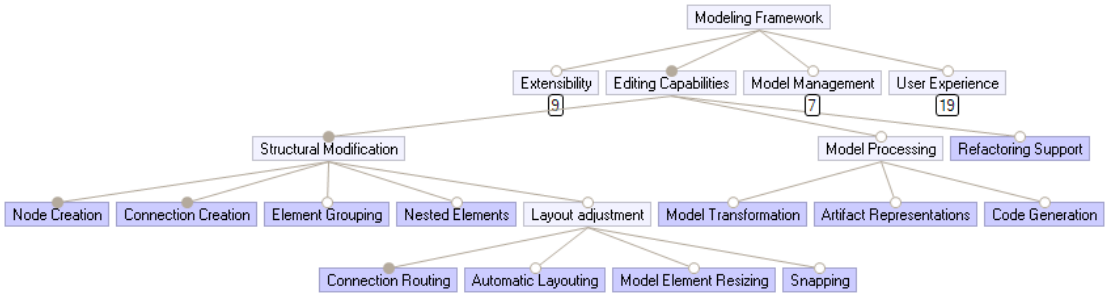

Figure 2.4: Editing capabilities

Full Support: Users can easily create nodes or connections using streamlined workflows, keyboard shortcuts, or context-aware menus. Drag-and-drop functionality may be available, allowing users to place nodes precisely within the modeling environment or connecting nodes with one another.
Partial Support: Node- and connection-creation workflows may be available, but they could be less intuitive or streamlined. Users may have access to basic methods for creating them, but advanced features may be limited. Visual feedback or assistance during node and connection creation might be less sophisticated.
No Support: Users may encounter challenges in creating nodes and connections, relying on manual and tedious processes and unintuitive workflows. Lack of support for efficient node and connection creation can impact the overall user experience, especially in scenarios where it is a frequent and critical task.

The grouping of elements is a feature that allows users to combine multiple elements into a single unit. This can simplify the organization and manipulation of related elements within a modeling framework. Grouping is particularly useful for maintaining coherence in complex models by enabling simultaneous operations on multiple elements [34].
Full Support: Users can select multiple elements and group them together. The grouped elements behave as a single unit, allowing for simultaneous movement, resizing, or other operations.
Partial Support: There could be constraints on the types of elements that can be grouped or the operations that can be performed on grouped elements.
No Support: Users are limited to organizing elements individually without the ability to group related elements.

A feature enabling users to further enhance the structural capabilities of their models is nesting. Nesting elements involves creating hierarchical relationships in which one element is contained within another. This parent-child structure helps manage and visualize complex models by establishing clear relational contexts among elements. Nesting enhances the clarity and functionality of models by reflecting structural dependencies [12].

Full Support: Users can nest elements within other elements, creating a parent-child

relationship. The framework provides tools or interfaces to easily create, modify, and manage nested elements.
Partial Support: Certain types of elements or relationships may not be supported within nested structures.
No Support: Users are limited to a flat structure without the ability to create hierarchical relationships between elements.

The following few features can still be classified as features that enable users to modify the structure of models, but additionally pertain to a subset of features that allow for more convenient layout adjustments. One such feature, connection routing, can enhance visual clarity of complex models. With the ability to adjust the routing path of connections, users can ensure that connections do not overlap with other elements resulting in a more organized structure and increased maintainability off models [41].
Full Support: Users can easily define and customize the routing path for connections, ensuring visually coherent and organized representations in the model. The framework offers automatic routing algorithms that intelligently adjust connection paths to avoid overlap with other elements and improve the overall layout.
Partial Support: Users may have access to basic connection routing options, but customization or advanced routing algorithms could be limited. Users may need to manually adjust connection paths.
No Support: Users are unable to modify connection paths. Lack of support for connection routing may result in less visually organized and coherent representations in a complex model.

Unlike connection routing, the next feature is not only limited to the relocation of connections but can be applied to all elements. Automatic layouting refers to the frameworks ability to organize model elements in a visually cohesive and readable manner without requiring manual intervention. This feature improves the overall presentation of the model by ensuring consistent alignment and spacing, thus improving the readability and user experience [20] [41].
Full Support: Users can rely on the framework to automatically arrange model elements in a visually cohesive manner, enhancing the overall presentation and readability of the model. Customization options may include settings for layout styles, alignment, and spacing, allowing adopters and users to tailor the automatic arrangement to their preferences.
Partial Support: The automatic layout algorithms may have constraints or may not handle complex model structures as efficiently.
No Support: Users need to manually organize model elements without the assistance of automatic layout tools.

A feature allowing for dynamic size adjustments of model elements further enhances layout clarity. Model element resizing allows users to adjust the size of elements within the modeling environment. This feature is essential for maintaining precise control over

the visual layout and to ensure that the presentation of information is clear and organized. Resizing can be performed manually or automatically, depending on the frameworks capabilities [34] [36].
Full Support: Users can easily resize model elements, providing flexibility in visual layout and ensuring precise control over the presentation of information within the modeling environment. The resizing mechanism is intuitive, allowing users to manually drag and resize elements on the canvas. Additionally, the framework may support automatic resizing, where elements adjust their size based on content or other layout constraints.
Partial Support: Users may have access to basic resizing options, including manual resizing, but the range of supported features or automatic resizing capabilities could be limited.
No Support: The framework does not have built-in support for Model Element Resizing. Lack of support for model element resizing may impact the precision and control over the presentation of information within the modeling environment.

The last feature aiding users in layout adjustments is snapping. Snapping between elements and on a grid is a feature that assists users in aligning model elements accurately by automatically snapping them into place when moved into close proximity to other elements or to a predefined grid. This feature ensures consistent spacing and alignment, contributing to a well-organized and visually appealing model layout [20].
Full Support: Elements can automatically snap to one another when moved in close proximity, facilitating alignment without manual adjustment. Snapping may include options for aligning to a grid, ensuring consistent spacing and arrangement of elements.
Partial Support: Snapping to nearby elements might be available, but it could have constraints or limitations on the types of elements that can be snapped together. There may be no grid available to guide the placement of model elements.
No Support: Users are required to manually align elements without the assistance of automatic snapping to nearby elements or a grid.

The next set of features involves the model having to be processed in some way. Model transformation, for example, refers to the ability to convert models from one format or language to another. This process includes defining transformation rules and often involves validation and verification to ensure the correctness of the converted models [40] [51].
Full Support: Adopters can define transformation rules to convert models from one format or language to another. Transformation processes may include validation and verification steps to ensure the accuracy of the converted models. The framework offers tools or interfaces for users to easily create, manage, and execute model transformations.
Partial Support: Users may have access to predefined transformation capabilities or templates. Customization of transformation rules may be limited, and the range of transformations might not cover all aspects of the model.
No Support: Model transformation needs to be performed manually or using external

tools, as the framework lacks native support.

Frameworks supporting different artifact representations allow interaction with models more than one type of visualization. For example, being able to switch from modifying a rendered version of a model using drag-and-drop tools to editing an equivalent fully textual version [20]. Of course, depending on how this feature is realized, multiple representations could also be synced and work in conjunction with each other.
Full Support: Users can choose between different artifact representations, such as graphical or textual visualization, based on their preferences and specific modeling needs.
Partial Support: Switching between representations may be available but with certain restrictions or manual steps. For example, the synchronization between different representations may be limited, requiring users to update changes manually.
No Support: Users are limited to a single type of representation for artifacts. Switching between representations is therefore not supported.

Code generation is the process of converting abstract models to executable code. This may also involve the ability for users to define custom mapping rules, not limiting the frameworks capabilities to a subset of coding languages [40].
Full Support: The framework fully supports code generation, enabling the automatic generation of code based on predefined models. Adopters can define mapping rules or templates that specify how elements in the model should be translated into code.
Partial Support: Users may have access to predefined code generation templates for specific languages.
No Support: Users are not able to define mapping rules or use templates to generate code based on an established model.

In order to maintain consistency and integrity within complex models, some frameworks implement support for refactoring. This implies that changes propagated throughout all parts of a model encompassing all dependencies related to the change [20] [38].
Full Support: Users can efficiently modify model elements, such as renaming classes, attributes, or other entities, with the assurance that changes automatically propagate throughout the entire model and other affected artifacts.
Partial Support: Refactoring tools may lack advanced intelligence or may require more manual intervention to ensure consistency throughout the model.
No Support: Changes do not propagate and must be applied throughout the model to maintain consistency between dependencies.

2.3 Model Management

Model management involves all processes concerned with organizing and maintaining models. The availability of such features depends for the most part on the type of framework. If a framework does not concern itself with the persistence of models, model

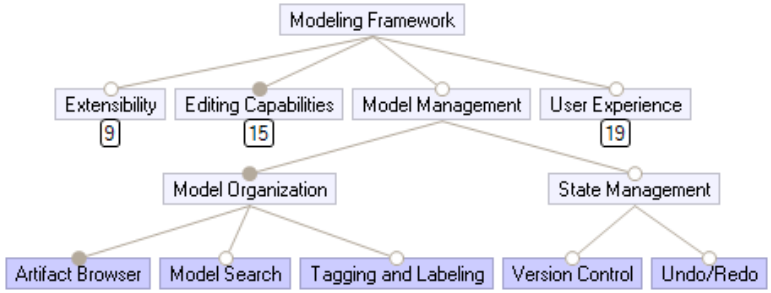

Figure 2.5: Model management

management does not become obsolete but is arguably less relevant [12]. This is specifically applicable to strictly code libraries that will be relevant for the subsequent analysis.

As mentioned above, model management typically implies some form of model persistence, which in turn requires model organization. This means users will need a way to easily access and interact with saved artifacts. There are several features that can enhance model organization among them are artifact browsers. They are a mandatory feature as they represent a central hub users have to use to access saved models or any other relevant artifacts. Most often artifact browsers organize models in a hierarchical folder structure and typically supports operations such as creating, renaming and deleting of models.
Full Support: Users can organize models in a hierarchical folder structure or a similarly organized layout. The artifact browser allows users to easily create, rename, move, and delete folders and models. Users can efficiently navigate through the models using the artifact browser, making it a central hub for managing and accessing modeling artifacts.
Partial Support: Users may have access to a basic organizational view, but customization options or features could be limited. Certain operations, such as moving or renaming models, may have constraints.
No Support: The framework does not provide built-in support for an artifact browser.

Unlike an artifact browsers a model search feature is not mandatory and only serves to enhance the model organization capabilities of a framework. A dedicated search functionality can enable users to quickly and efficiently locate specific models using criteria such as model names [52].
Full Support: Users can search through a collection of models using criteria such as model names to locate specific models.
Partial Support: Visual feedback or highlighting in the user interface is available but may be less comprehensive, providing limited enhancements to the search experience. This might make it slightly more challenging for users to identify relevant models quickly.
No Support: Users are unable to efficiently locate models within a repository through a dedicated search functionality.

Tagging and labeling refers to the ability to associate descriptive tags or labels with models for better categorization and organization. This feature allows users to manage models more effectively based on the assigned tags. This feature works best when implemented in combination with the previously mentioned model search.
Full Support: The framework fully supports tagging and labeling, allowing users to associate tags or labels with models for categorization. Tagging is flexible, enabling multiple tags to be assigned to a single model and vice versa.
Partial Support: Users may only have access to predefined tags and labels, or can assign only one tag per model.
No Support: Users are unable to associate tags or labels with models for categorization within the modeling environment.

The following two features involve state management, referring to functionalities within a modeling framework that allow users to control, track, and revert the state of models over time. State management features ensure that users can maintain control over a models development process, safeguard against errors, and allow collaborative contributions, all of which are vital for the management of complex modeling projects [38].
A framework with version control capabilities typically has a system in place, responsible for tracking changes made to models, providing a history of modifications and managing different branches for concurrent development [38].
Full Support: Users can view a history of changes made to the model, which may include additions, modifications, and deletions. The editor allows models to be reverted to a previous version. In addition, users can also create branches to work on different features, changes, or versions of a model concurrently. Branches can be merged, allowing changes from one branch to be incorporated into another.
Partial Support: Users may be able to access a simplified history of changes, possibly limited to specific types of modifications. The depth of the change history may be limited, covering only recent or significant changes. Merging may have limitations, and conflict resolution may require manual intervention.
No Support: Users are unable to view or revert to previous versions of the model, as the framework does not include a system that tracks changes. Users are limited to a linear development process without the ability to work on multiple branches concurrently.

An undo or redo feature is essential for flexibility in correcting errors and the ability to refine models by navigating through multiple levels of changes [34].
Full Support: Users can undo or redo a series of actions to revert the model to a previous state or reapply actions that were undone. Undo/Redo functionality is available for all types of changes, allowing users to navigate through multiple levels of modification to fine-tune the editing process.
Partial Support: Users can undo and redo basic actions, but the history might not cover all types of changes. The depth of the undo/redo history may be limited, allowing users to revert or reapply only a certain number of steps.
No Support: The framework does not have built-in support for undo/redo functionality.

2.4 User Experience

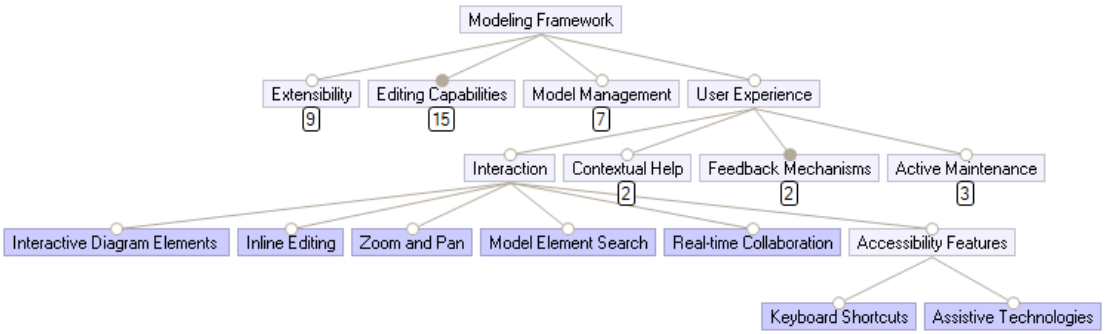

Figure 2.6: User Experience 1

User experience with regard to modeling frameworks can be described as the overall usability and efficiency of the environment. These features are designed to streamline workflows, provide visual and interactive enhancements, and ensure that the modeling environment is user-friendly and accessible to a broad range of users [41]. They could be realized as tools or simply as an enhanced ease of interaction with the framework.

The first type of features classifiable as user experience are interaction-based features. They mostly change how users interact with the modeling environment and its tools, increasing efficiency and accessibility. The first feature within this group are interactive diagram elements. For this feature we consider functionality that enables both adopters and users to configure and use model elements that are not only meant to be looked at but also interacted with. This could be realized through embedded hyperlinks, actions triggered via clicking, or other custom functions.
Full Support: The framework includes extensive capabilities for clickable and dynamic elements within diagrams. Interactive elements may trigger events or actions, allowing users to navigate to related information, open external resources, or execute specific functions. Adopters may be able to create interactive elements such as hyperlinks, buttons, or custom actions, or add them to existing model elements.
Partial Support: Users may have access to basic interactive elements with predefined actions and are unable to define custom ones.
No Support: The framework does not support interactive elements such as buttons, hyperlinks, or other elements with customizable actions.

Inline editing, just as the previous feature, also improves interaction directly with model elements. Instead of having to change names and perhaps properties through a separate input field, changes can be made directly within the modeling canvas [36]. This feature makes modeling more intuitive for users and makes workflows more efficient [34].
Full Support: The framework allows direct text editing within the diagram for various modeling elements. Changes made through inline editing are immediately reflected in the

model, providing a real-time editing experience. The inline editing feature is intuitive and user-friendly, enhancing the overall modeling workflow.
Partial Support: Inline editing may be available for certain types of modeling elements or specific properties. Changes made through inline editing might not always be immediately reflected in the model, requiring additional actions to apply the edits.
No Support: Users are required to open separate dialogues or forms to edit properties or textual information associated with modeling elements.

A zoom and pan feature allows users to increase the zoom of the modeling canvas and pan across it. This is especially useful when working on complex models that are more easily reviewed when looked at closely [36] [20].
Full Support: Users can zoom in and out of the model to examine details or get an overview of the entire model. Pan functionality allows users to navigate across the model canvas seamlessly.
Partial Support: Zoom and pan actions are not as smooth or responsive as in fully supported frameworks.
No Support: The framework does not have built-in support for zoom and pan.

Frameworks supporting a model element search feature allow users to search for specific model elements. Depending on the implementation they can be filtered based on their individual metadata, most often through their identifier alias name. This feature can also involve visual responses for found elements further enhancing the user experience and interaction.
Full Support: Users can efficiently locate specific model elements using search criteria, such as element names, types, attributes, or metadata. Visual feedback or highlighting in the user interface enhances the search experience, making it easy for users to identify relevant model elements.
Partial Support: Users can search for model elements, but visual feedback or highlighting in the user interface may be less comprehensive, providing limited enhancements to the search experience. This might make it slightly more challenging for users to identify relevant elements quickly.
No Support: Users are unable to locate specific model elements within a model through a dedicated search functionality.

Real-time collaboration enables multiple users to work simultaneously on the same model or project, with changes being reflected for all users. This implies the support of concurrent editing and live updates, as well as possibly presence indicators which in combination facilitates effective teamwork [38].
Full Support: The framework fully supports real-time collaboration, providing extensive capabilities for users to engage in collaborative activities in real-time. Multiple users can simultaneously work on the same model or project, whilst changes are reflected instantly to all participants. Real-time collaboration features include things such as concurrent editing, live updates, and presence indicators to show the contributions of each user.

Partial Support: Users may be able to collaborate in real-time, but the range of supported collaboration features could be limited. Concurrent editing may be supported, but with restrictions or potential conflicts that require manual resolution.
No Support: Users are unable to engage in real-time collaboration within the modeling environment.

The next two features, although still considered interaction enhancements as illustrated in Figure 2.6, can specifically aid impaired users by making the framework more accessible. They are thereby classified as accessibility features within the feature model.
Using keyboard shortcuts, users are able to more easily perform specified actions. Adopters may also be able to implement shortcuts with custom actions assignable to any combination of keys [34]. Users with certain impairments or disabilities may be more restricted in their ability to use common peripherals like mouses and keyboards. Frameworks with keyboard shortcuts or the ability to implement custom ones can support more restrictive layouts [34, 35]
Full Support: Users can perform a wide range of operations using keyboard shortcuts, including, for example, creating, editing, and navigating model elements. The framework provides documentation or an easily accessible reference for users to learn and memorize available keyboard shortcuts. Customization options allow adopters or users to define to modify keyboard shortcuts based on their preferences.
Partial Support: Users may have access to only the most basic keyboard shortcuts like redo, undo or copy and paste. Documentation or any other way to look up available shortcuts might not exist.
No Support: Users are unable to use keyboard shortcuts for efficient navigation or operations within the modeling environment.

"Assistive technologies" is a more general feature compared to others that includes various accessibility enhancements. Disabilities can be classified into five main categories: auditory, cognitive/learning/neurological, physical, speech, and visual with each requiring different considerations in tool design [44]. While we cannot list every specific feature that enhances accessibility, this inclusion in our feature model allows us to highlight more efforts towards improving framework accessibility. This includes functionalities that enhance the usability of the framework for all users, such as compatibility with assistive tools.
Full Support: The framework enables users with disabilities to more easily navigate, interact, and perform modeling tasks in some way. Documentation includes guidance on accessibility features and other aspects relevant to users with disabilities. The framework may have compatibility with assistive tools like screen readers.
Partial Support: Certain functions or features may have limited accessibility, and some interactions may not be fully compatible with assistive tools. The framework may not advertise any specific accessibility features but might still use technologies that are compatible with assistive technologies.
No Support: The framework does not target to improve accessibility. Users with dis-

abilities may face significant barriers when attempting to use the modeling environment.

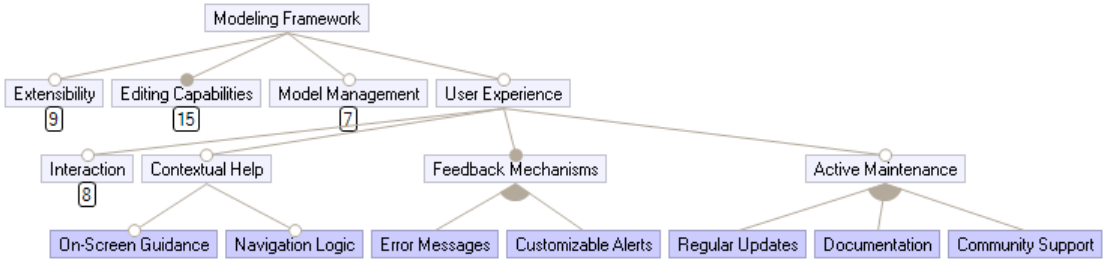

Figure 2.7: User Experience 2

The next set of features are responsible for giving users contextual help and making it easier to efficiently navigate the modeling environment. On-screen guidance, for example, dynamically adjusts to the users actions or selected elements, providing real-time support to facilitate the learning and use of the framework. The guidance may include tool tips, help messages [34], and contextual hints that adapt to different scenarios [36], making it easier for users to understand and utilize the tools available effectively.
Full Support: Users receive relevant guidance and tool tips based on their current actions, context, or selected elements in the modeling environment. Adopters are able to define tool tips and other forms of contextual help to aid users navigate and use the framework or modeling tools build upon it.
Partial Support: The framework might include predefined contextual help and tool tips but adopters are unable to customize this aspect.
No Support: Users do not receive contextual help or tool tips within the modeling environment,

Navigation logic enhances the users ability to traverse complex models by using predefined relationships and structures. This feature allows users to move seamlessly between related elements, using intuitive shortcuts or context-aware menus. This can streamline the process of exploring and managing interconnected components [36].
Full Support: Users can effortlessly traverse between model elements based on predefined logic, enhancing the overall modeling experience. Navigation options may include intuitive shortcuts, context-aware menus, or smart suggestions for transitioning to related model elements. The editor may support features such as hyperlinking between elements, making it easy to navigate to associated or referenced elements.
Partial Support: Navigation features may lack advanced context awareness or may not seamlessly transition between elements based on intricate logic.
No Support: Users cannot navigate between model elements using shortcuts, context-aware menus, or smart suggestions based on predefined logic.

Feedback mechanisms are similar to contextual help features, with both fulfilling a guidance role for the user. Whilst context menus can be used to guide the user through

most general workflows, feedback mechanisms are integral to ensure that users receive timely updates and error reports [34], enhancing their ability to understand and resolve issues within the modeling environment.

Error messages provide this sort of feedback, providing users with detailed and actionable feedback when something goes wrong in the model. This feature helps users identify problems quickly with clear explanations of errors and potential solutions.

Full Support: Users receive detailed and actionable error messages when issues or validation errors are encountered in the model.

Partial Support: Because of non-descriptive error messages users may need to rely on external resources or documentation for additional information on how to resolve errors.

No Support: Users are not provided with clear and informative feedback when errors occur.

The last three primitive features within the feature model all concern the degree of maintenance of the framework. It seems reasonable if at least one of the following features needs to be supported for the framework to be considered actively maintained.

The first feature, "Regular Updates", ensures that the framework evolves with new features, security enhancements, and bug fixes. This feature keeps the modeling environment current, addressing any issues and adding functionalities that improve user experience and framework stability.

Full Support: The framework provides adopters with timely and consistent updates. Updates include enhancements to functionality, new features, improvements in security, and bug fixes.

Partial Support: Updates may be less frequent or may focus on specific aspects such as functionality or security.

No Support: Adopters are not provided with regular updates regarding functionality or security. There may be infrequent or no releases that address improvements or bug fixes.

An actively maintained framework should, in most cases, also provide up-to-date documentation. This includes detailed information about the frameworks features, usage, and best practices. Well-maintained documentation enhances the learning curve and aids users in utilizing the framework efficiently.

Full Support: Documentation detailing the various features and how to use them is available, providing up-to-date and comprehensive information for adopters. The documentation is well organized, searchable, and easily accessible, enhancing the adopters learning experience.

Partial Support: Documentation may cover essential aspects, but certain areas could be less detailed or lack comprehensive information. Adopters may find the documentation helpful for basic tasks, but advanced topics or use cases might be less well documented.

No Support: There is no documentation provided or the provided documentation is severely out of date.

The last primitive feature, "Community Support", constitutes the existence of a platform for users to connect, share knowledge, and seek assistance from other users. This can include forums, discussion boards, or social media groups where users can discuss issues, exchange tips, and collaborate on problem solving.
Full Support: Adopters can engage in discussions, ask questions, and seek assistance from the community. There are dedicated forums, discussion boards, or social media groups where users can share experiences, tips, and best practices.
Partial Support: Adopters may have access to forums or discussion boards, but community activity could be less vibrant. Documentation and official support channels may play a more prominent role, with limited community-driven support.
No Support: Adopters lack access to active forums, discussions, or social media channels for community support. There is little to no community-driven knowledge sharing or assistance.

CHAPTER 3

Modeling Framework Classifications

In this chapter we will both present all frameworks evaluated in this thesis and analyze them individually based on the features previously defined. To increase the accuracy of the evaluation results, we will also introduce an additional fourth level of support, namely "not supported by core". This level pertains to all features that are not covered by the core implementation of a framework but are stated to be available through other means such as integration of other tools and libraries or custom implementations.

3.1 AToMPM

AToMPM is a fully web-based modeling framework, meaning it is not restricted by things such as operating systems, platforms or a client device. It is referred to as a research framework, meaning its main objective is to further research in Model-driven engineering (MDE), allowing for rapid prototyping of new modeling concepts, languages, or transformation techniques [51].
Adopters and users interact with AToMPM mostly using the already implemented graphical user interface GUI. The main interface allows users to create models visually, drag and drop elements, connect them, and edit the properties. Using the same GUI adopters can define their own DSML or define model transformations [6]. AToMPM is based on a client-server architecture that is illustrated in Figure 3.1. The clients are web browsers, and the server is a Node.js application. At its heart, AToMPM has a minimal core called "Stateful". This core does not have any specific modeling functionality but serves as the foundation for everything else. The reason for its high modularity and extensibility is its plugin-based system. Almost everything, except for the minimal core, such as modeling functionalities and toolbars, is built as a plugin. The architecture allows for easy extension. New plugins can be added, and even external servers in different programming

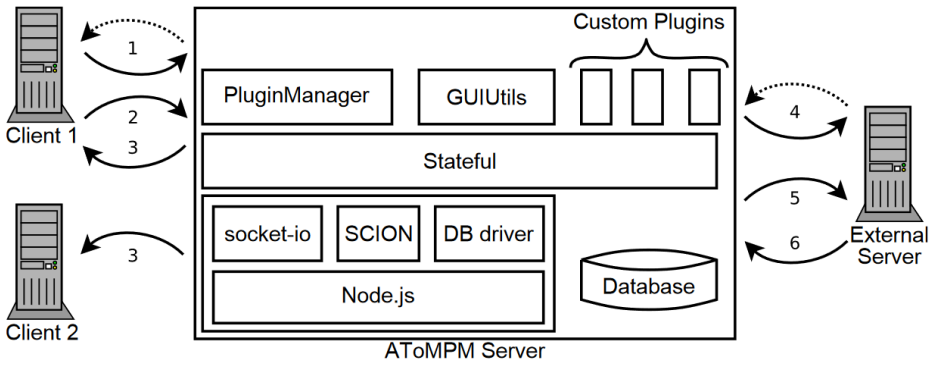

Figure 3.1: AToMPM Architecture. Reprinted from Ref. [51]

languages can be integrated. This is further supported by a plugin called "PluginManager" explicitly depicted in Figure 3.1, which handles the loading and unloading of other plugins [51].

AToMPM excels in its extensibility features. It offers complete support for constraint definition, custom validation, and custom element type definition. Adopters can define constraints either globally or locally within model element types using JavaScript [6]. The framework also supports meta-modeling, allowing users to define metamodels primarily using class diagrams [5]. However, AToMPM lacks integration capabilities with external tools or libraries [6]. Whilst the framework does not support the import of (meta-)models, it supports three export options: saving and using the AToMPM native ".model" extension, exporting to an Ecore file usable in Eclipse and exporting to a SVG file [6]. None of the mentioned file formats can be considered a widely adopted data interchange format.

The framework boasts a robust set of editing capabilities including intuitive node and connection creation, as well as element grouping and nesting [6] [5]. Additionally, AToMPM supports model transformations and advertises it as one of it's main features. It allows for the definition of model transformations within a modeling language, for example, supporting general transformations from one language to another and even very specific use cases such as petri net simulations [6]. It mostly shows deficiencies in regards to convenient layouting features, namely automatic layouting and snapping with model element resizing and connection routing also being supported only partially due to their implementations seeming somewhat unintuitive for users [6].

AToMPM includes an artifact browser that functions similarly to a file explorer, allowing users to create, move, and delete folders and files. Apart from this, it lacks useful model management features like model search and version control. The framework does support undo and redo functionality, although it is only accessible via buttons in the GUI, which might be an issue from an accessibility perspective.

Some user experience enhancements, such as interactive diagram elements and zoom/pan functionality, are offered [6]. It also supports real-time collaboration, allowing multiple

users to work simultaneously on the same model [6]. However, it falls short in areas like inline editing and accessibility features such as keyboard shortcut support [6] and assistive technologies. The framework provides basic on-screen guidance through tool tips, but lacks comprehensive contextual help [6]. It would be virtually impossible to navigate the features of AToMPM successfully or at least inefficient by relying solely on these tool tips. Considering this, the documentation seems indispensable. The documentation is extensive and up-to-date [6] but unfortunately the frameworks development appears to have halted, with the newest update released in February 2023. Moreover, community support is limited and the last issue on the GitHub page was opened in August 2023 [7].

3.2 JointJS

JointJS is a JavaScript library used to build both diagrams and diagramming applications. It is fully open source with a solid foundation and some very useful features for building modeling editors [34]. As the name implies, it is based on JavaScript and uses Scalable vector graphics (SVG) for rendering graphics. Moreover, it depends on additional technologies such as jQuery, Lodash, and Backbone.js [30]. Unlike AToMPM JointJS does not include features such as an already implemented modeling editor with a fully formulated GUI or model persistence. Instead, it is better categorized as a coding library that provides tools and building blocks to build unique modeling solutions. One could argue that this allows for more freedom of extension and customizability but also requires additional effort compared to frameworks akin to AToMPM. In addition to providing predefined elements and features for creating diagrams, JointJs also allows creating reusable custom diagram elements, as well as an extensive API for customization [30].
The developers behind JointJS also offer a commercial version of this framework called "JointJS+" with many more features built on top of the open source version [34]. The main difference between the two versions is the amount of manual and additional work needed for building fully fleshed out modeling editors. Whilst implementing a lot of the features in the open source version requires developers to create fully custom implementations on top of the functionality the open source version provides, many of these features are already supported "out of the box" in the commercial version that streamlines this process [28].
We mentioned that this thesis will only focus on open source platforms since those tend to be more relevant for research. Despite that, we have decided to include the commercial version in this analysis since it is so closely related to JointJS and will also give an interesting perspective on the proposed benefit of a paid framework.

JointJS provides a foundation for extensibility, with partial support for constraint definition and custom element type definition [28]. Adopters are able to fully realize these features by using provided components, such as the event listener and adding custom implementations [34]. The commercial version, JointJS+, offers complete support for these features, including a validation component for more streamlined handling of constraints [34]. Both versions have considerable integration capabilities, supporting

various JavaScript frameworks and libraries, as well as both versions possessing built-in import and export capabilities [28].
Some editing capabilities are already fully implemented in the open source version, like connection creation [54] and nested elements [34]. However, it lacks node creation workflows and convenient features, such as element grouping. JointJS+ addresses these limitations, providing advanced functionalities like drag-and-drop, node creation and a selection plugin for operating on multiple elements simultaneously [28]. Both versions support connection routing [54] and automatic layouting, with JointJS+ offering a wider selection of layouts [28].
The biggest weakness of the library is its lack of model management support. The open source version does not support any of the features in this category, whereas the commercial version only offers undo and redo functionality [28].
JointJS+ significantly improves the user experience compared to the open source version. Although both support interactive diagram elements [54] [34], JointJS+ adds features such as inline editing, zoom and pan, and keyboard shortcuts. Unlike the open source library, JointJS+ also offers comprehensive tools that can be configured to provide contextual help and tool tips. This includes plugins and components such as the popup and tool tip components designed to enhance user interaction and provide necessary information directly within the modeling environment [28]. A big strong point of this library is without a doubt its relevancy and active maintenance. The library is frequently updated with new releases that not only include bug fixes, but also introduce new functionality [13] and forums or social media pages like GitHub or Twitter are easily accessible and lively [14]. The documentation for this library is not only easy to access and find, but also very detailed, covering most aspects of the library [34]. Additionally, JointJS provides many demo applications that showcase all features included in the open source and commercial versions of the library, along with their corresponding implementations [37].

3.3 React Diagrams

Like JointJS, React Diagrams is better described as a code library. As stated in the official documentation of React Diagrams, the library was actually heavily inspired by JointJS. In general, the library was designed for creating customizable and extendable diagramming tools. The developer intended it to be used in conjunction with the JavaScript framework React and wrote the implementation mainly in TypeScript [42]. One notable feature and significant difference from JointJs is the use of HTML to render nodes as opposed to SVG, making use of its unique properties such as being able to embed input fields or drop-down menus. The library provides adopters with default models and widgets that can be easily extended and tailored to ones own domain [4].

React Diagrams possesses close to no built-in extensibility features. It was designed with ease of integration in mind as shown in a demo application that extends the core library with an auto-layouting feature [45]. It excels in custom element type definition, allowing adopters to extend three base model types: nodes, ports, and links. Each can

be customized in appearance and behavior [17]. While it does not natively support any others, most features could be implemented by extending the codebase. For instance, it supports model serialization and deserialization, which could be used to implement an import and export feature[45].
The library provides robust editing capabilities. All basic features, namely node creation, connection creation and element grouping, are fully implemented. Additionally, the library offers full support for connection routing, allowing users to freely adjust routing paths. However, it lacks built-in support for many other features, for example automatic layouting [45], nested elements and model element resizing [3]. Furthermore, no feature involving any kind of model processing is supported, meaning no built-in model transformation or code generation.
React Diagrams is primarily focused on the front-end diagramming experience and lacks built-in model management features. Whilst features like an artifact browser and model search are most likely not easily realizable, others like a version control or undo and redo feature are very much feasible. The library is easily integrated with the version control system GIT and custom implementations for an undo/redo feature have already been proposed in a GitHub issue [55].
User experience falls a bit short, as many features are unsupported. It fully supports interactive diagram elements, allowing for the integration of buttons and other interactive HTML elements within nodes. Zoom and pan functionality is also fully supported [45]. That said, accessibility features such as customizable keyboard shortcuts and assistive technologies are missing, as well as important feedback mechanisms such as on-screen guidance and error messages. React Diagrams has seen less frequent updates recently, with a focus on bug fixes and maintenance rather than new feature development. The documentation is described as a work in progress and may not be comprehensive [4]. The community surrounding this project used to be very active, but ever since development has slowed down, so has the community activity. Questions still get posted from time to time, but do not always get answered [42].

3.4 Sirius Web

Sirius Web is an open source project under the Eclipse Foundation that provides a modeling workbench to create custom graphical modeling tools. It allows users to define their own modeling languages and design their own editors for various domains. With Sirius Web, users can easily create and customize visual representations, define validation rules, and specify the behavior of their models. It is a powerful tool for creating specialized modeling environments tailored to specific needs. Sirius Web is a low-code modeling editor comparable to a framework like AToMPM, as it also provides an already fleshed out modeling environment with complex tolling and functionality. Extending and customizing it requires much less coding then aforementioned libraries [52]. The platforms technical stack holds up to today's standards including modern technology such as Spring, React, PostgreSQL and GraphQL. Additionally, it is worth noting that Sirius Web is a sub-project of Eclipse Sirius and closely related to its predecessor Sirius

Desktop that was later remade to be a web-based platform [47].

The extensibility of this framework is one of the best we found in our analysis. It supports all features we considered at least partially. Constraint definition is fully supported through so-called "domain models", allowing adopters to define restrictions on attributes and relationships between model elements with basic validation functionality [52]. Custom element type definition is robust, with adopters able to define nodes with various attributes and relationships in the domain model, and further customize them in a so-called "view model". The platform also fully supports meta-modeling, enabling the creation of domain-specific languages through class diagram syntax [46]. Apart from this importing and exporting of models is convenient and Sirius Web also allows for certain types of integration. This pertains mostly to integration with an Integrated development environments (IDE). Using Sirius Web in conjunction with other applications, libraries and so on is made a lot easier by utilizing webhooks. Unfortunately, those are only available as an enterprise feature [52]
It has built-in support for editing capabilities such as node and connection creation, element grouping and nested elements [46]. Automatic layouting can be enabled and configured by adopters when creating a new DSML [52]. Sirius Web also supports model transformation, allowing multiple representations of the same model, including diagrams, forms [46] and others [43]. However, it lacks connection routing capabilities, which may limit flexibility in complex diagrams.
The framework includes a fully featured artifact browser referred to as a "project browser" that lists all models and associated artifacts. The artifact browser offers a basic model search functionality [52], but more elaborate features like tagging and labeling are missing. Model management features some might consider very essential are missing. This includes version control features like a change history or branching and undo/redo functionality, which could be all be limitations for complex modeling projects.
In regards to user experience Sirius Web offers an impressive coverage of features. It fully supports interactive diagram elements, inline editing [52], and zoom and pan functionality. It even allows real-time collaboration, meaning simultaneous editing of the same model with changes being reflected instantly for the respective other user. It lacks keyboard shortcuts and assistive technology support, which is a detriment to accessibility. On-screen guidance is well implemented, with tool tips and context-aware menus that can be customized by adopters [46]. The Sirius Web GitHub page is very active, with frequent commits to its repository. The documentation is very detailed and regularly updated for new versions of the framework. Both the issue and discussion section of the Sirius Web GitHub are still seeing almost daily activity [25]. Additionally, developers of Sirius Web can be directly contacted for support [16].

3.5 Sprotty

Sprotty is a diagramming framework that can be used to create complex diagrams or entire modeling tools bolstering their own domain-specific elements and logic. Despite

not having a GUI like other tools previously referred to as frameworks, like AToMPM and Sirius Web, Sprotty still cannot be classified as just a code library [21]. Unlike libraries that usually provide specific functions that can be called upon as needed, Sprotty boasts a comprehensive architecture, an extensive feature set, and the ability to shape an applications structure [31]. The architecture of Sprotty allows for both rich-client applications and applications with client and server interplay. Compared to some of the other frameworks discussed in this paper that are less code-heavy, Sprotty demands considerable technical knowledge from its adopters [21]. It leverages technologies such as TypeScript for client-side implementation, Java and Node.js for server-side architecture and SVG for rendering diagrams. A key feature of Sprotty is its ability to extend the Language server protocol (LSP). The LSP is a client-server approach that replaces today's mostly monolithic Integrated development environments (IDE) that are used in the field of software development. It has mainly been developed to counteract the n-to-m complexity of having to individually integrate every programming language into every code editor by decoupling the language-specific logic from the usability-centric text editor and therefore lowering the complexity [8]. LSP establishes a uniform protocol that standardizes the communication between a language client (e.g., an IDE like Eclipse) and a language server (e.g., for a programming language like Java). The language client only needs to be able to interpret and understand the protocol instead of the specific programming language. Likewise, the language server can focus on language support and does not need to consider the specifics of a respective IDE [8]. The LSP separates language-specific logic (handled by a "language server") from the editor-specific implementation, meaning language-specific features only have to be implemented once. Sprotty's ability to extend the LSP to integrate graphical modeling with textual languages allows for hybrid modeling approaches [31].

Sprotty provides moderate extensibility features. While it does not have a dedicated constraint definition tool, adopters can allow or disallow user interaction on certain elements such as the ability to select, drag, or connect elements [21]. More intricate constraint definition can be achieved by integrating with Xtext, a separate framework [31]. The same goes for validation capabilities. Although natively only supported in minor ways [26], full support can be achieved by integrating with Xtext [31]. Adopters can implement custom elements with application-specific properties. Views can be defined for each element based on different SVG element types with individual CSS styling [21]. Sprotty excels in its integration capabilities. Integration with Xtext, Langium, the Language Server Protocol, VS Code and Theia to name a few are supported by Sprotty with existing packages to ease the process [21]. Features such as meta-modeling and imports are not implemented.

The framework provides a lot of means to edit and modify models as part of its core implementation. Especially structural modifications and layout adjustments are mostly covered, with the only missing features being connection creation [21] [2] [48] [50] and element resizing [39]. All features, including model processing (model transformation, artifact representation and code generation) are only supported via integration [31].

Sprotty is among the frameworks that is not responsible for model persistence. Considering this, most model management features are not inherently supported. Only when integrated with for example an LSP they are realized [31]. Undo and redo functionality does not mandate model persistence and is supported by Sprotty [21]. A wide variety of user experience features are part of the core functionality of the framework. It fully supports interactive diagram elements, inline editing, and zoom and pan. Keyboard shortcuts are well supported and customizable [21]. The framework also offers a command palette [49] that can be used to search for model elements within an open model [48]. Other features such as collaborative features or assistive technologies are not supported. Adopters are given tools to implement on-screen guidance for users, but it is not part of the core functionality. Error messages are also supported and can be fully customized by adopters utilizing the decoration feature on specified model elements [21]. To this day, Sprotty still receives regular updates with the newest minor version being released on April 9th 2024. Almost all updates are accompanied by a change log detailing the improvements and new features [26]. The documentation gives adopters a good introduction into the framework, is comprehensive and for some topics goes into a fair amount of detail [21]. Lastly, the Sprotty project spans over four repositories and all of them seem to have an actively maintained issue section for adopters to ask questions and give suggestions [26].

3.6 GLSP

GLSP is a web-based client-server framework that can be used to develop modeling editors with great emphasis on customizability and extensibility of the editors functionality. As detailed by De Carlo et al. [18], the platform is split up into three parts: The client, which is responsible for rendering the diagram and providing editing tools for the operations defined by the server. The server, which is responsible for things such as loading, interpreting, and editing diagrams according to the rules of the graphical diagram language and the source model, which is a representation of the model in a format that can be easily saved [22].
GLSP is also closely related to Sprotty and even uses the framework in its clients implementation [9]. Whilst Sprotty mainly focuses on rendering and interaction with diagrams on the web, giving users direct control over the visual elements [21], GLSP builds upon this, providing additional tools and a protocol for servers to easily communicate with diagram editors [22]. This communication is realized through a sequence of specialized operations that handle model updates, bounds computation, and action processing [18].

Extensibility-wise, GLSP performs similar to Sprotty. The only significant difference being constraint definition and validation. GLSP offers full support for both features, making it possible for adopters to define constraints mostly on the server-side. It is even possible to define quick fixes, which can be accessed via the context menus that are supported by GLSP editors, to resolve constraints that have been violated following validation [22]. Server-side validation rules can be customized and per default an editor built upon GLSP renders a so-called “Palette” that, among other functions, includes

a button to validate the model. When pressed, the model is validated and the user gets immediate feedback through the validation markers, which in themselves are also customizable [22]. On the server-side, it is possible to customize existing base elements that are already preconfigured like basic nodes or edges with, for example, additional attributes or relationship behaviors or configure completely new element types [22]. From the client the adopter is able to customize the rendering of the elements limited by SVG and CSS supported operations [36].

Considering the editing capabilities we analyzed as part of this research, GLSP performed exceptionally well. It fully supports all structural modifications, such as basic functionality, namely node creation [36] or more advanced capabilities such as nesting nodes [36] [23]. Layout adjustment capabilities are also fully supported, covering connection routing, automatic layouting, model element resizing and snapping [36]. Despite that, model processing is a weak point, with model transformation being unsupported and code generation only supported by integrating GLSP with another framework like the Eclipse Modeling Framework (EMF) cloud. An example implementation can be found in the "ecore-glsp" project, where ecore files can be used to generate corresponding Java files [23, 27]. Regarding different artifact representations, users are always able to right click a model file and choose whether to open the file in a code editor (text-based) or the rendered version. The editor could even be set up in a way where opening a model file results in a split view with the textual and rendered view of the model being displayed side by side synced to one another. GLSP is also one of the frameworks that has fully implemented refactoring support. For example, if set up correctly, the labels (names) of certain elements can reference other elements, which are synced with the original element and propagating changes [36].

GLSP, like Sprotty, does not concern itself with model persistence, implying that most model management features will not be supported. Whilst model search as well as tagging and labeling are not supported by the framework, an artifact browser and version control can be used together with GLSP when integrated with an IDE, which is very feasible because of the frameworks outstanding integration capabilities.

The framework also boasts a rich user experience, again supporting almost all relevant features analyzed. It supports interactive diagram elements, inline editing, and zoom and pan functionalities [36]. Developers have recently added new features for a model element search, real-time collaboration and improved accessibility, allowing users to access features like model element search, canvas moveability, zoom and resizing with only their keyboard. The features listed are still in an experimental state, mostly with limited integration support regarding specific IDEs [19]. Keyboard shortcuts are also well supported, especially when integrated with IDEs. Contextual help and feedback mechanisms are fully covered, meaning on-screen guidance, navigation logic [36] and error messages [22] are all implemented in the core of the framework. Furthermore, GLSP is actively maintained, with regular updates [24] and a recent major release (v2.0) introducing new experimental features [19]. The documentation is fairly detailed and almost completely up to date. Pages in the documentation that have not yet been adjusted for the newest release of the framework are marked as such [22]. In addition, the

projects GitHub page is very active with a dedicated discussion section where adopters can ask questions, raise concerns and browse the archive of previous support tickets [24]. They also have an email address and a support forum for more specialized cases [15].

3.7 WebGME

WebGME is a web- and cloud-based collaborative modeling tool that focuses on domain-specific modeling that allows users to define DSMLs and corresponding domain models. It is similar to aforementioned tools AToMPM and Sirius Web providing a fleshed out GUI. WebGMEs implementation is written in JavaScript, uses MongoDB for model storage and provides a REST API for language-independent access. Some of its distinctive features include prototypical inheritance that fuses meta-modeling with modeling, scalability, extensibility and its built in version control.[38]
Much of the research on WebGME is based on the generic GUI, accessible by logging into your WebGME account via the official homepage. It is the default model editor that contains all the core features of WebGME and combines them in an intuitive GUI. Based on the documentation it is possible to implement a completely new GUI and integrate it with a frontend framework [33].

Based on our metrics WebGME performs great in terms of extensibility, fully supporting all features we considered. Firstly, it has a fully implemented solution for meta-modeling, which allows adopters to easily define DSMLs via class diagram-based models [38]. Constraints can also be defined implicitly through the meta-model and are enforced by restricting user interaction. Constraints not definable through meta-rules can otherwise be specified for each element using JavaScript code [33]. The validation of said constraints is also supported by WebGME. If meta-model changes conflict with already existing models, meta-rules can be validated project-wide via a button in the user interface. Custom JavaScript constraints are not validated by the user interface, but can also be checked by pressing a button in the user interface. The framework also has great integration capabilities. As stated in the documentation, if only the WebGME client API is used, it can function as a library that can be integrated with any frontend framework [33]. Additionally, a framework is provided that makes it easy to integrate user-defined plugins. WebGME also offers its own format for importing and exporting files (.webgmexm) but by utilizing the “WebGME-JSON” repository adopters and users can also import and export models in a JSON format [59].
The framework not only performs well when it comes to extensibility, but also supports quite a lot of editing features. Node creation is fully supported with an intuitive drag-and-drop interface. Connection creation is partially supported, requiring some configuration in the metamodel [1]. WebGME fully supports nested elements [33] and element grouping. While manual connection routing is limited, the framework offers three different automatic routing styles [38]. However, it lacks support for model element resizing. Model transformations are not supported by the framework itself, but there is other ways to implement custom ones. Plugins can be used to define them [33].

There is also an npm package called "webgme-transformations" that can be integrated into WebGME and used as a language to define model transformations [58]. Apart from transformations, WebGME also supports different and even fully custom model representations, so called "visualizations". In addition to pre-configured visualizations, adopters can define different types of visualizer that present the model structure in different ways [33]. Code generators are available by taking advantage of the WebGME plugin framework and using one of the interpreter plugins available or writing a completely new one [33]. Lastly, refactoring is also fully supported. Changes made in one location propagate to all other related areas of the model. For instance, alterations within the meta-model, such as adjusting the rendering specifics of a model element, ripple through the entirety of the model.
The framework does concern itself with model persistence, making model management features one of its strong suites. An object browser is available within its model editor with three different views. The composition view is very similar to a standard folder view of the models inside the project. Despite that, this feature is only partially supported, since the user is not able to organize the models himself. The structure and organization is automatic and can not be adjusted within the model editor. This could hypothetically be circumvented by replacing the generic GUI of the model editor with a new GUI [33]. The artifact browser also enables users to search for specific models. Models can be filtered by their names or meta-type. It is also possible to hide certain artifacts, such as abstract nodes or connections. A standout feature is its built-in Git-like version control system, which supports version history, commits, branching, and merging [33].
Regarding user experience, WebGME provides a fair amount of features. Interaction-based features are for the most part at least partially supported, with inline editing, zoom and pan functionality [61], and real-time collaboration being fully supported. Accessibility is not mentioned in any of the documentation [33] as a major focus, therefore, no assistive technologies are employed to enhance accessibility. Furthermore, keyboard shortcuts are only partially supported, with only generic shortcuts being available and no ability to customize and implement new ones. Contextual help features are also missing support, with no sophisticated on-screen guidance like helpful tool tips and no implementation for navigation logic. Error messages are shown to users whenever appropriate and display descriptive messages. Examples found during testing are validation errors or conflicts that occur during a merge. In terms of current relevance of the framework, it appears that the main repository of WebGME is not updated very frequently, but it is also not completely abandoned [60]. Documentation is very easily accessible via the home page of WebGME [56]. Unfortunately, all official social media accounts associated with WebGME, such as YouTube seem mostly inactive [57]. Similarly, the GitHub issue section is also inactive with many unresolved tickets [60].

Table 3.1: Legend

+	Full support
0	Partial support
-	No support
*	Not supported by core

Table 3.2: Extensibility

	Constraint Definition	Custom Validation	Custom Element Type Definition	Meta-Modeling	Integration Capabilities	Import	Export
AToMPM	+	+	+	+	-	-	0
JointJS	0	-	0	-	+	+	+
JointJS+	+	+	+	-	+	+	+
React Diagrams	*	-	+	-	+	*	*
Sirius Web	+	0	+	+	0	+	+
Sprotty	0	0	+	-	+	-	0
GLSP Eclipse	+	+	+	-	+	-	0
WebGME	+	+	+	+	+	+	+

Table 3.3: Editing Capabilities 1

	Node Creation	Connection Creation	Element Grouping	Nested Elements	Connection Routing	Automatic Layouting
AToMPM	+	+	+	+	0	-
JointJS	-	+	-	+	+	+
JointJS+	+	+	+	+	+	+
React Diagrams	+	+	+	*	+	*
Sirius Web	+	+	+	+	-	+
Sprotty	+	-	+	+	+	+
GLSP Eclipse	+	+	+	+	+	+
WebGME	+	0	+	+	0	+

Table 3.4: Editing Capabilities 2

	Model Element Resizing	Snapping	Model Transformation	Artifact Representations	Code Generation	Refactoring Support
AToMPM	0	-	+	0	*	-
JointJS	-	+	-	*	-	*
JointJS+	+	+	-	*	-	*
React Diagrams	-	+	-	-	-	-
Sirius Web	+	+	+	+	*	+
Sprotty	-	+	*	*	*	-
GLSP Eclipse	+	+	-	+	*	+
WebGME	-	+	*	+	*	+

Table 3.5: Model Management

	Artifact Browser	Model Search	Tagging and Labeling	Version Control	Undo/Redo
AToMPM	+	-	-	-	+
JointJS	-	-	-	*	-
JointJS+	-	-	-	*	+
React Diagrams	-	-	-	*	*
Sirius Web	+	0	-	-	-
Sprotty	*	-	-	*	+
GLSP Eclipse	*	-	-	*	+
WebGME	+	+	-	+	+

Table 3.6: User Experience 1

	Interactive Diagram Elements	Inline Editing	Zoom and Pan	Model Element Search	Real-time Collaboration	Keyboard Shortcuts	Assistive Technologies
AToMPM	+	-	+	-	+	0	-
JointJS	+	-	-	0	-	-	0
JointJS+	+	+	+	+	-	+	0
React Diagrams	+	0	+	*	-	-	-
Sirius Web	+	+	+	0	+	-	-
Sprotty	+	+	+	+	-	+	-
GLSP Eclipse	+	+	+	0	0	+	0
WebGME	0	+	+	0	+	0	-

Table 3.7: User Experience 2

	On-Screen Guidance	Navigation Logic	Error Messages	Regular Updates	Documentation	Community Support
AToMPM	0	-	+	-	+	0
JointJS	-	-	*	+	+	+
JointJS+	+	-	+	+	+	+
React Diagrams	-	-	-	0	0	0
Sirius Web	+	0	0	+	+	+
Sprotty	*	0	+	+	+	+
GLSP Eclipse	+	+	+	+	+	+
WebGME	-	-	+	0	+	-

CHAPTER 4

Evaluation

After having analyzed the feature support of each framework, we can draw several conclusions and reflect on the insights gained. We can not only attempt to evaluate the frameworks against each other, but also contemplate current modeling framework developments like vital missing features and areas that should get further development.

4.1 Critical Feature Gaps

Our results highlighted several features that lacked support in many frameworks, although they are arguably important or extremely beneficial for adopters, users, or both.
For instance, meta-modeling is unsupported by many frameworks. While some tools like AToMPM, WebGME and Sirius Web offer robust meta-modeling capabilities, others, namely GLSP, Sprotty, JointJS and React Diagrams, lack native support for this feature. Although this feature is not necessary for defining DSMLs, it removes complexity, makes languages easily readable and interpretable and improves extensibility. Frameworks that do not support this feature, often require adopters to implement all modeling rules in code, adding a prerequisite to said tools [10].
Most tools lacked native support for both model transformation and code generation. They are considered essential for MDE, allowing for the conversion between different model representations or translating model files to code [10]. The lack of built-in model transformation and code generation capabilities may require users to implement custom solutions or integrate external tools, potentially complicating the modeling workflow [51].
As noted during the analysis part of the thesis, many frameworks reviewed do not have built-in model persistence, making model management obsolete for the most part. This means that features like tagging, labeling, and advanced search capabilities are often missing or limited across the evaluated frameworks. This suggests that many tools are focused more on model creation and editing than on long-term model management and organization. Comprehensive model management support becomes increasingly important

for larger and more complex modeling projects [11]. Framework adopters that implement modeling tools might have to come up with solutions for model persistence.

Depending on one's requirements, real-time collaboration might also be a crucial feature, that unlike other less complex features could be difficult for adopters to implement themselves. Support for this feature makes it much easier to work in a distributed environment [38]. JointJS, React Diagrams and Sprotty do not support real-time collaboration.

Navigation logic is another feature that becomes increasingly useful when working with large and complex models. Using predefined relationships and structures makes traversing and understanding such models significantly easier [11]. This feature cannot be found in tools like AToMPM, JointJS, and React Diagrams, while others such as Sirius Web and Sprotty only provide partial support.

The aforementioned features highlight some of the gaps in current modeling frameworks. In order to create more comprehensive modeling frameworks, developers should consider implementing these features, especially if the tools are being designed with complex modeling tasks or large-scale projects in mind.

4.2 Standout Feature Support

Some features are especially well implemented by certain frameworks. They are either outright the only tool supporting a feature or might have the most fleshed out implementation.

For example, WebGME is the only framework among all evaluated tools to have a built-in version control system, supporting branching and merging, therefore enabling collaborative modeling [38]. The integration of version control directly into the modeling environment simplifies the development process and provides a more cohesive user experience [38].

Depending on the use case, real-time collaboration is a highly valued feature that allows simultaneous model changes. Sirius Web, GLSP [19] and WebGME all offer robust solutions that fulfill this role. This feature can be particularly important for team-based modeling projects that benefit greatly from direct collaboration.

Whilst we generally only reviewed frameworks with the ability to somewhat define domain-specific concepts, not all tools approached this matter the same. Some frameworks rely on in-code solutions to define domain-specific logic and others tried streamlining this process by utilizing meta-models. AToMPM, Sirius Web and WebGME adopted such an approach [5] [46] [38].

Lastly, AToMPM posses the most complex and formulated model transformation capabilities. This, combined with its multiparadigm approach to modeling, makes AToMPM especially relevant for advancing research in MDE.

4.3 Framework Deficiencies

Not only did the analysis highlight particularly well rounded frameworks but also tools with a lacking assortment of features. JointJs for example hides many of its most essential features behind a subscription model making them only available in the commercialized version[28]. Some of its shortcomings, such as cumbersome constraint and model element definition [34], lack of some essential editing capabilities and narrow focus on user experience [28], make it unfit for full-fledged and complex modeling projects.
The reason React Diagrams is similar in this regard to JointJs is probably because it takes a lot of inspiration from it [42]. Although it improves somewhat in terms of basic editing capabilities [45] [17], it misses advanced features such as model transformation, different artifact representations and refactoring support. It also lacks essential extensibility features and additionally falls short in terms of user experience features.
Despite that, JointJs and React Diagrams are lightweight alternatives to more complex frameworks, making them very easy to integrate into existing web applications.

4.4 Areas for Future Development

Our research shows multiple features that are unsupported by a majority of frameworks or underdeveloped. In this section we want to highlight features we deem important for the future of MDE and which would benefit greatly from further development. One such feature is accessibility, or more specifically custom keyboard shortcuts and assistive technologies. Many of the evaluated frameworks lack proper support for these features. Modeling tools need to evolve in this regard in order to support users with diverse needs. Recent research by Sarioglu [44] emphasizes the increasing importance of accessibility in modeling tools, highlighting that about 16% of the world's population live with some form of disability. GLSP shows the most promise in this area, introducing new experimental features that aim to help improve accessibility, but as our research indicates, there is still considerable room for improvement [19].
Many frameworks also do not concern themselves with model persistence and therefore do not support a majority of model management features. This in itself is not a problem since different frameworks cover different use cases. Generally, model persistence can be adapted using external tools or frameworks built specifically for this purpose [31]. Still, some teams might prefer this process to be streamlined and incorporated into the modeling tool solution. Furthermore, even frameworks including model persistence often lack features such as advanced search capabilities, namely support for tagging and labeling of artifacts. Big and complex modeling projects rely on the ability to reliably organize and retrieve models.
Lastly, some frameworks seem to have limited import and export functionality, often not supporting any commonly used data interchange format [6]. This limitation greatly restricts model interoperability and therefore cross-framework compatibility. We should aim to improve the support for existing interchange formats to further interoperability between frameworks and prevent vendor lock-in [10].

In conclusion, the evaluation reveals strengths and weaknesses of the considered frameworks and although tools such as GLSP, Sirius Web, and WebGME offer very comprehensive feature sets, we have also discovered considerable gaps. Gaps shown such as accessibility, model management support and framework compatibility can be seen as future opportunities for innovation and development in upcoming iterations of the frameworks or entirely new tools.

CHAPTER 5

Recommendation

The choice of a modeling framework depends heavily on the specific requirements of a project, the expertise of the team, and the intended use case. Based on our analysis, we can group the frameworks to help readers identify which tool best suits their needs.
If a fleshed out solution with comprehensive feature coverage is a priority, three frameworks come to mind. WebGME is a great all-in-one solution providing great extensibility features, having built-in model persistence, version control and well supported real-time collaboration features. Its biggest shortcomings are in the area of user experience, like for example the lack of community support and infrequent updates [60]. It is a great fit for large-scale collaborative modeling projects. Similarly, Sirius Web is also a good choice for an all encompassing solution. The framework relies on a low-code approach to define domain-specific concepts, which means that using the tool does not require substantial coding knowledge. It offers comprehensive extensibility features such as meta-modeling, good editing capabilities, built-in model persistence and rich user experience, like collaboration capabilities [52]. Despite this, it has limited integration capabilities [52], no sophisticated version control and an insufficient focus on accessibility. In addition to having an especially good coverage of the features considered, GLSP bridges some of these gaps, with exceptional integration features [22] [27] and makes strides toward better accessibility [19]. It should be noted that, unlike the previously mentioned frameworks, GLSP does not include model persistence but supports it through integration.
Some frameworks are designed specifically for research and prototyping purposes. AToMPM labels itself as a research framework [7]. This claim is supported by its substantial support of model transformation and its multi-paradigm modeling approach [51]. It provides the ability to easily experiment with novel modeling concepts, making it suitable for academic and research projects.
When choosing a framework, the ease of use and required level of coding knowledge should be considered. Sprotty and GLSP are best suited for modeling projects that require very specific and fine-grained solutions. Their highly configurable nature makes them more complex and assumes a high level of expertise from adopters [21] [22].

On the other hand, frameworks such as Sirius Web and WebGME decided to opt for a low-code solution. They rely on an intuitive user interface, that for example lets users model domain concepts in a meta-model instead of having them defined through code-specified rules [46] [38].
If a project requires flexible and lightweight solutions, both JointJs and React Diagrams can be considered. Both frameworks are not overly complex and rely on modern web technologies, promoting easier integration with existing web application. JointJs offers a good foundation with basic diagramming capabilities [28] that can be extended by manually implementing additional features [34]. React Diagrams is similar, although it was created specifically to be used in conjunction with the React JavaScript framework [4] and unlike JointJs it lacks proper maintenance and community activity [42].
Maintenance, community activity and continuous development are important considerations for choosing a framework. Our research has shown some tools to be vary of in this regard. AToMPMs development has halted with its most recent update in February 2023. Soon after, community activity also came to a stop, with the latest GitHub issue having been opened in August 2023 [7]. Even though its relevance for research cannot be understated, its long-term viability is questionable. React Diagrams has seen a considerable decrease in development activity, with updates focusing primarily on bug fixes instead of new feature additions [7]. Documentation is labeled as work in progress [6] and, as development regressed, so has community activity [7]. Lastly, while WebGME has not been completely abandoned, updates have become more and more infrequent [60]. Official social media accounts have also become inactive [57], which might indicate a decrease in community activity.
Generally, when selecting a framework, consider the following factors:

- Project Scope and Complexity: The scale and complexity of a project play a pivotal role in the selection of frameworks. Large-scale projects with multiple teams and complex requirements may require frameworks that offer extensive collaboration features and scalability. On the other hand, smaller or simpler projects might benefit from lightweight solutions that focus on ease of use and quick setup.

- Required Features: Identify essential features like meta-modeling, model transformation, specific editing capabilities, or real-time collaboration. Choose a framework that offers native support or robust integration options for those features.

- Technical Expertise: Assess the team's familiarity with different technologies and the level of coding required for customization. Decide on a framework that aligns with the assessed expertise.

- Active Maintenance and Documentation: Choose actively maintained frameworks with comprehensive documentation and active communities to facilitate troubleshooting and knowledge sharing.

Choosing the right modeling framework requires careful consideration of individual requirements and project constraints. By grouping frameworks based on their strengths

and weaknesses, this chapter assists readers in navigating the landscape of web-based modeling tools. Factors such as extensibility needs, desired level of customization, integration requirements, and emphasis on specific features play a crucial role in the selection process. The insights presented here help readers to identify the framework that best aligns with their goals, enabling them to leverage the power of modeling for enhanced software development and research endeavors.

CHAPTER 6

Conclusion

This thesis conducts a comprehensive feature-based analysis of extensible modeling frameworks and libraries, with a focus on web-based platforms. By examining the key features of various frameworks, this paper sheds light on their respective strengths, limitations, and applicability in different scenarios. The classification and evaluation of these tools aim to assist both researchers and general users in selecting the most suitable framework for their specific needs, as well as to guide future developments in the field.

The analysis highlights that some frameworks such as Sirius Web and GLSP standout due to there exceptional feature coverage. They excel in providing robust extensibility, including support for domain-specific modeling language creation and integration capabilities. These features make them particularly appealing for complex and customizable modeling tasks. On the other hand, we also analyzed lightweight libraries such as React Diagrams and JointJS offering simplicity and high customizability, catering to projects requiring streamlined, focused solutions. However, certain shortcomings were observed, such as limited support for advanced model management functionalities, collaborative editing, and accessibility features. These deficiencies suggest that, while the field has matured significantly, there remains considerable room for growth to address broader and more diverse user needs.

For practitioners, this research highlights the critical importance of aligning the choice of framework with project-specific requirements. For researchers, identified gaps, particularly in areas such as accessibility enhancements, advanced collaboration tools, and comprehensive integration with external systems, represent valuable opportunities for further investigation and innovation. Bridging these gaps can lead to frameworks that are more accessible, scalable, and adaptable to emerging trends in software development and engineering.

Once again we would like to emphasize that this study was not exhaustive. Many frameworks and libraries, including potentially innovative solutions, were not covered in this thesis. Furthermore, the analysis focused on the technical aspects of frameworks

and did not focus on their performance in real-world applications. Future research could expand on these aspects.
In conclusion, this thesis provides a structured and detailed exploration of web-based modeling frameworks that offers valuable insights. While the work contributes significantly to understanding the current landscape, the rapidly advancing nature of the field makes ongoing research to develop frameworks that are not only feature-rich but also user-centric, accessible, and capable of supporting the diverse challenges of modern software development necessary.

List of Figures

2.1 Feature model notation 3
2.2 Top level feature model 4
2.3 Extensibility 5
2.4 Editing capabilities 8
2.5 Model management 12
2.6 User Experience 1 14
2.7 User Experience 2 17

3.1 AToMPM Architecture. Reprinted from Ref. [51] 22

List of Tables

3.1 Legend 32
3.2 Extensibility 32
3.3 Editing Capabilities 1 32
3.4 Editing Capabilities 2 33
3.5 Model Management 33
3.6 User Experience 1 34
3.7 User Experience 2 34

Acronyms

API Application programming interface. 7, 23

AToMPM A Tool for Multi-paradigm modeling. vii, 21–23, 25, 27, 30, 35, 36, 39, 40

DSML Domain-specific modeling language. vii, 1, 5–7, 21, 26, 30, 35

EMF Eclipse Modeling Framework. 29

GLSP Graphical Language Server Platform. vii, 28, 29, 35–39, 43

GUI Graphical user interface. 21, 23, 27, 30, 31

IDE Integrated development environments. 26, 27, 29

LSP Language server protocol. 27, 28

MDE Model-driven engineering. 21, 35–37

SVG Scalable vector graphics. 22–24, 27, 29

WebGME Web-based Generic Modeling Environment. vii, 30, 31, 35, 36, 38–40

Bibliography

[1] *04. Tutorial - Connections and Ports.* URL: https://www.youtube.com/watch?v=QSthYt-j6oI&list=PLhvSjgKmeyjiwOzNmQq8OY6KG7Cz-y-1d&index=5 (visited on 05/05/2024).

[2] *Add context menu support - Issue 139 - eclipse-sprotty/sprotty.* URL: https://github.com/eclipse-sprotty/sprotty/issues/139 (visited on 04/24/2024).

[3] *Any plans on 'group-nodes' support? Issue 27 projectstorm/react-diagrams.* URL: https://github.com/projectstorm/react-diagrams/issues/27 (visited on 03/27/2024).

[4] *Architecture Questions | React Diagrams.* URL: https://projectstorm.gitbook.io/react-diagrams/about-the-project/architecture-questions.

[5] *AToMPM - Features | PPT.* URL: https://de.slideshare.net/slideshow/atompm-features/26363260 (visited on 03/24/2024).

[6] *AToMPM Documentation — AToMPM 0.10.0 documentation.* URL: https://atompm.readthedocs.io/en/latest/ (visited on 03/25/2024).

[7] *AToMPM/atompm: A Tool for Multi-Paradigm Modeling.* URL: https://github.com/AToMPM/atompm (visited on 03/25/2024).

[8] D. Bork and P. Langer. "Catchword: Language Server Protocol: An Introduction to the Protocol, its Use, and Adoption for Web Modeling Tools". In: *Enterprise Modelling and Information Systems Architectures: International Journal of Conceptual Modeling* 18.9 (2023), pp. 1–16. DOI: 10.18417/emisa.18.9.

[9] Dominik Bork, Philip Langer, and Tobias Ortmayr. "A vision for flexible glsp-based web modeling tools". In: *IFIP Working Conference on The Practice of Enterprise Modeling*. Springer. 2023, pp. 109–124.

[10] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[11] Erwan Breton and Jean Bézivin. "Model driven process engineering". In: *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. IEEE. 2001, pp. 225–230.

[12] Frank Budinsky et al. *EMF: Eclipse Modeling Framework*. 2nd. The Eclipse Series. Addison-Wesley, 2009. ISBN: 0321331885,9780321331885. URL: http://gen.lib.rus.ec/book/index.php?md5=1747769a6ad818f458e3e2c9a25ab7c8.

[13] *clientIO/joint: A proven SVG-based JavaScript diagramming library powering exceptional UIs*. URL: https://github.com/clientIO/joint (visited on 05/22/2024).

[14] *Community of diagramming experts and geeks – JointJS*. URL: https://www.jointjs.com/community (visited on 05/24/2024).

[15] *Contact - Eclipse Graphical Language Server Platform*. URL: https://eclipse.dev/glsp/contact/ (visited on 04/22/2024).

[16] *Contact us - Obeo*. URL: https://www.obeosoft.com/en/contact (visited on 04/09/2024).

[17] *Customizing | React Diagrams*. URL: https://projectstorm.gitbook.io/react-diagrams/customizing (visited on 03/26/2024).

[18] G. De Carlo. "Integrating extended visualization and interaction functionalities into language server protocol based modeling tools". Diploma Thesis. Technische Universität Wien, 2022. URL: https://doi.org/10.34726/hss.2022.99900.

[19] *Diagram Editors Boosted: Collaborative, Testable and Accessible diagrams with Eclipse GLSP - YouTube*. URL: https://www.youtube.com/watch?v=RBbI_QBzwl4&t=34s (visited on 04/22/2024).

[20] Hanns-Alexander Dietrich et al. *Developing graphical model editors for meta-modelling tools-requirements, conceptualisation, and implementation*. 2013.

[21] *Docs | Sprotty*. URL: https://sprotty.org/docs/ (visited on 04/25/2024).

[22] *Documentation - Eclipse Graphical Language Server Platform*. URL: https://eclipse.dev/glsp/documentation/ (visited on 04/22/2024).

[23] *eclipse-emfcloud/ecore-glsp: ecore-glsp*. URL: https://github.com/eclipse-emfcloud/ecore-glsp (visited on 04/23/2024).

[24] *eclipse-glsp/glsp: Graphical language server platform for building web-based diagram editors*. URL: https://github.com/eclipse-glsp/glsp (visited on 04/22/2024).

[25] *eclipse-sirius/sirius-web: Reusable frontend and backend components for Sirius Web*. URL: https://github.com/eclipse-sirius/sirius-web (visited on 04/08/2024).

[26] *eclipse-sprotty/sprotty: A diagramming framework for the web*. URL: https://github.com/eclipse-sprotty/sprotty (visited on 04/24/2024).

[27] *Ecore tools in the cloud - behind the scenes - YouTube*. URL: https://www.youtube.com/watch?v=YQyaCR_V5zc (visited on 04/24/2024).

[28] *Explore the features of our diagramming library – JointJS*. URL: https://www.jointjs.com/features (visited on 05/23/2024).

[29] Robert France and Bernhard Rumpe. "Domain specific modeling". In: *Software and System Modeling* 4 (Feb. 2005), pp. 1–3. DOI: 10.1007/s10270-005-0078-1.

[30] Marco Geue et al. "Entwicklung eines grafischen Editors für XProc-Pipelines mit dem SVG-basierten JavaScript-Framework JointJS". MA thesis. Hochschulbibliothek, Hochschule Merseburg, 2019.

[31] Philipp-Lorenz Glaser. "Developing Sprotty-based Modeling Tools for VS Code". In: (2022).

[32] Soichiro Hidaka et al. "Feature-based classification of bidirectional transformation approaches". In: *Software & Systems Modeling* 15 (2016), pp. 907–928.

[33] *How to build a Design Studio with WebGME — WebGME 1.0.0 documentation*. URL: https://webgme.readthedocs.io/en/latest/index.html# (visited on 05/05/2024).

[34] *Joint API (v4.0) - JointJS Docs*. URL: https://docs.jointjs.com/ (visited on 05/23/2024).

[35] Philip Langer Jonas Helming Maximilian Koegel. *Accessibility in Diagram Editors with Eclipse GLSP*. 2024. URL: https://eclipsesource.com/blogs/2024/02/07/accessibility-in-diagram-editors-with-eclipse-glsp/ (visited on 07/10/2024).

[36] Philip Langer Jonas Helming Maximilian Koegel. *Web-based diagram editor features in Eclipse GLSP*. 2021. URL: https://eclipsesource.com/blogs/2021/02/10/web-based-diagram-editor-features-in-eclipse-glsp/ (visited on 04/24/2024).

[37] *Kitchen Sink App – Demo applications & examples*. URL: https://www.jointjs.com/demos/kitchen-sink-app (visited on 05/23/2024).

[38] Miklós Maróti et al. "Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure." In: *MPM@ MoDELS* 1237 (2014), pp. 41–60.

[39] *Moveable ports and live node rescaling - Issue 268 - eclipse-sprotty/sprotty*. URL: https://github.com/eclipse-sprotty/sprotty/issues/268 (visited on 04/24/2024).

[40] Jon Oldevik et al. "Framework for model transformation and code generation". In: *Proceedings. Sixth International Enterprise Distributed Object Computing*. IEEE. 2002, pp. 181–189.

[41] Jakob Pietron et al. "A study design template for identifying usability issues in graphical modeling tools." In: *MoDELS (Workshops)*. 2018, pp. 336–345.

[42] *projectstorm/react-diagrams: a super simple, no-nonsense diagramming library written in react that just works*. URL: https://github.com/projectstorm/react-diagrams (visited on 03/27/2024).

[43] Axel Richard. *Sirius Web 2024.1 - The official voice of the Obeo experts*. 2024. URL: https://blog.obeosoft.com/sirius-web-2024-1 (visited on 04/06/2024).

[44] A. Sarioğlu, H. Metin, and D. Bork. "How Inclusive Is Conceptual Modeling? A Systematic Review of Literature and Tools for Disability-Aware Conceptual Modeling". In: *Conceptual Modeling: 42nd International Conference, ER 2023, Lisbon, Portugal, November 6–9, 2023, Proceedings*. Ed. by J. P. A. Almeida et al. Springer, 2023, pp. 65–83. DOI: 10.1007/978-3-031-47262-6_4.

[45] *Simple Usage - Demo Simple Storybook*. URL: http://projectstorm.cloud/react-diagrams/?path=/story/simple-usage--demo-simple (visited on 03/26/2024).

[46] *Sirius Web 101: Create a Modeler With No Code - YouTube*. URL: https://www.youtube.com/watch?v=p_tDEzGtS0o (visited on 04/06/2024).

[47] *Sirius WebHome*. URL: https://eclipse.dev/sirius/sirius-web.html.

[48] *sprotty/examples at master - eclipse-sprotty/sprotty*. URL: https://github.com/eclipse-sprotty/sprotty/tree/master/examples (visited on 04/24/2024).

[49] *sprotty/packages/sprotty/src/features/command-palette/command-palette.ts at master - eclipse-sprotty/sprotty*. URL: https://github.com/eclipse-sprotty/sprotty/blob/master/packages/sprotty/src/features/command-palette/command-palette.ts (visited on 04/23/2024).

[50] *sprotty/packages/sprotty/src/features/move/snap.ts at master - eclipse-sprotty/sprotty*. URL: https://github.com/eclipse-sprotty/sprotty/blob/master/packages/sprotty/src/features/move/snap.ts (visited on 04/24/2024).

[51] Eugene Syriani et al. "AToMPM: A web-based modeling environment". In: *Joint proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*. 2013, pp. 21–25.

[52] *The Obeo Studio Documentation*. URL: http://docs.obeostudio.com/2023.12.0/help_center.html (visited on 04/06/2024).

[53] Thomas Thum et al. "Abstract Features in Feature Modeling". In: *2011 15th International Software Product Line Conference*. 2011, pp. 191–200. DOI: 10.1109/SPLC.2011.53.

[54] *Tutorials - JointJS Docs*. URL: https://resources.jointjs.com/tutorial (visited on 05/23/2024).

[55] *Undo / History Functionality Issue 391 projectstorm/react-diagrams*. URL: https://github.com/projectstorm/react-diagrams/issues/391 (visited on 03/26/2024).

[56] *WebGME Homepage*. URL: https://webgme.org/ (visited on 05/08/2024).

[57] *WebGME YouTube channel*. URL: https://www.youtube.com/channel/UC1cPQP4jjsXRhpXUnoPZQWg (visited on 05/08/2024).

[58] *webgme-transformations | listen.dev*. URL: https://verdicts.listen.dev/npm/webgme-transformations (visited on 05/06/2024).

[59] *webgme/webgme-json: Simple JSON domain that allows the manipulation and use of json based documents in other projects*. URL: https://github.com/webgme/webgme-json (visited on 05/05/2024).

[60] *webgme/webgme: Web-based Generic Modeling Environment*. URL: https://github.com/webgme/webgme (visited on 05/08/2024).

[61] *zoom +/- buttons with collapsible zoom slider - Issue 907 - webgme/webgme*. URL: https://github.com/webgme/webgme/issues/907 (visited on 05/05/2024).