Incremental Formalization for Informal Architectural Diagramming

Malvina Latifaj, Jan Carlson, Antonio Cicchetti, Robbert Jongeling, Ifrah Qaisar
Mälardalen University, Västerås, Sweden
{firstname.lastname}@mdu.se

Keywords: Informal Architectural Diagrams, Incremental Formalization, Flexible Modeling

Abstract: Informal diagrams created in general-purpose diagramming tools are widely used in software architecture because they are quick to produce and easy to share. However, the lack of constraints in such tools often yields inconsistent notations and ad-hoc conventions, which in turn invite misinterpretation when diagrams are read outside their original context. Dedicated modeling languages and environments can mitigate these issues but are frequently resisted due to steep learning curves and disruptive adoption costs. Building on the flexible modeling paradigm, this paper proposes an approach to the lightweight formalization of informal diagrams. Grounded in observed industrial challenges and prior work on flexible modeling, we derive a set of design principles and instantiate them in an approach realized as a Draw.io plugin. We propose an approach that addresses challenges from industrial settings by enabling practitioners to introduce structure incrementally into the informal diagrams they already create, thereby helping resolve notational inconsistency and clarify meaning. Moreover, we show how our approach satisfies the guiding flexible modeling principles by introducing these model-like benefits without compromising the accessibility and speed of informal diagramming as valued in practice. The contribution enhances clarity and consistency within familiar workflows and lays the groundwork for subsequent capabilities such as rapid dissemination of conventions, enterprise-level aggregation, and additional quality checks should organizations choose to adopt them.

1 INTRODUCTION

In the early stages of software design, architects find informal diagrams useful for exploring architectural alternatives, investigating emerging design concepts, and facilitating collaborative discussions (Jongeling et al., 2024; Hasselbring, 2018; Wüest and Glinz, 2011). Their inherent flexibility and minimal overhead, make them particularly well-suited for the rapid ideation and negotiation that characterize initial exploration. The utility of these diagrams, however, often extends beyond this phase. Subsequently, they may be used later in development by a wider audience and for different purposes than initially foreseen (Baltes and Diehl, 2014). Because these diagrams are often created in unconstrained, general-purpose diagramming tools, valued for their flexibility, but where no standard or convention is enforced, authors freely shape them according to personal preferences and interpretations. While this freedom supports quick exploration, it also leads to ambiguity when the diagrams are later read or reused by others. As these diagrams travel beyond their original context, the flexibility valued during brainstorming must be balanced with the clarity required for long-term communication.

To impose rigor, organizations might turn to formal or semi-formal modeling languages and modeling environments. Although these languages can offer precise syntax and semantics, their adoption is often hindered by significant barriers: a substantial learning curve, considerable modeling effort, and strict conformance demands are frequently perceived as incompatible with the iterative nature of modern agile development (Hutchinson et al., 2014; Whittle et al., 2013; France and Rumpe, 2007), leading practitioners to resist them or underutilize them.

To bridge the gap between informal diagramming and formal modeling, the flexible modeling paradigm seeks to balance the expressive freedom inherent in informal diagrams with mechanisms for incremental formalization as a design matures (Jongeling and Ciccozzi, 2024; Guerra and de Lara, 2018). This idea is also highlighted in recent discussions, which assert that postponing structural constraints is critical for the success of informal methods (Bucchiarone et al., 2021).

While flexible modeling provides a strong foundation, practical challenges still exist. Existing solu-

tions require the adoption of tools that pull architects away from their preferred, general-purpose diagramming environments (Di Rocco et al., 2023; Guerra and de Lara, 2018; Izquierdo et al., 2013; Wüest et al., 2013; Cho et al., 2012). However, this requirement reintroduces the very workflow disruptions and adoption hurdles that cause practitioners to resist formal modeling languages and environments. The key challenge, therefore, is to enhance rather than replace the established, informal tools and workflows that architects use in their daily practice. The goal is to introduce just enough incremental formalization to informal diagramming, in order to yield tangible modeling benefits, without imposing the rigidity of strict conformance or metamodel commitments.

To this end, we engage with software architecture teams at a partner company that make frequent use of informal software architecture diagrams. Through this partnership, we elicit a set of core challenges, which we then enrich with our cumulative insights from other industrial experiences and the existing literature on flexible modeling. This analysis forms the basis for this paper's two main contributions: (i) an approach to lightweight formalization of informal diagramming that provides model-like benefits without sacrificing flexibility, and (ii) a practical implementation of this approach as a Draw.io plugin designed for seamless integration into existing tools and workflows. The reader is encouraged to have a look at our replication package[1] including the implementation and a demo video.

The remainder of this paper is organized as follows. Section 2 presents the industrial context, the challenges elicited, and the resulting principles that guided our solution's design. Section 3 introduces the proposed approach, and Section 4 details its implementation. Section 5 details how the principles and features of our approach resolve the identified challenges, and Section 6 discusses the implications and limitations of the approach. Section 7 reviews related work and Section 8 concludes the article.

2 INDUSTRIAL CONTEXT

We investigate software architecture practices through a collaboration with a large, European Original Equipment Manufacturer (OEM). Through five working sessions with two principal architects, we analyze a set of informal diagrams and documentation workflows from various teams to identify their recurring challenges. The findings highlight that their architectural practices are fundamentally shaped by two key factors: i) organizational heterogeneity, with documentation responsibility distributed across diverse teams, and ii) a preference for lightweight tooling, such as Draw.io, that favors expressive freedom over rigorous modeling governance. The interplay of these factors creates three main challenges, which in turn serve as the foundation for a set of guiding principles that a successful solution must embody.

[1] https://github.com/MLJworkspace/Drawio_Formalization_Plugin/tree/main

2.1 Challenges in Current Architectural Practice

The context outlined in the previous section (i.e., organizational heterogeneity coupled with a reliance on unconstrained diagramming tools) defines the landscape of challenges faced by architects at our partner company. We detail them below.

C1. Conceptual ambiguity. In the studied industrial setting, we observe that teams sometimes lack a shared understanding of what diagram elements represent. Element names are often informal or overloaded, and a shared vocabulary of concepts such as component, service, or database is not always available, which could leave an element's role unspecified. Where a vocabulary exists, it occasionally becomes a source of confusion when it is outdated, incomplete, or applied inconsistently, which leads to differing interpretations. Relationships between elements are another area where intent is not always explicit. Connectors often lack a clear description of what they represent, and intended direction is sometimes absent, so readers infer the nature of the interaction from context. We also note that this issue of meaning can persist even when visuals are consistent. A diagram may present a uniform appearance for its elements, yet readers may still be uncertain about what those items are meant to be. The practical effect is that identity and role are inferred on a case-by-case basis, which increases cognitive load, slows reviews, and can lead to divergent readings across teams.

C2. Notational variability. Beyond conceptual ambiguity, we also notice notational variability, which concerns how elements are shown and how appearance relates to concepts. Notational variability stems from a work culture that prioritizes speed and immediate communicative intent over strict adherence to a shared visual standard. We observe that architects, under delivery pressure, often select notations based on convenience or personal preference, using them as

they see fit to make a point in the moment. This practice leads to a highly ad-hoc and inconsistent visual syntax across diagrams, worsened by the frequent absence of a shared legend to map concepts to their visual representation, which makes interpretation a matter of guesswork. As an architect quotes "*...notation leaves too much to interpretation, which will result in a different outcome from what is expected"*. This notational issue is distinct from conceptual ambiguity as even when a team's concepts are well-defined, representing them inconsistently makes the diagrams difficult to read and recognize. In practice, this imposes a significant cognitive load that impedes the efficiency of reviews and analysis.

C3. Adoption friction. Despite recognizing the need for improvement, teams are hesitant to adopt new tools or practices. Architects are motivated to reduce notational variability and conceptual ambiguity so that diagrams are clearer and less open to misinterpretation. They also see that greater consistency could enable useful capabilities such as automated checks and impact analysis. At the same time, past experiences with cyclical adoption and abandonment of tools have created skepticism toward solutions that promised long term benefits but add workflow overhead. Architects valued the expressive freedom and low overhead of their current diagramming tools, which they view as essential for meeting delivery commitments. As a result, proposed changes are more likely to gain traction when they work inside existing environments and allow teams to adopt improvements gradually and on their own terms.

Collectively, these challenges undermine the utility of informal diagrams at scale. They impede the construction of a cohesive enterprise view, as source diagrams are often incomplete, contradictory, or incompatible. This uncertainty hinders reasoning about system-wide dependencies and decision-making. Furthermore, the reliance on tacit knowledge creates barriers for newcomers, prolonging onboarding and increasing the risk of design errors.

2.2 Guiding Principles

Building on the observed challenges, insights from discussions with other industrial partners, and prior work on flexible modeling (Jongeling and Ciccozzi, 2024; Guerra and de Lara, 2018), we sought to define a set of principles to guide the design of the solution. Informed especially by the third challenge (i.e., adoption friction), we introduce a *cross-cutting constraint* on everything that follows. The mechanisms we propose, must *preserve the flexibility and low overhead that practitioners already have*. This constraint is not a separate principle. Rather, it conditions how all principles are realized and how their supporting features behave. With this constraint in mind, we define the following principles.

P1. Incremental formalization. The solution must support the incremental formalization of informal diagrams.

This principle recognizes that architectural design often begins with informal diagrams to facilitate rapid communication and iteration. Imposing strict formalism at this early stage can impede progress and creative discourse. A more effective approach introduces clarity selectively and incrementally as decisions solidify, thereby helping to counteract conceptual ambiguity and notational variability over time. Well-understood elements can be formalized early, while evolving ones may remain informal until clarity emerges or a specific need for precision arises. Any imposed formalism must be malleable, allowing for easy modification or removal to prevent premature commitment. This incremental pathway aligns with the natural evolution of design thinking, supporting rapid sketching when ideas are fluid and enabling incremental formalization as they stabilize.

P2. Informative conformance checking. The solution must provide conformance checks that point out deviations from agreed conventions and inform authors without interrupting their work.

The purpose of this principle is to balance flexibility with guidance. Without consistency checks against agreed conventions – the outcomes of incremental formalization – notational inconsistency and conceptual ambiguity will persist. Yet strict, interruptive enforcement deters adoption. The approach should therefore offer informative, non-blocking feedback that flags departures from the conventions and allows architects to defer, prioritize, or dismiss fixes at their discretion. Feedback should identify the exact element or relation involved and explain the issue plainly, so authors get useful guidance without sacrificing productivity or autonomy.

P3. Convention sharing and reuse. The solution must enable the capture, distribution, and reuse of defined conventions.

To extend consistency beyond a single diagram, the solution must enable reusable convention sets. Teams need a way to define, once, the shared choices about element types, their standard visual forms, and

basic constraints, and then propagate these choices across other diagrams. For this to be practical, these conventions must be simple to package, import, and apply, both to new diagrams and to existing ones for alignment. Such capability can reduce variation in visual notation and uncertainty about element meaning at scale, while saving architects time and lowering the cognitive effort required to interpret diverse diagrams.

3 APPROACH OVERVIEW

Grounded in the defined principles, this section presents our proposed approach, which supports two workflows. First, for *existing diagrams*, our approach acts as a repair tool. It allows teams to retrospectively standardize how diagrams are defined and used without having to recreate them from scratch. Second, for *new diagrams*, previously defined conventions can be applied from the outset, so diagrams are consistent and coherent from the start. The following subsections describe the approach's features.

3.1 Incremental Formalization

What typing means in our setting. Our goal is to enable incremental formalization of otherwise informal diagrams. In formal modeling, an abstract syntax defines a metamodel of concepts and their connections, a concrete syntax specifies the visual notation, a mapping links concepts to their visual forms, and semantics supplies well-formedness rules and constraints (Kleppe, 2007). We operationalize this through a lightweight, user-driven typing process. A type acts as a minimal meta-element representing an architectural concept and is linked to a specific visual representation. Each type captures three essentials: (i) a name that identifies the concept, (ii) a fixed visual representation that ensures notational consistency, and (iii) a set of constraints currently applicable to connections.

F1. Defining types from examples. In contrast to traditional meta elements defined upfront, our types emerge bottom up from concrete elements in the diagram. The definition is intentionally manual because user intent is the primary source of truth in informal diagrams and visual cues alone are often too ambiguous for reliable automated inference. To define a type, the author selects an *exemplar* element that carries the visual properties expected to be shared by all instances of the intended type. The author then specifies the *type name* and two further specifications:

- A *matching rule* that identifies other elements to be treated as instances of the type based on visual attributes such as shape, fill color, or line style.
- A *propagation profile* that defines the exemplar's visual attributes that are transferred to matched elements when visual alignment is applied.

To keep the vocabulary of types visible, the link between each type and its visual representation must be explicit. The approach ensures this by automatically creating and maintaining a *legend* that lists every defined type together with its visual representation.

F2. Automatic typing by example. Once a type is defined, the user can apply it to other elements by automatic propagation. The two specifications created at definition time drive this process. The matching rule selects the elements that qualify as instances of the type. The propagation profile enforces type assignment and restricts visual propagation to the subset of the exemplar's appearance attributes marked for propagation in the profile. Propagation can be run for the entire diagram to ensure consistency at scale or invoked for a single or a set of elements when a targeted update is needed. The system does not infer types on its own. Automation executes only the rules that the user has declared and these rules can be refined at any time.

F3. Typed instantiation. After types have been defined, authors can create new elements that are already typed. The legend remains the declarative record of the type vocabulary, while a separate actionable library mirrors its contents as an interactive palette. Users select a type from the library and place a correctly styled and typed element directly onto the diagram. New instances immediately adopt the recorded appearance of the chosen type, similar to palettes in modeling environments.

F4. Dynamic typing. As understanding evolves, an element may need a different type. Retyping replaces the previous type without recreating the element. The element keeps its identity and connections and immediately adopts the appearance defined by the new type's propagation profile. Each element holds exactly one type at a time. Retyping occurs in two ways. First, a user can reassign a typed element to another existing type. Only that element changes. Second, a typed element can serve as an exemplar to define a new type. The exemplar is reclassified, and the new type can then be propagated to all other elements that satisfy its matching rule.

F5. Type modification. Over time, types themselves may need revision. The approach supports changing a type's name, appearance, and constraints. To avoid confusion, local edits to typed instances are separated from changes to the type definition. A visual change made directly on an instance is treated as a local deviation when it affects a property governed by the type's propagation profile. In contrast, genuine modifications to a type are performed centrally in the legend. Updates made in the legend, such as renaming a type or adjusting its appearance, are then propagated to all instances. This ensures that changes are applied consistently while local deviations remain visible.

3.2 Informative Conformance Checking

What conformance means in our setting. In our approach, conformance is the degree to which a diagram follows the shared choices a team has made about the set of existing types, their visual appearance, and constraints. Conformance is assessed against three criteria: (i) the local vocabulary and recorded visual notation, (ii) type-specific constraints that state which connections are allowed, and (iii) a set of predefined, generic rules that apply to all diagrams. Validation is on demand and non-intrusive. Architects run checks when they choose, and drawing, editing, and saving continue uninterrupted. Findings appear as pointers on the relevant elements or connections, each with a brief explanation, so users decide what to address and what to defer.

F6. Conformance checking. For targeted and non-blocking feedback, the approach carries a conformance check, evaluating whether elements apply the local vocabulary and the recorded visual notation. When an element's visual appearance diverges from the recorded notation, the approach permits the drift, but issues a warning. Second, it checks connections between elements against their assigned types, clarifying which pairings are allowed or disallowed and flagging any that contradict the intended structure. Third, it applies generic rules that improve consistency across all diagrams. Typical findings include untyped elements, missing descriptive labels, and isolated nodes that are not connected to the rest of the diagram. Findings point to the affected element and briefly explain why it was flagged.

F7. Statistics. While the aforementioned conformance checks identify issues for individual elements, a holistic perspective helps teams assess the overall state of a diagram and guide subsequent development. To facilitate this, the approach provides a statistics dashboard that provides a quantitative, aggregate overview of the entire diagram. This view updates dynamically as the diagram is modified and reports on key metrics, including total element and concept counts, the ratio of typed to untyped elements, the number of orphan nodes, and instance totals for each concept. These metrics provide quantitative insights into the diagram's structure and maturity, helping to direct refactoring and design efforts. For example, a low proportion of typed elements may serve as an indicator for an early stage, exploratory design. Similarly, a heavily unbalanced distribution of elements across concepts can highlight a need for a focused architectural discussion.

3.3 Convention Sharing and Reuse

F8. Convention bundle. Within a single diagram, local conventions can keep types and notation consistent. When work spans multiple diagrams or teams, that clarity depends on a portable convention bundle that can be shared and adopted. In essence, the bundle records the common ground: agreed vocabulary, visual conventions, permissible relations, and any scoped restrictions, and it can evolve through versioned updated as teams refine their practice. Adopting the bundle in a new diagram provides a shared starting point where teams can begin with the same vocabulary and conventions, avoiding one-off definitions and reducing setup friction. Adopting it in an existing diagram enables on-demand validation and optional alignment to the shared conventions, so users can reconcile unintended inconsistencies.

4 IMPLEMENTATION

We implemented our approach as a plugin for the Draw.io diagramming tool, which is used for architecture description by our industrial partner and, presumably, other organizations. To support adoption and reproducibility, the plugin and a demonstration video are available in an anonymized repository [1]. Figure 1 summarizes the plugin's architecture which the following sections detail.

4.1 Draw.io Diagram Files

A Draw.io file is serialized as an XML document where each diagram is encoded as an `mxGraphModel`. The fundamental unit is the `mxCell` element, which represents all graphical entities. Geometric properties such as position and dimensions are specified in

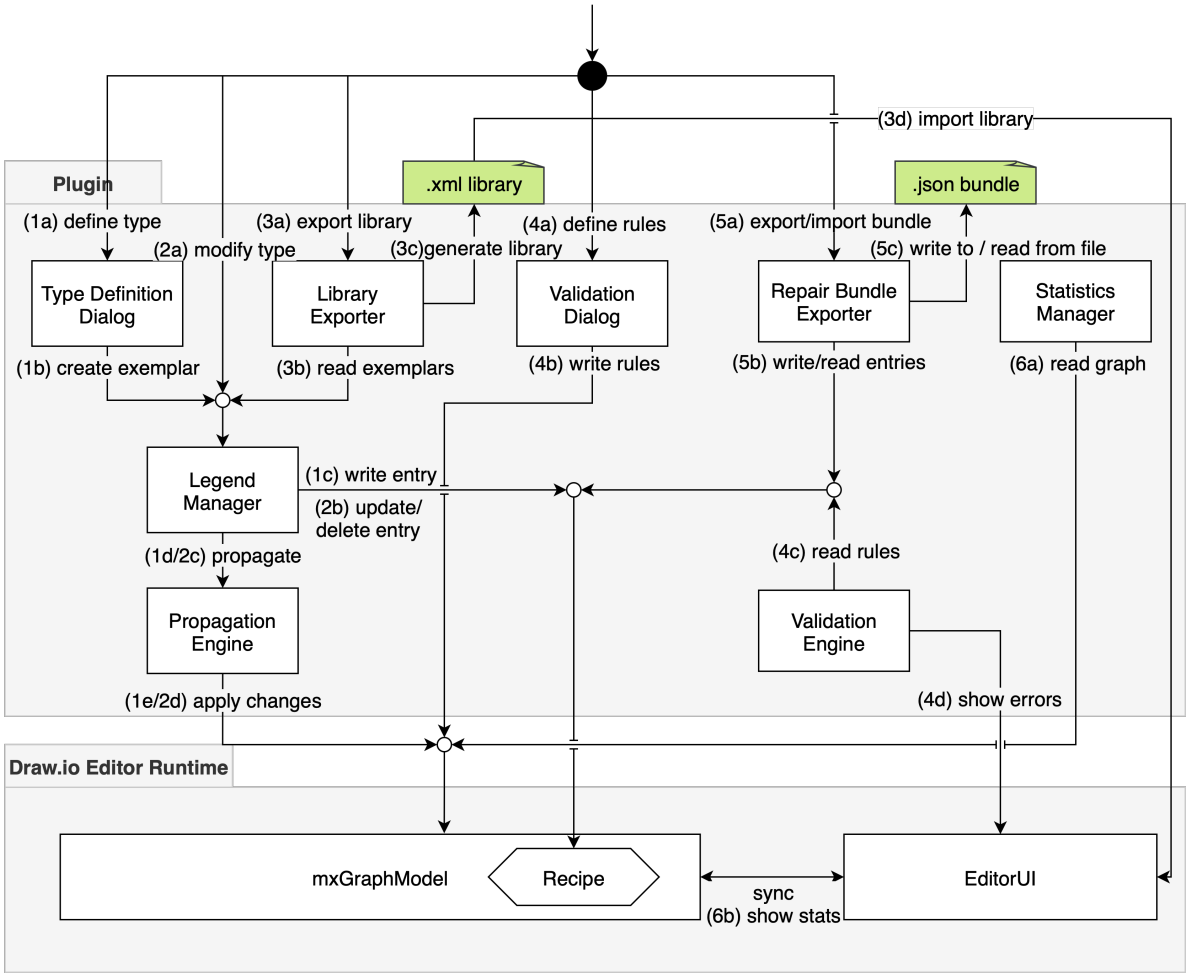

Figure 1: Workflow of technical steps underlying the implementation

a child `mxGeometry` element. An element's visual appearance is defined by a semicolon-delimited style string of `key=value` tokens, while its value attribute stores either a visible label or a user object for arbitrary metadata.

At runtime, the editor deserializes this XML into an in-memory `mxGraphModel` that is bound to an `mxGraph` view. The editor's user interface operates by issuing edits to this model. The model then emits change and undo events that ensure the visual canvas and the underlying XML data remain synchronized. Our plugin integrates into this process by operating exclusively on the native `mxGraphModel`, traversing it to read and write the properties of `mxCells`, such as their styles, geometry, and connectivity references. Because we only store information in places the editor already supports (cells, styles, geometry, and optional user-object attributes), one can open, edit, and save the file in plain Draw.io and everything remains intact across both the XML on disk and the interactive editor UI.

4.2 Type Definition and Propagation

To define a type, the user can invoke `Generate Type` on an existing element (Figure 1;1a). At this point, the dialog requires defining the type name and two specifications: the *matching rule* (visual attributes that qualify other elements as instances), and the *propagation profile* (styling attributes enforced on those instances when applying the type). Upon *Save*, that element becomes the exemplar of a named type (1b), and the workflow proceeds as follows:

- The *Legend Manager* writes a type entry in what we refer to as a *recipe* that is embedded in the same `mxGraphModel` file (1c). At a high level, the recipe functions as a map from type names to their visual definitions. It records the type's name, its kind (shape or edge), the specifications, and a

serialized appearance.

- It creates or updates a visible exemplar in the diagram's legend in the EditorUI. If a legend does not exist, it creates one. The exemplar is a standard cell with the type's style applied and a type-identity attribute stored as a metadata.

Save and Propagate command hands over work to the *Propagation Engine* (1d), which (i) resolves the chosen scope, (ii) matches instances of the type (i.e., matching rule), and (iii) applies *delta* updates as only attributes marked as propagable by the user in the dialog (i.e., propagation profile) are written key-by-key, while non-propagable fields are preserved. All edits are batched in a single `mxGraphModel` transaction (1e) and can be reversed with a single undo step.

The *Type As* command can be invoked on one or more selected elements. It allows typing only those selected elements by presenting the user with a list of all existing types. Once a type is chosen, the system assigns it to the selected elements and applies its propagation profile. Any local styling that conflicts with the type's properties is overridden, while non-conflicting properties are preserved. The command can be used both to assign a type to previously untyped elements and to retype already typed ones.

4.3 Legend-driven Type Management

Once a type is defined and added to the legend, the legend becomes the single point of control for refining that type and distributing changes. Modifying a type in the legend (e.g., changing name, fill color, line styling) (2a), updates its details in the recipe (2b), and, when propagated, applies those updates to all elements of that type(2c:2d). The legend supports the following actions:

- **Rename type.** Change the type's name. The corresponding recipe entry and the `typeName` metadata on all affected elements are updated.
- **Remove type.** Delete the type from the legend. The recipe entry is removed and any element bound to it has its `typeName` cleared. The diagram's current appearance is preserved and local styling remains.
- **Propagate.** Apply the type's current name and visual appearance from the legend to all instances in the diagram, according to its propagation profile. Text labels, positions, and non-propagable attributes are preserved.
- **Configure propagation profile.** Open the propagation panel to (re)select which attributes are propagable. The selected flags are saved in the recipe. Future propagation only applies those attributes to matching instances.
- **Export legend as library.** On export (3a), the *Library Exporter* queries the *Legend Manager*'s recipe catalog (3b) and converts each recipe to a minimal `mxGraphModel` snippet packaged as an `<mxlibrary>` XML file (3c). The resulting library mirrors the legend and can be imported into this or other Draw.io files to enable reuse.

4.4 Conformance Checking

Conformance checking is implemented as an on-demand, non-blocking process. In step (4a), users define conformance rules in the validation dialog. At present, the user-defined rules focus on constraints that specify the elements that the connectors are allowed to connect, specifically the `allowedSources` and `allowedTargets` for each connector type. These rules are then serialized together with the type definitions in the recipe (4b). When a conformance check is invoked, the validation engine loads the relevant rules from the recipe and the built-in generic checks implemented in the plugin (4c). It then performs a single pass over the diagram, evaluating each element and connection. First, the engine verifies adherence to the recipe by checking that every element's `typeName` is defined and that its visual appearance matches the recorded notation. Untyped elements are flagged, and visual drift is reported. Second, it validates type-specific connection constraints by inspecting each typed connector, retrieving the applicable `allowedSources` and `allowedTargets`, and comparing these with the actual source and target node types. Disallowed connections are reported. Third, the engine applies generic rules that support readability and reuse, including checks for missing labels, isolated nodes, and duplicate type names. Findings are displayed directly in the editor as temporary visual markers with explanatory tooltips (4d). When checking is deactivated, the diagram is restored to its original state. The process never blocks editing or saving, allowing users to continue working regardless of any unresolved findings.

4.5 Recipe Bundle Portability

The plugin keeps the *recipe* (i.e., two JSON maps for node and edge types) embedded in the diagram as attributes (`nodeTypeRecipes`, `edgeTypeRecipes`) on the value of the root model cell. The legend is a synchronized view over this recipe, hence edits to legend elements update the recipe, but both export and import operate directly on the latter. When the

Export/Import Bundle (5a) is invoked, the exporter reads the recipe from the document root (5b), extracts the JSON text from its wrapping XML structure, and generates a JSON bundle (5c). The bundle contains `shapes` and `edges` maps keyed by type name. Each type entry carries the selected propagation flags `props` that govern updates to instances, `basis` flags used for peer auto-binding, a `styleSnapshot` for visual appearance, and only for shapes, a `geometrySnapshot` with width and height. Edge types may declare connection constraints via `allowedSources` and `allowedTargets`. The exporter does not traverse the drawing but it serializes the recipe only, so the cost is linear in the number of types and independent of diagram size. In import (also 5a), the provided JSON bundle (5c) is parsed and written back into the document recipe (5b). The legend is rebuilt from the updated recipe, and imported edge constraints take effect immediately in subsequent validations.

4.6 Statistics

The statistics panel provides a live snapshot of the diagram. An event listener triggers a function that performs a single pass over all graph cells (6a). This process first filters out the entire legend subtree, layout swimlanes, node and edge labels, and text-only shapes. For each remaining cell, the engine inspects the `typeName` attribute to calculate instances for each type and to count any untyped elements. It also identifies orphan nodes by finding vertices with zero connected edges. The results are displayed in a sidebar panel that summarizes the total number of elements, distinct types, untyped elements, and orphans, with an expandable view for per-type instance counts (6b). These statistics are recalculated on each change.

5 ADDRESSING THE CHALLENGES

This section outlines how the proposed approach addresses the observed challenges. We map each challenge to the principles that guide the approach and indicate the features that operationalize those principles. Table 1 presents this mapping. The first two challenges are closely related, so we address both using the same set of features under the same principles. The third challenge follows a similar pattern, since every design choice is constrained by the need to preserve flexibility and avoid workflow disruption. Although the same sets of features apply across challenges, their contribution differs in focus. The following subsections explain each cell in more detail, showing how specific features defined under a principle contribute to overcoming a particular challenge.

5.1 Conceptual Ambiguity

C1 × P1. To resolve ambiguity and uncertainty about element identity, the approach makes implicit knowledge explicit as shared types. An architect defines a named type from an exemplar element (F1), establishing an identity, and assigning it to that element. That identity is then applied to all visually matching elements (F2), ensuring consistent interpretation rather than ad-hoc readings. During ongoing work, conceptual clarity is maintained by creating new elements as typed from the start (F3), allowing elements to adopt a new identity when their type changes (F4), and editing the type definitions centrally, so that all elements reflect the clarified conceptual intent (F5).

C1 × P2. The approach surfaces issues that obscure meaning through on-demand conformance checks (F6). These checks flag untyped or unlabeled elements, because without a declared type or label, the intended semantics remain ambiguous, and they report connections that break allowed source and target policies, because those links misrepresent relationships. A statistics view (F7) highlights areas where clarity is low, for example a high count of untyped elements, so users can focus where disambiguation is most needed.

C1 × P3. To keep a shared understanding beyond a single diagram, the convention bundle (F8) packages concept-related information including the established types and constraints into a reusable vocabulary. By importing this bundle, teams reuse the same set of types and constraints, ensuring that elements are interpreted consistently across diagrams.

5.2 Notation Variability

C2 × P1. To reduce variation in the way elements are shown, the approach establishes a corresponding notation per type by using an existing element as an exemplar (F1). The recorded notation is propagated to all matching elements to avoid inconsistencies (F2), applied automatically to newly created instances to prevent future drift (F3), updated when an element's type is changed (F4), and modified and propagated centrally via the legend so updates reach all instances at once (F5).

	P1. Incremental formalization	P2. Informative conformance checking	P3. Convention sharing and reuse
C1. Conceptual ambiguity	F1–F5	F6, F7	F8
C2. Notational variability	F1–F5	F6, F7	F8
C3. Adoption friction	F1–F5	F6, F7	F8

Table 1: Mapping of challenges to principles, showing the features through which each principle is realized to address the corresponding challenge (F1 - defining types from examples; F2 - automatic typing by example; F3 - typed instantiation; F4 - dynamic typing; F5 - type modification; F6 - conformance checking; F7 - statistics; F8 - convention bundle).

C2 × P2. To help architects identify and resolve notational drift, on-demand conformance checks (F6) compare each element's styling with the notation recorded for its type and flag mismatches. The statistics view (F7) provides an aggregate picture, such as the ratio of typed to untyped elements and counts per type, so teams can track progress and verify that visual alignment is being restored.

C2 × P3. To carry consistent notation across diagrams and over time, the convention bundle (F8) exports the full notational setup including visual appearances, matching rules, and propagation profiles so other teams can import and apply the same notation without redefining it.

5.3 Adoption Friction

C3 × P1. To ensure adoption is frictionless, the solution is built for incremental formalization that respects existing workflows. Teams can apply types to existing elements, create new typed elements, and refactor them non-destructively (F1-F5). This model avoids rigid, upfront commitments and allows structure to be introduced incrementally as its value becomes clear.

C3 × P2. Feedback is informative rather than interruptive. Conformance checks (F6) can be turned on and off on demand and appear as advisory markers with short explanations. The statistics view (F7) offers a non-intrusive progress overview. Architects choose when to check, what to fix, and what to defer.

C3 × P3. To accelerate adoption, F8 reduces setup costs by allowing teams to import reusable convention bundles. This provides immediate access to a shared convetions, offering the benefits of a structured approach with near-zero initial configuration.

How the design honors the constraint. The approach is designed so that each feature is optional and non-blocking, operates directly on existing diagrams, and keeps outputs as standard `.drawio` files. Formal and informal content can coexist, adoption can be incremental and partial, and if teams decide to stop using the approach, there is no penalty or lock-in. The diagrams remain intact and usable in the same workflow.

6 DISCUSSION

In this section, we discuss on the study's outcomes and future potential. We first address the practical limitations, and then outline the envisioned modeling extensions and advanced architectural capabilities enabled by the proposed approach.

6.1 Practical Limitations

Our approach has several inherent limitations that stem from its user-centric, flexible design. The clarity of the resulting diagrams is directly dependent on how carefully teams define and maintain their conventions. Because formalization is optional and incremental, diagrams may remain partially typed, leading to uneven precision across an artifact set. Similarly, since conventions are shared on demand rather than enforced globally, cross-team uniformity relies on voluntary adoption. Finally, our on-demand checks are advisory by design. They minimize disruption but do not guarantee full compliance. These characteristics are a deliberate trade-off, accepted to preserve the agility and low overhead of the team's existing practices. We have also deliberately excluded certain capabilities from our scope. The approach does not support rich formal semantics or heavyweight governance mechanisms. This is a conscious design choice reflecting the contexts we target, where architects choose general-purpose tools precisely for their speed and accessibility. Adding the complexity of full-fledged modeling capabilities would directly undercut these core priorities. In situations where such rigor is required, practitioners are better served by dedicated modeling tools. Finally, the generalizability (Runeson and Höst, 2009) of our findings should be interpreted with care. Our approach was designed

and motivated by observations within a specific industrial setting. Its successful adoption in other contexts will likely vary depending on organizational culture, existing governance models, and documentation maturity. The approach should therefore be viewed as a transferable design proposition rather than a universal solution, and readers are encouraged to assess fit and calibrate conventions to their own context.

6.2 Future Directions

While the approach and implementation presented in this paper focuses on the core typing infrastructure, we envision expanding this baseline to support richer modeling mechanisms and deeper architectural reasoning.

In terms of extending the modeling approach, our future work will focus on enhancing how users define and interact with system abstractions. We aim to introduce semantic zooming, which utilizes hierarchical rules to allow users to navigate varying levels of detail, similar to the C4 model, without needing to maintain separate artifacts. We also plan to explore projective multi-view modeling, which would utilize overlays to filter elements based on their type. Furthermore, we intend to investigate multi-level modeling to resolve conflicts between stakeholders. By defining a generic baseline vocabulary that can be specialized by different teams, we can maintain a single source of truth that supports distinct domain concerns.

The explicit typing system also opens significant opportunities for automated validation. We intend to leverage this metadata to perform fine-grained change impact analysis, evolving from generic connectivity checks to precise, rule-based impact assessment. Furthermore, we will investigate automated architectural analysis, where fitness functions could continuously detect structural drift or forbidden connections. Finally, we aim to support traceability through typed contracts. By defining schemas for expected links to external artifacts, the system could enforce consistency between the as-designed diagram and the as-built implementation.

7 RELATED WORK

Our work is situated within the broader field of i) formalizing informal diagrams and ii) introducing flexibility to formal modeling.

On one side of the spectrum, research focuses on example-driven, bottom-up (meta)modeling (Izquierdo et al., 2013; Cho et al., 2012; López-Fernández et al., 2019; Wüest et al., 2013). These approaches typically engage domain experts as language designers, using small, clean example sets to infer a metamodel. This process often relies on explicit, per-instance bindings (Zolotas et al., 2014) to define the language. Our objective is fundamentally different. While possible, we do not seek to elicit a metamodel as the end product, nor do we require to push practitioners towards modeling environments. Instead, we aim to bring lightweight modeling benefits into the large-scale, often inconsistent artifacts that teams already use and have accumulated over time, augmenting their work directly in their current tools.

Other efforts focus on automatic type inference or full migration to formal models. For instance, some work automatically infers formal models from domain-specific diagrams (e.g., business processes) (Mukherjee et al., 2010). Similarly, other methods aim to reduce annotation effort by resolving type omissions via inference (Zolotas et al., 2019). While we acknowledge the utility of type inference as an optional, assistive feature, the variable accuracy reported in the literature (e.g., 23% to 100% (Zolotas et al., 2019)) reinforces our focus on user-guided typing as the core mechanism.

The approach in (Jongeling et al., 2022) partially formalizes diagrams by synchronizing visual content with a textual model. Its primary limitation is the required introduction of a textual syntax, creating overhead for visual-first stakeholders. Furthermore, it relies on static properties and lacks in-editor support like validation or style propagation. In complementary work, (Kaplan and Rabelo, 2024) focus on an extraction and conversion pipeline to prepare artifacts for external analysis (e.g., in NetworkX), rather than supporting flexible modeling like our approach.

On the other side of the spectrum, research introduces flexibility to relax the constraints of formal modeling. Tools like Jjodel (Di Rocco et al., 2023; Bucchiarone et al., 2025) (a cloud platform) and Kite (Guerra and de Lara, 2018) (in an Eclipse/EMF environment) enhance flexibility, e.g., via shapeless objects or phased conformance checks, but operate entirely within their dedicated modeling toolchains. Our approach differs by extending the everyday practice of informal diagramming with incremental types and constraints, bringing lightweight formalization benefits directly to the existing tools where practitioners already draw.

In summary, our work focuses on the incremental formalization of diagrams directly within the tools practitioners already use, bypassing the need for complex solutions like metamodel inference or dedicated modeling environments. Rather than disrupting existing practices, we augment artifacts with lightweight

modeling benefits, achieving clarity and consistency while maintaining the essential accessibility and flexibility of informal diagramming.

8 CONCLUSIONS

This paper presents a lightweight approach to the incremental formalization of informal architectural diagrams, realized as a Draw.io plugin. Our work targets three challenges common in industrial practice where informal diagrams are a primary design artifact: conceptual ambiguity, notational variability, and adoption friction. The first two arise from working in unconstrained, general-purpose tools, where visuals and meanings are not governed by a shared standard. The third is a contextual barrier rooted in experience with heavyweight modeling tools, where the perceived near-term costs of adoption outweigh uncertain long-term benefits. Our approach addresses these challenges through lightweight typing. The latter makes element types explicit and links each type to a standard visual appearance, which reduces conceptual ambiguity and stabilizes notation. Automatic propagation and on-demand conformance checks keep diagrams aligned without interrupting work. Because everything runs in the familiar tool and formalization is optional, incremental, and reversible, adoption friction is minimized. Looking forward, we plan to empirically validate our approach in industrial settings. We will focus on user studies to measure the effort required for adoption, analyze usage patterns, and quantify the impact of our approach on conceptual clarity and notational consistency. Based on the evidence gathered, we will extend the approach where it demonstrably helps. Possible directions include LLM-assisted support (for example, natural-language querying or guided updates), multi-level or multi-view modeling, and targeted analyses such as change-impact assessment. To broaden participation and gather community feedback, we make the plugin publicly available on GitHub and invite issues, suggestions, and contributions.

ACKNOWLEDGEMENTS

This work was funded by Software Center[2] and Vinnova through the Continuous Digitalization project (2023-00546)[3].

[2]www.software-center.se

[3]www.vinnova.se/en/p/continuous-digitalization/

REFERENCES

Baltes, S. and Diehl, S. (2014). Sketches and diagrams in practice. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 530–541.

Bucchiarone, A., Ciccozzi, F., Lambers, L., Pierantonio, A., Tichy, M., Tisi, M., Wortmann, A., and Zaytsev, V. (2021). What is the future of modeling? *IEEE software*, 38(2):119–127.

Bucchiarone, A., Di Rocco, J., Di Vincenzo, D., and Pierantonio, A. (2025). Modeling in Jjodel: Bridging complexity and usability in model-driven engineering. *arXiv preprint arXiv:2502.09146*.

Cho, H., Gray, J., and Syriani, E. (2012). Creating visual domain-specific modeling languages from end-user demonstration. In *4th International Workshop on Modeling in Software Engineering (MISE)*, pages 22–28. IEEE.

Di Rocco, J., Di Ruscio, D., Di Salle, A., Di Vincenzo, D., Pierantonio, A., and Tinella, G. (2023). jjodel–a reflective cloud-based modeling framework. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 55–59. IEEE.

France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE'07)*, pages 37–54. IEEE.

Guerra, E. and de Lara, J. (2018). On the quest for flexible modelling. In *the 21th ACM/IEEE International conference on Model Driven Engineering Languages and Systems*, pages 23–33.

Hasselbring, W. (2018). Software architecture: Past, present, future. In *The essence of software engineering*, pages 169–184. Springer International Publishing Cham.

Hutchinson, J., Whittle, J., and Rouncefield, M. (2014). Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161.

Izquierdo, J. L. C., Cabot, J., López-Fernández, J. J., Cuadrado, J. S., Guerra, E., and De Lara, J. (2013). Engaging end-users in the collaborative development of domain-specific modelling languages. In *Cooperative Design, Visualization, and Engineering: 10th International Conference*, pages 101–110. Springer.

Jongeling, R., Cicchetti, A., and Ciccozzi, F. (2024). How are informal diagrams used in software engineering? an exploratory study of open-source and industrial practices. *Software and Systems Modeling*, pages 1–13.

Jongeling, R. and Ciccozzi, F. (2024). Flexible modelling: a systematic literature review. *Journal of Object Technology*.

Jongeling, R., Ciccozzi, F., Cicchetti, A., and Carlson, J. (2022). From informal architecture diagrams to flexible blended models. In *European conference on software architecture*, pages 143–158. Springer.

Kaplan, J. and Rabelo, L. (2024). Preliminary studies to bridge the gap: Leveraging informal software architecture artifacts for structured model creation. *Information*, 15(10):642.

Kleppe, A. (2007). A language description is more than a metamodel. In *Fourth International Workshop on Software Language Engineering*, volume 1, pages 1–4.

López-Fernández, J. J., Garmendia, A., Guerra, E., and de Lara, J. (2019). An example is worth a thousand words: Creating graphical modelling environments by example. *Software & Systems Modeling*, 18:961–993.

Mukherjee, D., Dhoolia, P., Sinha, S., Rembert, A. J., and Gowri Nanda, M. (2010). From informal process diagrams to formal process models. In *International conference on business process management*, pages 145–161. Springer.

Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164.

Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., and Heldal, R. (2013). Industrial adoption of model-driven engineering: Are the tools really the problem? In *International Conference on Model Driven Engineering Languages and Systems*, pages 1–17. Springer.

Wüest, D. and Glinz, M. (2011). Flexible sketch-based requirements modeling. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 100–105. Springer.

Wüest, D., Seyff, N., and Glinz, M. (2013). Semi-automatic generation of metamodels from model sketches. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 664–669. IEEE.

Zolotas, A., Kolovos, D. S., Matragkas, N. D., and Paige, R. F. (2014). Assigning semantics to graphical concrete syntaxes. *XM@ MoDELS*, 1239:12–21.

Zolotas, A., Matragkas, N., Devlin, S., Kolovos, D. S., and Paige, R. F. (2019). Type inference in flexible model-driven engineering using classification algorithms. *Software & Systems Modeling*, 18(1):345–366.