From informal architecture diagrams to flexible blended models*

Robbert Jongeling[1 [0000−0002−1863−3987]], Federico Ciccozzi[1 [0000−0002−0401−1036]], Antonio Cicchetti[1 [0000−0003−0416−1787]], and Jan Carlson[1 [0000−0002−8461−0230]]

Mälardalen University, Västerås, Sweden firstname.lastname@mdu.se

Abstract. For the early design and communication of software systems, architects commonly use informal diagrams. Due to their notational freedom and effectiveness for communication, informal diagrams are often preferred over models with a fixed syntax and semantics as defined by a modeling language. However, precisely because of this lack of established semantics, informal diagrams are of limited use in later development stages for analysis tasks such as consistency checking or change impact analysis. In this paper, we present an approach to reconciling informal diagramming and modeling such that architects can benefit from analysis based on the informal diagrams they are already creating. Our approach supports migrating from existing informal architecture diagrams to flexible models, i.e., partially treating diagrams as models while maintaining the freedom of free-form drawing. Moreover, to enhance the ease of interacting with the flexible models, we provide support for their blended textual and graphical editing. We validate our approach in a lab setting and perform an evaluative case study in an industrial setting. We show how the approach allows architects to continue informal diagramming, while also benefiting from flexible models for consistency checking between the intended architecture and the implementation.

Keywords: architecture consistency, software design sketching, blended modeling, flexible modeling.

1 Introduction

Software architects make extensive use of informal diagrams for early design and communication [12]. These diagrams can be used to depict, e.g., the intended decomposition of the system into components, the deployment of components across various hardware systems, or the communication between various parts of the system. The use of free-form drawing for these diagrams allows the ad hoc insertion of new semantic elements and provides freedom of sharing ideas without the inhibition of editing tools or stringent adherence to modeling languages. Being free-formed makes these diagrams easy to adopt and, in many cases, preferred

* This research was supported by Software Center https://www.software-center.se

to models created in modeling tools that enforce their conformity to modeling languages. Thus, informal diagrams are vital tools for architects.

In later phases of development, architects may want to use these previously created informal diagrams to reason about the system. For example, it is commonly of interest to check the completeness of the implementation by comparing the consistency between models of the intended architecture and the implementation. Moreover, such consistency is a prerequisite for using the models for further analysis tasks such as change impact analysis. Their lack of committed semantics prevents informal diagrams from being used for these analysis tasks. Therefore, the question arises: *How can we benefit more from informal early architecture diagrams for analyzing systems under development?*

Based on the prevalence of informal diagrams used for communication between stakeholders, as reported in earlier surveys [19,1], modeling experts have argued that informal modeling (often done in drawing tools and contrasted to more formal and less flexible modeling) can be successful if limitations to their structure are postponed as long as possible [2]. In this study, we consider the point of limiting the structure to come at the moment when informal models are starting to be used for analysis tasks such as consistency checking or change impact analysis. The latter use of sketches is not uncommon; in one of the mentioned surveys, half of the respondents mentioned that sketches were useful to later understand the relationships between development artefacts [1].

Analysis tasks can be better supported by considering development artifacts as models of the system. To bridge informal diagramming and modeling, the concept of flexible modeling proposes to extend modeling tools to support more informal tasks [11]. Generally, flexible modeling is about providing the freedom to deviate from the strict graphical syntax of a modeling language. Blended modeling proposes the seamless use of multiple concrete syntaxes for the reading and editing of models [5].

In this paper, we consider a combination of flexible and blended modeling to provide architects with an approach to benefit from their informal diagrams for analysis tasks. Our approach entails the definition of a grammar for a textual model that shall captures specific aspects of the diagram. A set of model transformations then provides a round-trip synchronization between the informal diagrams, the derived textual model, and back, so that the graphical notation can be updated and can continued to be used after changes in the textual model. We validate our approach in a lab setting and perform an evaluative case study at our industrial partner.

2 Motivation and challenges

In their raw form, informal diagrams lack defined semantics as well as a workable representation and therefore require pre-processing before they can be used for semi-automated analysis tasks such as consistency checking or change impact analysis. Model-based development [18] provides the means to process diagrams in structured and tool-supported ways; by considering them as models of the

system, i.e., abstract representations with a defined syntax and semantics. To maintain the freedom of informal diagramming while also providing the additional benefits of modeling, we aim to consider part of the informal diagrams as models and to preserve the remaining parts of the diagrams as well. In the remainder of this section, we discuss four challenges when we want to migrate from informal diagrams to flexible and blended models.

Knowledge preservation from existing diagrams. The first challenge is to preserve the knowledge captured in a potentially large set of existing informal diagrams. Therefore, the challenge is to boost existing informal diagrams with modeling information or to automatically replace them while maintaining their current content. In both cases, manual migration is infeasible due to the scale and number of existing diagrams.

Adding a textual syntax. Having a textual syntax in addition to the existing graphical one can be beneficial for various tasks such as version control, creating scripts that interact with the textual models, or simply because a textual representation may be the preferable way to edit the model for some stakeholders. Therefore, we identify the need for blended modeling [5], i.e., the use of a textual and graphical representation of the architecture.

Synchronizing graphical and textual syntax. The textual syntax should not replace graphical representations, but should be a supplementary means of manipulating the models. Moreover, both representations shall be editable, and changes to one of them shall be automatically propagated to the other. Synchronization of textual and graphical notations brings additional challenges in flexible modeling settings, since the layout of diagrams must be maintained during synchronization.

Preserving the graphical layout. The layout of informal diagrams may capture the implicit semantics of the design. Therefore, the model should be flexible to include not only semantically relevant modeling information but also implicit information related to the topology of the model, including, e.g., the color, size, location, and rotation of shapes. For example, in a deployment diagram, the placement of boxes relative to each other or the distance between them may convey some meaning to architects about the relative grouping of components. Not all of these implicit semantics are meaningful to capture in a modeling language, nor are they easily formalized. In other modeling formalisms, e.g., UML class diagrams, the exact arrangement of the classes carries no semantics (although it can still be relevant for communication). However, for informal diagrams, these implicit semantics are key, especially since diagrams are used mainly for communication between architects and other stakeholders. Hence, while it may be hard to make these implicit semantics explicit, the layout should be preserved so that the intended meaning of informal diagrams is not lost.

3 Flexible and blended modeling of architectures

In this section, we propose an approach that meets the challenges identified in Section 2.

3.1 Approach overview

Our approach boosts existing informal diagrams by considering parts of them as models. To do so, we define specific types of graphical elements (shapes) in the informal architecture diagram as metamodel elements, and thereby all occurrences of those shapes as model elements. Furthermore, we propose two unidirectional transformations that (i) create a textual model from an informal diagram, and (ii) restore a graphical representation based on the textual model, preserving the layout of the pre-existing informal diagrams. Hence, parts of the informal diagram that are not defined as model elements are not included in the textual model, but are restored when transforming the textual model back to a graphical representation. This approach allows architects to continue with informal diagramming, also after the definition of the model. An overview of the transformations and other parts of the approach is shown in Fig. 1. In the following, we include a detailed description of the approach, in which we distinguish between the blended modeling loop and the preparation steps that define the textual and graphical formats.

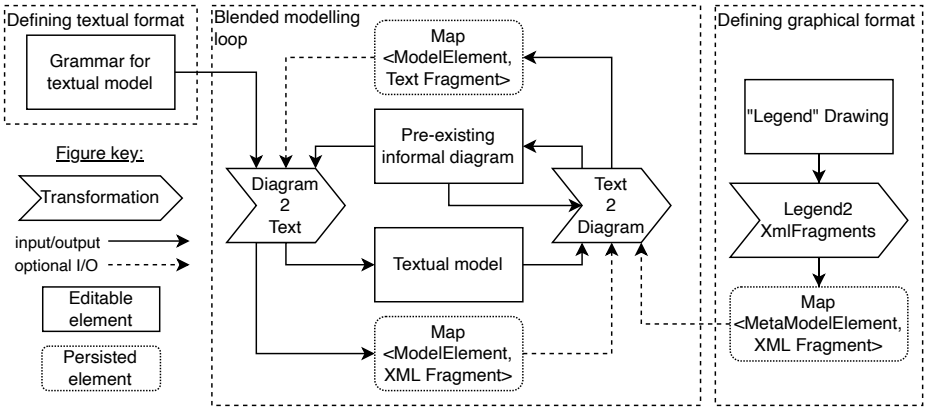

Fig. 1. Schematic overview of our flexible and blended modelling approach.

3.2 Defining textual and graphical formats

The first step of the approach is to define a grammar to which the textual model of the architecture diagram will conform. To do so, the architects should identify those aspects of the diagrams they have created that are of interest for analysis. The grammar should contain concepts for each type of element in the diagram that will be considered a metamodel element. The textual model is obtained through the transformation *Diagram2Text*, which in the first iteration of the blended modeling loop only takes as input the grammar and the informal diagram.

To define the transformation *Diagram2Text* , it is necessary to define which shapes of the diagram are to be considered as metamodel elements. This information can be directly included in the transformation or fed to the tool by means of a "legend", which is a separate diagram in which the metamodel elements are defined by example. For example, if all yellow rectangles in a diagram are to be considered as instances of the metamodel element "component", then the legend should contain a single yellow rectangle named "component". Then, using the transformation *Legend2XmlFragments*, we create a map of the metamodel element to the XML fragment, in this case from the metamodel element "component". The legend is used to transform newly added elements in the textual representation to a graphical representation, in transformation *Text2Diagram*. Conversely, if no new elements are defined in the textual model, the entire diagram can be reconstructed based on the map of the model element to the XML fragment stored during transformation *Diagram2Text*.

As we will see in Section 4, transformations *Diagram2Text* and *Text2Diagram* are currently implemented manually. This is a limitation of the implementation, but not of the approach. In future work, our aim is to automatically generate the transformations inside the blended modeling loop provided a grammar for the textual model and a legend drawing.

3.3 Blended modeling loop

The middle part of Fig. 1 labeled "Blended modeling loop" refers to the synchronization between the informal diagram and the derived textual model. We follow a cycle of the loop starting from the box "Pre-existing informal diagram". Given the initial diagram and the grammar, the transformation *Diagram2Text* results in (i) a textual model of the architecture, containing all the model elements identified in the diagram; and (ii) a map of the model elements to XML fragments, containing for each of the identified model elements an XML fragment containing their corresponding graphical representation as obtained from the diagram file.

After changes to the textual model, we can reconstruct the diagram from the textual model by transformation *Text2Diagram*. Each model element in the textual model is expanded to its graphical representation using either (i) the XML fragment stored during the previous transformation *Diagram2Text* or (ii) the XML fragment as a result of transformation *Legend2XMLFragments*. The latter option is used for elements that are newly added or changed in the textual representation and therefore do not occur in the stored map of model elements to XML fragments. Therefore, although both inputs corresponding to these options are marked as optional in Fig. 1, exactly one of them must be used to expand each model element. Thus, *Text2Diagram* makes use of the preserved layout information associated with the model elements during transformation *Diagram2Text*. Furthermore, transformation *Text2Diagram* also restores the remainder of the diagram, i.e., those shapes that were not considered model elements. To do so, it replaces in the original diagram only those shapes that were identified as model elements in the previous step and leaves the remaining shapes unaltered.

To some extent, the layout information of the textual representation is preserved as well; the *Text2Diagram* transformation produces a map that contains an associated text fragment for each model element, which includes all the text after the previously occurring model element and until the model element itself. Hence, all comments related to an element are preserved and are later restored when re-running the *Diagram2Text* transformation. This input is marked as optional because, during the first time the transformation is executed, no such text fragment information is known, and therefore, is not used.

The approach is thus flexible, as it allows the user to continue to combine modeling and drawing by preserving the existing diagram layout in its entirety, except for the added, modified, or deleted model elements. Moreover, synchronization between textual and graphical representations makes the approach *blended*. Thus, we have obtained flexible and blended models from informal architecture diagrams.

4 Implementation[1] and validation

In this section, we describe our implementation, validate it by showing how the textual and graphical representations can be automatically synchronized while preserving their layout, and show how architects in our industrial setting benefited from our approach for establishing consistency checks between the architecture model and the implementation.

4.1 Implementation and validation in lab setting

To demonstrate the approach, we implemented it by boosting the functionality of the drawing tool `diagrams.net` with a grammar to capture some aspects of pre-existing informal diagrams in textual models. We now show an example architecture diagram, a corresponding grammar, and transformations for round-tripping between the diagram and the textual model.

Example architecture diagram An example architecture diagram is shown in Fig. 2. The example is an extended version of the architecture diagram encountered in the industrial setting discussed in Section 4.2. The diagram shows several layers that represent hardware, drivers, services, and others. The different columns roughly represent groupings of functionality. In addition, the diagram contains the components and their position within the layers and columns. Components are distinguished between in-house developed components (yellow) and third-party components (blue). Dependencies between components are indicated by means of dashed arrows. The example is an anonymized (by removing the

[1] The reader is encouraged to have a look at our replication package with our implementation and demo videos, in the following GitHub repository: `https://github.com/RobbertJongeling/ECSA-2022-replication-package`.

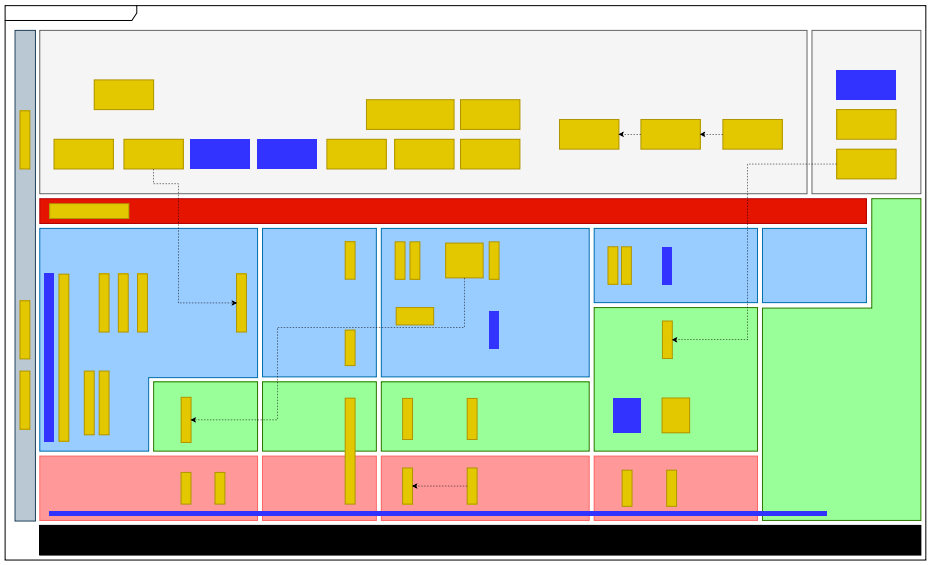

Fig. 2. Example anonymized informal diagram of an instance architecture.

names of layers and components) and extended (by adding dependencies) version of a real architecture diagram of the company that could not be shared for intellectual property reasons.

Defining a grammar We aim to capture three aspects of the diagram: layers, components as residing in layers, and dependencies between the components. In this implementation, we create the grammar shown in Listing. 1.1 using TextX [6], which is based on Xtext [21], but does not rely on the Eclipse ecosystem; instead, it allows for grammar specification in Python. The choice of Python is motivated from an industrial perspective to ease the adoption of the approach.

Listing 1.1. Grammar specification for textual model, expressing an architecture in terms of components, dependencies, and layers.

```
Architecture:
    ('components' '{' components+=Component ( "," components
        ↪ +=Component)* '}' )
    ('dependencies' '{' dependencies+=Dependency ( ","
        ↪ dependencies+=Dependency)* '}' )
    ('layers' '{' layers+=Layer ( "," layers+=Layer)* '}' )*
;
Component:
    'component' name=ID
    ('inlayers' '{' layers+=[Layer] ( "," layers+=[Layer])* '
        ↪ }' )*
```

```
;
Dependency:
    ('from' fromcomp=[Component])
    ('to' tocomp=[Component])
;
Layer:
    'layer' name=ID
;
//lines starting with # are comments
Comment:
    /\#.*$/
;
```

The grammar is limited to the properties of the architecture that we want to model. Additional properties of the components could also be included in the model, e.g., whether they are third-party, open-source, or in-house developed. In this example, we consider that this additional information included in the diagram is not needed in the model and show that it is nevertheless preserved in the blended modeling loop.

Creating a "legend" drawing The legend contains shapes for all metamodel elements that occur in the grammar. Fig. 3 shows the legend drawing that is used as input to derive the shapes of the metamodel elements "layer", "component", and "dependency". Since the approach is based on the XML representation of the diagram, a requirement when using multiple shapes is that they have a different shape definition (e.g., rectangles and rounded rectangles could have the same "shape" value and would thus be indistinguishable). In the example legend, the element "layer" is defined as a polygon, the component and the third-party component are both rectangles, but with different styles.

Transforming diagrams to text and back The transformations, like the grammar, are also implemented in Python. To process the diagram, the first

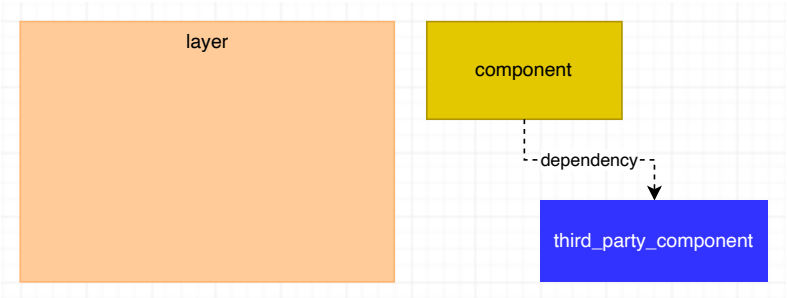

Fig. 3. "Legend" defining the shapes of the metamodel elements *layer*, *component*, and *dependency*.

step of the transformation *Diagram2Text* obtains a readable XML string from the persisted format. To create the textual model, first all shapes that denote model elements are collected, and then relationships between model elements are identified. During generation of the textual model, it is checked if there is an existing map of model elements to textual fragments that was created during a previous iteration of the blended modeling loop. If so, the textual representation of these model elements is restored; if not, they are newly created based on the grammar. When creating the textual model, for each model element, the XML fragment corresponding to its graphical representation is stored.

To recreate the diagram, the transformation *Text2Diagram* replaces all the identified model elements in the existing diagram with those model elements of the textual model. If they already exist, then their graphical representation is taken from the map of model elements to XML fragments. For new components, layers, and dependencies, we rely on the legend to derive the representation of the component. In the current implementation, the legend also determines the position of the new element; hence the new elements are rather crudely placed in the diagram with no regard to the pre-existing layout. This is an implementation detail and not a limitation of the approach.

To validate the transformations, we performed a test for a scenario in which we work with the example diagram. First, we run the transformation *Diagram2Text*. Second, we removed a component, changed a dependency, added a new component, and added a new layer in the textual model. We then executed the transformation *Text2Diagram*. We successfully obtained the new elements and maintained the layout of the unchanged elements. We then moved around elements in the graphical layout and re-run the transformation *Diagram2Text*, we observe in the textual model that the "inlayers" attribute of the moved component has been updated to reflect its current position. A demo video of this procedure is included in the replication package linked at the beginning of this section.

4.2 Evaluative case study

In addition to the above validation on a constructed example, we evaluated our approach by implementing it in a concrete industrial setting and using the textual model for consistency checking between an intended architecture and a corresponding implementation.

Industrial setting We collaborated with a group of software architects who design and maintain the software architecture for several variants of embedded systems developed in their company. The group has created a reference architecture that is used as a template for deriving the architecture of new products. In practice, architecture descriptions are drawings similar to UML deployment diagrams, but are instead informal diagrams created in Microsoft Visio. When a new product is created, the reference architecture diagram is copied and components are deleted or added as required for that particular product, analogously

to *clone-and-own* practices common in the software product line engineering domain.

The created diagrams are similar to the one shown in Fig. 2. However, since the architects are only interested in checking the consistency between included components between the architecture and implementation, we ignore dependencies, the origin of components (third-party or in-house developed) and what layers they belong to.

Need for consistency checking In the studied setting, informal diagrams, such as the one shown in Fig. 2, are used mainly for communication. The ability to edit them freely provides architects with a highly accessible way to make changes. However, now the group has run into the limitations of these diagrams, since the desire has arisen to check the consistency between specific instance architectures and their implementations. Consistency is relevant, since throughout evolution of the system, features and software components may be added, and thereby the instance architecture diagram might go out of date, making it no longer a suitable artifact to use for reasoning about the system.

Envisioned way of working We discussed with architects what way of working could be adopted to facilitate consistency checks between their intended architecture and the corresponding implementation. The following steps were seen as a typical scenario for creating an instance architecture. Step 1 is to duplicate the reference architecture and in the new diagram remove, edit, and add software components to customize an instance architecture. Step 2 is then to create a list (textual representation) of the software components in the instance (this is automated by transformation *Diagram2Text* in our approach). Step 3 is to analyze consistency by checking that the components included in the instance architecture and those included in the implementation are the same. Once defined, the consistency check can be scripted and executed repeatedly throughout the system's evolution.

Consistency check implementation To check the consistency between the architecture diagrams and their implementation, we compare the components included in both. Currently, there are no explicit links between the code and the diagram. In this setting, the architecture and implementation should be considered consistent if they include the same components. Hence, to create a consistency check for this setting, it is required to obtain from the diagram the components it depicts and to obtain from the software implementation a list of the components it contains. Fig. 4 shows an overview of the final consistency check implemented.

In Script A (in Fig. 4), a list of components is extracted from the software configuration files, based on several rules that determine whether or not an entry should be considered a component. These rules were defined after discussions with the group of architects and a few iterations to narrow down an exact definition. To extract components from the architecture diagram, we first applied our

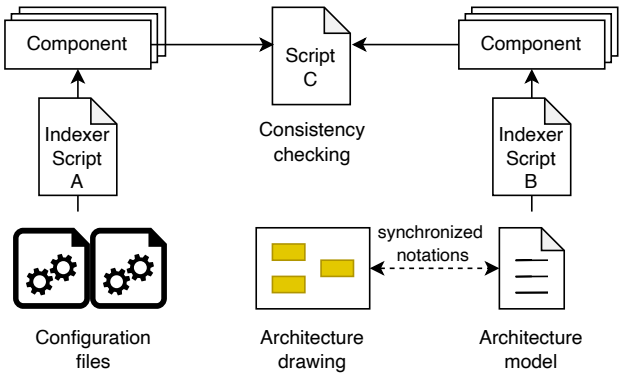

Fig. 4. Consistency checking source code and architecture by first extracting sets of components from configuration files (Script A) and textual model (Script B) as obtained from the informal architecture diagram, and then comparing these sets (Script C).

approach as outlined in Section 3 to enhance the informal architecture diagram with modeling information. We define a grammar that expresses the architecture as a set of components (a simplified version of the grammar shown in Listing 1.1) and use our approach to transform the diagram into a textual model conforming to the grammar. In Script B (in Fig. 4), we then extract the names of all the components by obtaining all "component" model elements from the textual model file.

Script C (in Fig. 4) then compares the sets of components obtained from the configuration files and the textual architecture model. When executing consistency checks in the industrial setting, we identified about 50 components in the architecture and about 40 components in the implementation. According to the architects, this discrepancy is due to an inaccurate model. We found that the model included more boxes marked as components than boxes that should be considered as components. Therefore, in this case, the consistency check provided feedback on the quality of the modeling, rather than on the implementation.

4.3 Experiences from the studied industrial setting

To evaluate our implementation, we have asked the architects of the group to use our tool for a typical task for them: deriving a new instance architecture and modifying it. The architects have copied the existing graphical reference architecture diagram and derived the textual representation using our tool. The textual representation was then modified by removing and adding components. Finally, the architects recreated a graphical representation of the model, where all components had the same layout as before (except for the newly created components, which are placed at a default location in the diagram).

The architects could use our implementation and agree that the functionality meets their requirements as listed in Section 2. As we showed, the implementation allows for the creation of automated consistency checks between the

architecture and implementation due to (i) capturing the semantics of the informal diagram and (ii) providing an accessible format for the obtained model elements. Finally, we allow the architects to keep using the graphical format they are used to and the topology contained within existing diagrams by means of the synchronization transformations in the blended modeling loop. One of the main benefits the architects experienced was that our approach allowed them to implement consistency checks and thereby showcase the benefits of modeling to their colleagues.

5 Discussion

5.1 When to use this approach

Our approach provides the following benefits to architects:

1. a textual model capturing aspects of the informal diagrams that are of interest for analyzing the system under development;
2. the ability to continue informal diagramming, supported by synchronization of the textual model and the informal diagram that preserves its layout; and
3. the ability to only partially model the architecture, since not all aspects of the diagram have to be included in the textual model.

Our approach has value when applied in settings with pre-existing architecture diagrams. In such settings, it may not be straightforward to migrate those diagrams to models with strict semantics. Then, a flexible approach allows the continued use of informal diagrams mixed with some stricter semantics for certain aspects of the diagrams. Moreover, the blended modeling of our approach allows one to maintain the layout of the diagrams, which is useful, e.g., in those cases where the layout conveys implicit semantics.

Our intention is to provide a small degree of modeling in settings in which diagrams are used so that the benefits of modeling can be shown. We showed that the approach can support complex graphical model elements, such as connections and hierarchies. However, supporting intricate informal diagrams that contain many of these complex constructs would require significant effort in the customization of the transformations to appropriately map these (connected) shapes to model elements. Therefore, for more complex graphical needs, it would still be a better option to use an existing modeling language (and the tools that support it) or to develop a DSML in a language workbench. In the evaluative setting, the approach sufficiently captures the model, since we are only interested in capturing the components as model elements.

5.2 Approach limitations

We identified several limitations of our approach that were acceptable in the studied industrial setting but may restrict its applicability in other settings.

To be able to use the approach, architects should be able to capture in a grammar the concepts from their drawing that are of interest for analysis. This task

becomes more challenging with more complex diagrams. Moreover, there might be cases where capturing the semantics of the drawing is made more difficult because the same notation is used to represent different concepts, in which case our approach cannot distinguish them. Nevertheless, when we consider grammar as a formalization of the ideas currently implicit in the architecture, we believe that architects should be able to perform this task.

A related challenge is scaling up the approach to deal with larger and more complex informal diagrams. In our example, the choice of components, layers, and dependencies was made to show that the approach can handle at least three commonly used graphical representations of elements: (i) stand-alone shapes, (ii) containment of shapes in other shapes, and (iii) connectors between shapes. Validating that we can continue to differentiate between different concepts in more complex diagrams remains a task for future work.

The nature of this approach is that the emerging workflow is not very robust to mistakes. For example, using unknown shapes in the diagram will cause them to not be recognized as model elements by the transformations. A way to address this weakness could be to provide custom shape sets as a way to provide a palette for the graphical syntax. In future work, we aim to study these and other means to provide robust ways to combine informal diagramming and modeling.

Our current implementation is based on `diagrams.net`, but our approach is not limited to it. The approach relies on matching model elements with fragments of their graphical representation, which is not limited to the XML-based persistence format of one particular tool. Supporting other drawing tools is a direction for future work.

5.3 Considered alternative approaches

Initially, we considered providing architects with a domain-specific modeling language (DSML) that would completely replace their need for informal drawing. To this end, we defined a grammar using Xtext [21] and created a graphical syntax for it using Sirius [8]. The intention was to maintain the graphical syntax that the architects were already using in their drawings, but it turned out to be too challenging to replicate some specific aspects of the existing graphical notation when developing this DSML. More problematically, the resulting tool was considered too heavyweight to be easily adopted in the studied industrial setting, since using the developed DSML requires architects to use an Eclipse instance with some plug-ins. Finally, we rejected the alternative of developing a new DSML following the realization that architects prefer to be able to maintain their current informal modeling practices.

We also looked for alternatives for blended architecture modeling and considered PlantUML. Although it is easy to define the components in its textual notation, positioning them in the graphical notation is not supported by design. Indeed, PlantUML's GraphViz-based generator creates the graphical view of the model, and the user should not want to control the relative positioning of elements too much.

5.4 Other threats to validity

External validity is related to the generalizability of the findings [16]. Because of the ubiquity of informal diagrams for architecture and the common desire to use them for analysis tasks, we see a broad applicability of our approach. Still, we need to be careful when claiming that our findings with respect to the usefulness of our approach are general. Future work is required to analyze the suitability of our proposed blended and flexible modeling to other problems in other settings.

6 Related Work

Sketching software architectures has a long history. Tools have been proposed that limit the interpretation of skecthes into existing diagrams, e.g., converting sketches of UML use case diagrams to proper models. Further sketching tools have provided more freedom and other usability features to support sketching for software design [15]. Our approach extends these by providing the possibility to treat the diagram as a model.

Among the existing approaches to textually describe models, there are a large number of text-to-UML tools such as PlantUML [3]. What is typically not supported in these tools is controlling the topology of graphical models, since this is not always relevant for the semantics of the model. However, there are many situations where the layout of the model also conveys semantics [7], such as in our studied setting, where the arrangement of the components has meaning for the architects.

The flexible modeling paradigm [11] can be approached from two directions. Researchers have also begun exploring the support of informal notation in existing modeling tools, e.g., to supplement UML and OCL models [10]. The other direction, studied in this paper, is to consider to what extent we can include formal modeling inside an existing informal diagramming tool. Similar initiatives have been undertaken; for example, it was shown how DSMLs can be created from graphical examples [14] in order to involve domain experts in DSML development. Our approach is able to additionally consider the non-modeling elements of the drawing and maintain its overall layout during a round-trip from graphical to textual notation and back. Additionally, FlexiSketch was developed, which is a tool that allows the user to first create a free-form sketch and then create a metamodel by annotating certain shapes in sketches with metamodel elements [20]. In our approach, we propose to benefit from a pre-existing mature and commonly used drawing tool as the source of informal diagrams and, in addition, provide a textual notation for flexibly created models.

Our solution requires manual specification of the metamodel elements. Another approach has shown the possibility of deriving metamodels by automatically selecting candidate elements and letting the user judge the candidates [4]. A further difference between common flexible modeling approaches and ours is that we do not infer types for later models, but instead boost an existing drawing with modeling concepts. Treating drawings as models has previously been explored,

by transforming annotated drawings into an intermediate model representation that can be interacted with using Epsilon model management tools [13,22]. Our approach is more flexible than bottom-up metamodeling [17] or metamodeling by example, since it allows for continued mixing of drawing and modeling, even after the instantiation of the underlying metamodel.

Another proposal for a flexible interpretation of icons for a concrete graphical syntax relies on associating model elements with snippets of vector graphics that represent them [9]. However, the approach does not provide a round trip to a textual notation and does not explicitly consider the layout of the graphical model.

In summary, our approach adds to the existing literature by allowing the migration from informal diagrams in existing drawing tools to flexible textual models that can be used for analysis, while maintaining the freedom of drawing by providing synchronization between the informal diagram and textual model.

7 Conclusion

In this paper, we showed an approach to benefit more from existing informal architecture diagrams by considering them, in part, as models. We describe our implementation and validate it in a lab setting. Additionally, we conducted an evaluative case study in an industrial setting where flexible models are used for consistency checking between the architecture and the implementation, while informal diagrams are continued to be used for communication between architects and other stakeholders. Our approach creates flexible models out of the diagrams; since it requires only their partial consideration as models, other aspects of the diagram can remain in free form. Moreover, our approach creates blended models due to our synchronization mechanism between the diagrams and textual models. The benefit of this approach is that users may continue using the informal graphical notations they were used to, but, in addition, they can benefit from the syntax and semantics brought by modeling. In future work, we plan to study how we can implement flexible and blended modeling in more robust and general ways for use cases where it can be useful.

References

1. Baltes, S., Diehl, S.: Sketches and diagrams in practice. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 530–541 (2014)
2. Bucchiarone, A., Ciccozzi, F., Lambers, L., Pierantonio, A., Tichy, M., Tisi, M., Wortmann, A., Zaytsev, V.: What is the future of modeling? IEEE Software **38**(2), 119–127 (2021)
3. Cabot, J.: Text to UML and other "diagrams as code" tools – Fastest way to create your models (March 2020), `https://modeling-languages.com/text-uml-tools-complete-list/`

4. Cho, H., Gray, J., Syriani, E.: Creating visual domain-specific modeling languages from end-user demonstration. In: 2012 4th International Workshop on Modeling in Software Engineering (MISE). pp. 22–28. IEEE (2012)
5. Ciccozzi, F., Tichy, M., Vangheluwe, H., Weyns, D.: Blended modelling-what, why and how. In: 2019 ACM/IEEE 22nd MODELS-Companion. pp. 425–430. IEEE (2019)
6. Dejanović, I., Vaderna, R., Milosavljević, G., Vuković, Ž.: TextX: A Python tool for Domain-Specific Languages implementation. Knowledge-Based Systems **115**, 1–4 (2017)
7. Di Vincenzo, D., Di Rocco, J., Di Ruscio, D., Pierantonio, A.: Enhancing syntax expressiveness in domain-specific modelling. In: 2021 ACM/IEEE MODELS Companion. pp. 586–594. IEEE (2021)
8. Eclipse Foundation: Sirius - the easiest way to get your own modeling tool (2022), `https://www.eclipse.org/sirius/`
9. Fondement, F.: Graphical concrete syntax rendering with svg. In: European Conference on Model Driven Architecture-Foundations and Applications. pp. 200–214. Springer (2008)
10. Gogolla, M., Clarisó, R., Selic, B., Cabot, J.: Towards facilitating the exploration of informal concepts in formal modeling tools. In: 2021 ACM/IEEE MODELS-C. pp. 244–248. IEEE (2021)
11. Guerra, E., de Lara, J.: On the quest for flexible modelling. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS). pp. 23–33 (2018)
12. Hasselbring, W.: Software architecture: Past, present, future. In: The Essence of Software Engineering, pp. 169–184. Springer, Cham (2018)
13. Kolovos, D.S., Matragkas, N.D., Rodríguez, H.H., Paige, R.F.: Programmatic Muddle Management. XMMoDELS **1089**, 2–10 (2013)
14. López-Fernández, J.J., Garmendia, A., Guerra, E., de Lara, J.: An example is worth a thousand words: Creating graphical modelling environments by example. Software & Systems Modeling **18**(2), 961–993 (2019)
15. Mangano, N., Baker, A., Dempsey, M., Navarro, E., van der Hoek, A.: Software design sketching with calico. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. pp. 23–32 (2010)
16. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering **14**(2), 131 (2009)
17. Sánchez-Cuadrado, J., Lara, J.d., Guerra, E.: Bottom-up meta-modelling: An interactive approach. In: International Conference on Model Driven Engineering Languages and Systems. pp. 3–19. Springer (2012)
18. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer **39**(2), 25 (2006)
19. Störrle, H.: How are conceptual models used in industrial software development? a descriptive survey. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering. pp. 160–169 (2017)
20. Wüest, D., Seyff, N., Glinz, M.: Flexisketch: a lightweight sketching and metamodeling approach for end-users. Software & Systems Modeling **18**(2), 1513–1541 (2019)
21. Xtext: Xtext – language engineering made easy! (2022), `https://www.eclipse.org/Xtext/`
22. Zolotas, A., Kolovos, D.S., Matragkas, N.D., Paige, R.F.: Assigning semantics to graphical concrete syntaxes. XM@ MoDELS **1239**, 12–21 (2014)